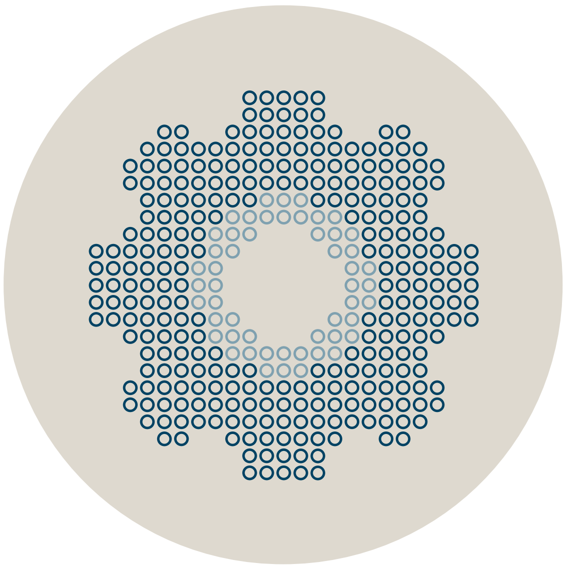


SMC API User's Guide

SMC API User's Guide 5.6.0

Created: January 14, 2014

STONESOFT
A McAfee Group Company

Table of Contents

Introduction 5

About This Guide 5

Audience 5

When to Use the SMC API? 5

Supported Use Cases 6

Security and Authentication 6

Backward Compatibility 7

Configuring SMC API in the Management Client 7

Enabling SMC API on the Management Server 7

Creating a New API Client Element 9

Defining API Client Permissions 10

Defining Permissions for Restricted API Client Accounts 11

Allowing SMC API Connections 12

RESTful Principles 13

Requests 13

Status Codes and Error Messages 14

ETags 14

Representations 14

Opaque URIs, URI Discovery, and Hypermedia 15

Body Content and Query Parameters 15

Entry-Point 16

Verbs 16

API Discovery 19

Data Elements 21

Collections 24

Searching for Resources 24

Retrieving a Resource 25

Creating a Resource 26

Updating a Resource 26

Deleting a Resource 26

Other Actions 27

Specific Searches 27

System Information 27

Examples 28

Version-Specific Entry-Point 28

Global Services and Element URIs 29

Logging in 30

Listing the Hosts Collection 30

Filtering Elements 31

Creating a New Host 31

Attempting to Create an Element with an Existing Name 32

Modifying an Existing Host 32

Deleting a Host 34

Modifying a Rule in a Policy 34

Performing and Following Up on a Policy Upload 36

Search Filtering by Group Type 38

Retrieving Routing/Antispoofing Information 38

Introduction

In addition to the Management Client user interface, the Stonesoft Management Center (SMC) can be accessed through an Application Programming Interface (API). The SMC API gives access to all the data in the system and enables API clients to manage SMC elements.

The purpose of the SMC API is to allow create, read, update, and delete operations as well as more complex actions, such as policy uploads, on all elements in the SMC.

About This Guide

This guide gives an overview of how the SMC API is used, with examples for a subset of accessible resources. The reference list of accessible resources and actions that can be performed on them is available in the complete generated API documentation at http://help.stonesoft.com/smc_api/5.6.0/.

Although the SMC API follows the architectural style of RESTful web APIs, this guide only introduces the basic concepts. Refer to http://en.wikipedia.org/wiki/Representational_state_transfer and the linked content for more information.

Audience

The target audience for this guide includes system administrators and developers. The Introduction also provides useful information to people who are interested in the possibilities of integrating the Stonesoft Management Center.

When to Use the SMC API?

There are a number of scenarios in which the SMC API can be useful:

- The SMC API makes it possible to integrate the SMC with third-party policy management and risk management applications. The SMC API is already used by vendors like Tufin and FireMon.
- The SMC API provides the necessary tools for managed security service providers (MSSPs) to include functions related to Stonesoft products on their own web portals.
- The SMC API allows you to automate frequent tasks through scripting without administrators manually configuring them in the Management Client.
- The SMC API allows the development of an alternative user interface for managing Stonesoft products.

Supported Use Cases

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 (Hosts, Networks, Address Ranges, etc.)
- Adding, editing, and removing rules that belong to an Access rule, NAT rule, or Inspection Policy.
- Uploading a policy to an engine
- Retrieving or changing the routing of an engine

The SMC API does not enable the editing of all the elements in the SMC or running all the actions that are available in the Management Client. For more information on how these use cases are supported, see the complete generated API documentation at http://help.stonesoft.com/smc_api/5.6.0/.

Security and Authentication

It is recommended to use HTTPS to secure access to the SMC API. Enable HTTPS by selecting **Server Credentials** on the **SMC API** tab in the Management Server properties. For more details on the SMC API tab configuration, see Enabling SMC API on the Management Server.

Authentication

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

In order to save server resources, clients should log out at the end of the session.

Sessions

The SMC API supports two ways of associating all requests with a single user between the login and logout actions:

1. Cookies: Clients are responsible for sending back in each request all the (non-expired) cookies that were sent by the server.
2. SSL Sessions: Sessions are tracked by the server based on SSL connections.

By default, cookies are used. If you want to use SSL Sessions instead, you must enable the **Use SSL for Session ID** option on the **SMC API** tab in the Management Server properties. For more details on the SMC API tab configuration, see Enabling SMC API on the Management Server.

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

It is possible to modify the inactivity timeout by adding the configuration parameter (WEB_SERVER_SESSION_TIMEOUT=X, where “X” is minutes) to the SGConfiguration.txt file, which can be found in the <*installation directory*>`/data` folder.

Backward Compatibility

Backward compatibility is only guaranteed back to the previous major SMC release.

For example, SMC 5.7 provides access to both the 5.6 and 5.7 versions of the SMC API, from two version-specific URIs. We do not guarantee that SMC 5.8 will support SMC API version 5.6.

Configuring SMC API in the Management Client

You must enable the Application Programming Interface for the Stonesoft Management Center in the Management Client. You can do this in the properties of the Management Server that handles the requests from the external applications or scripts.

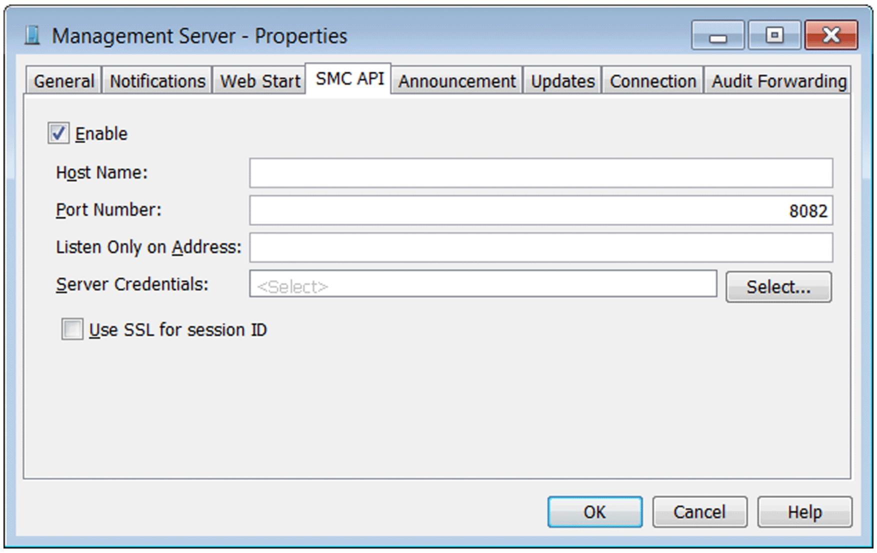
The API clients that use the SMC API must also be defined in the Management Client and given the appropriate permissions. You can define the API clients and their permissions using API Client elements.

In addition, you must allow SMC API connections from the IP addresses of the API clients to the Management Server.

Enabling SMC API on the Management Server

To Enable SMC API on the Management Server

1. Right-click the Management Server and select **Properties**. The Management Server Properties dialog opens.
2. Switch to the **SMC API** tab.



☑ Enable

Host Name:

Port Number: 8082

Listen Only on Address:

Server Credentials: <Select>

☐ Use SSL for session ID

3. Select **Enable** to activate the SMC API.

4. (*Optional*) Enter the **Host Name** that the SMC API service uses.

5. (*Optional*) Change the (TCP) **Port Number** that the SMC API service listens to.

   - By default, port 8082 is used.
   - In Linux the value of this parameter must always be higher than 1024.

Note - Make sure the listening port is not already in use on the server. For information on the ports that are reserved for Stonesoft system services, refer to the Management Client *Online Help* or the *Stonesoft Administrator's Guide*.

6. (*Optional*) If the Management Server has several addresses and you want to restrict access to one address, specify the IP address to use in the **Listen Only on Address** field.

7. (*Optional*) Click **Select** and select an existing Server Credentials element or create a new Server Credentials element.

8. (*Optional*) Select **Use SSL for Session ID** to track sessions to the Management Server in your application. Do not select the option if your network requires you to use cookies or URLs for session tracking.

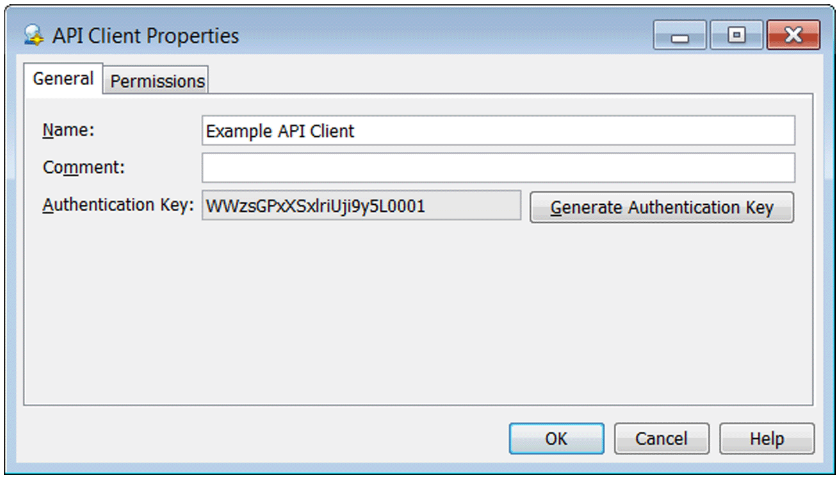
9. Click **OK**.

Creating a New API Client Element

To Create a New API Client Element

1. Select **Configuration→Configuration→Administration**. The Administration Configuration view opens.

2. Right-click **Access Rights** and select **New→API Client**. The API Client Properties dialog opens.



Name: Example API Client
Comment:
Authentication Key: WWzsGPxXSxlriUji9y5L0001

3. Give the API Client a unique **Name**.

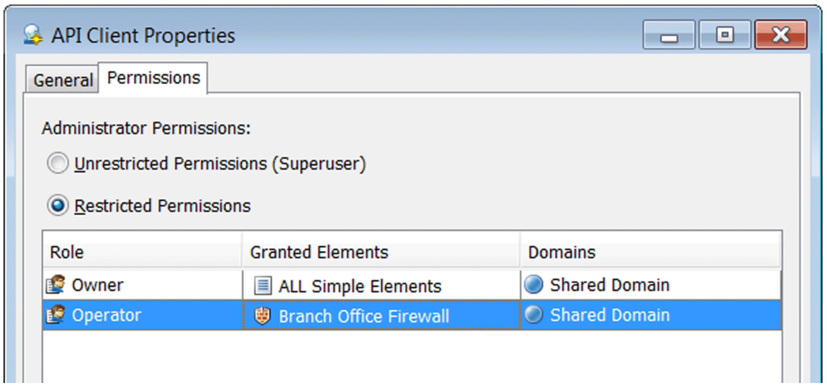
4. (*Optional*) Click **Generate Authentication Key** to generate a new Authentication Key. A random 24-digit alphanumeric authentication key is automatically generated.

   - The API Client uses the authentication key to log in to the SMC API.

Defining API Client Permissions

To Define an API Client's Permissions

1. Switch to the **Permissions** tab in the API Client element's properties.


API Client Properties

General | Permissions

Administrator Permissions:

( ) Unrestricted Permissions (Superuser)

(●) Restricted Permissions

Role	Granted Elements	Domains
Owner	ALL Simple Elements	Shared Domain
Operator	Branch Office Firewall	Shared Domain

2. Select one of the following options:

   - **Unrestricted Permissions (Superuser)**: The API Client can manage all elements and perform all actions without any restrictions.
   - **Restricted Permissions**: The API Client has a limited set of rights that apply only to the elements granted to the administrator.

Caution - Select only the minimum necessary permissions for each API Client account.

What's Next

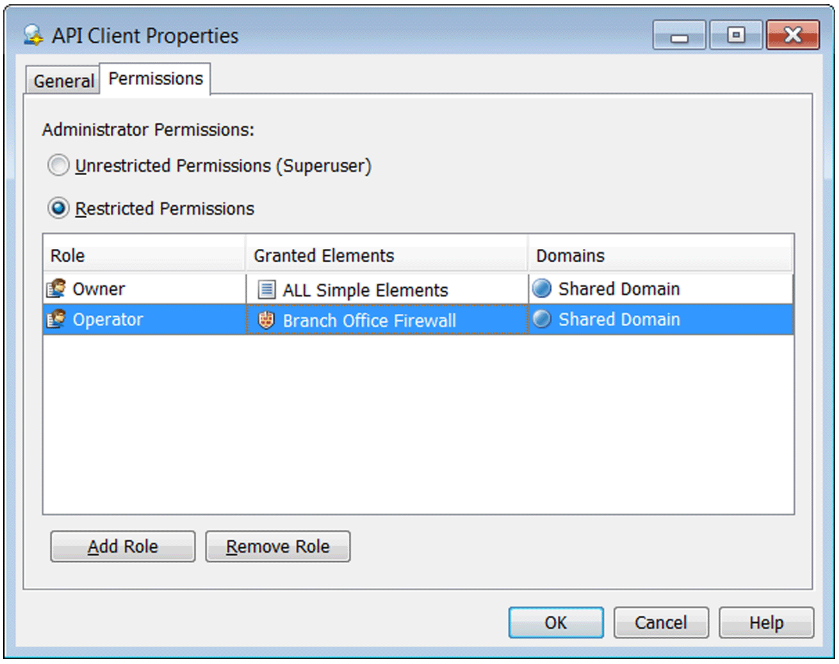
- If you selected **Restricted permissions** for this API Client, proceed to Defining Permissions for Restricted API Client Accounts.
- Otherwise, click **OK**. The API Client account is ready for use.

Defining Permissions for Restricted API Client Accounts

To define the permissions in detail as explained below, the API Client element must have **Restricted Permissions** selected as the API Client permissions level.

To Define Administrator Permissions for a Restricted API Client Account

1. On the **Permissions** tab of the API Client Properties dialog, click **Add Role**. A new Administrator Role appears in the list above.



Role	Granted Elements	Domains
Owner	ALL Simple Elements	Shared Domain
Operator	Branch Office Firewall	Shared Domain

2. Click the **Role** cell and select the Administrator Role that defines the rights you want to set. In addition to any customized roles, there are four predefined Administrator Roles:

   - **Operator**: Can execute GET requests on selected elements in order to retrieve the elements' properties. Can refresh and upload policies.
   - **Editor**: Can execute GET/POST/PUT/DELETE requests on selected elements in order to create, edit, and delete the elements. Can refresh and upload policies.
   - **Owner**: Can execute GET/POST/PUT/DELETE requests on selected elements in order to create, edit, and delete the elements.
   - **Viewer**: Can execute GET requests on selected elements in order to retrieve the elements' properties.

3. Double-click the **Granted Elements** cell and select the elements to which the rights granted by the Administrator Role apply.

    - The **Set to ALL** action depends on the type of elements. For example, if you browse to Firewalls and click **Set to ALL**, the item “All Firewalls” is added.
    - You can also select one or more predefined or user-created Access Control Lists. “Simple elements” includes all elements except elements that have a dedicated system Access Control List (for example, there are dedicated Access Control Lists for different types of security engines and their policies). See the *Stonesoft Management Center Reference Guide* for a description of all the pre-defined Access Control lists.

4. (*Optional*) If Domain elements have been configured, click the **Domain** cell to select the Domain(s) in which the rights granted by the Administrator Role and the selected elements apply.

    - You can leave the default **Shared Domain** selected in the Domains cell. All the elements automatically belong to the predefined Shared Domain if Domain elements have not been configured. For more information on Domains, refer to the Management Client *Online Help* or the *Stonesoft Administrator’s Guide*.
    - You can also select the **ALL Domains** Access Control List to grant permissions for all the Domains that exist in the system.

5. (*Optional*) Repeat Steps 2 to 5 to define additional administrator permissions for API Clients.

6. Click **OK**. The API Client account is ready for use.

In addition to the privileges you explicitly set, API Clients who are allowed to create and delete elements automatically have privileges to view, edit, and delete elements they create themselves, even if they are not allowed to view, edit, or delete any other elements.

Allowing SMC API Connections

You must modify the IPv4 Access rules in your policy to allow the connections from the Web Applications that use the SMC API to the Management Server.

To Configure the Access Rule for the SMC API

Create the following IPv4 Access rule to allow the SMC API connections:

Source	Destination	Port	Service	Action
A list of the IP addresses of the clients that you want to restrict access to the SMC API to.	Management Server	The Port defined for the SMC API Service in Management Server Properties.	TCP	Allow

Note - Remember to adjust the NAT rules as well if it is necessary in your network setup.

RESTful Principles

The SMC API is a RESTful API (see http://en.wikipedia.org/wiki/Representational_state_transfer).

This means that:

- The API is strictly based on the HTTP protocol and is platform-independent.
- Each resource is identified by a unique URI, which should be opaque to clients.
- URIs and actions that can be performed on resources are accessible through hyperlinks.
- The 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

- You can 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.
- You can read resources by using GET requests. In order to save network bandwidth and avoid transferring the complete body in the response, HEAD requests and ETags are supported.
- You can update resources by using PUT requests. All updates are conditionals, relying on ETags.
- You can delete resources by using DELETE requests.
- You can trigger actions such as Policy Uploads by using POST requests.
- For automatic discovery of supported requests for each URI, OPTIONS requests are supported.

Note - 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 redundant requests should be avoided. POST requests may have additional effects if they are called more than once with the same input parameters, and clients are responsible for avoiding multiple requests.

Status Codes and Error Messages

Requests return HTTP response status codes that follow the principles outlined in http://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

In the case of an error, the server also attempts to send relevant information in the response body.

ETags

GET requests return an ETag in the header that attests the version of the returned data. For more information, see http://en.wikipedia.org/wiki/HTTP_ETag.

ETags are used for:

- Caching purposes: When performing GET requests, the client should send back the ETags of the last fetched data in an `If-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.

Representations

See http://en.wikipedia.org/wiki/Content_negotiation.

Opaque URIs, URI Discovery, and Hypermedia

All URIs must be considered opaque values: clients should never 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.
- From top-level service URIs.
- From the top-level lists of elements.
- From links to other resources that are mentioned in these elements.
- From the action links identified by the verbs (e.g. `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

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

Entry-Point

Verbs

Verbs represent keywords for specific element operations. Verbs are listed in the XML/JSON element description as a ***link*** entry.

Self Verb

The self verb is included in each element. The self verb is REST-philosophy-oriented. The self verb allows you to retrieve the current element's API URL.

For example, for a host, you could have the following:

```
"link":
   [
       {
           "href": "http://localhost:8082/5.6/elements/host/86",
           "method": "GET",
           "rel": "self",
           "type": "host"
        }
   ]
```

A ***link*** entry has the following structure:

- href: The API's URL to the associated verb.
- method: The REST HTTP method to execute on the API's URL.
- rel: The keyword that is preserved beyond SMC versions. It represents the verb.
- type: Optional information about the return type.

Uploading a Verb From Policy Elements

In this example, the JSON description of a policy refers to the verb in the following way:

```
"link":
   [
       {
           "href": "http://localhost:8082/5.6/elements/fw_policy/56/upload",
           "method": "POST",
           "rel": "upload"
       },
       …
   ]
```

This verb can be found in each policy type. For 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:8082/5.6/elements/fw_policy/56/upload/NWYyMDBiOTA4ZTY3NDM0ZTo0YzM2ZTg5MDoxM2
ZlNzhhMDZlZTotN2VhZA==",
       "href": "http://localhost:8082/5.6/elements/fw_policy/56",
       "in_progress": true,
       "last_message": "",
       "success": true
    }
```

The upload status has the following structure:

- follower: The API's URL to the current upload status.
- href: The source of the upload (here, 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 a Verb From Engine Elements

In this example, the JSON description of an engine refers to the verb in the following way:

```
"link":
   [
       …
       {
           "href": "http://localhost:8082/5.6/elements/single_fw/1552/upload",
           "method": "POST",
           "rel": "upload"
       },
       …
   ],
```

This verb can be found in each engine type. For example, here 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 the same kind of upload status description as for the policy upload.

Refreshing a Verb From Engine Elements

In this example, the JSON description of an engine refers to the verb in the following way:

```
"link":
   [
       …
       {
           "href": "http://localhost:8082/5.6/elements/single_fw/1552/refresh",
           "method": "POST",
           "rel": "refresh"
       },
       …
   ],
```

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

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

API Discovery

In order to ease migration during major version upgrades, it is preferable to discover the API starting from an entry-point rather than define all the URIs. Verbs do not change in major version upgrades but URIs may change.

The execution of a GET request on the following URL returns a list of available functionalities.

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

```
GET https://localhost:8082/5.6/api
```

Links

GET 5.6/api allows the discovery of all the 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 URL to the associated entry-point.
- method: The REST HTTP method to execute on the API's URL.
- rel: The keyword that is preserved beyond SMC versions.

For example, for the entry-point ***host,*** the API's URL is ***GET 5.6/elements/host***. This means that the execution of ***GET 5.6/elements/host*** returns all the defined hosts in the system in the HTTP response body.

To log in, execute ***POST 5.6/login***.

Note - *GET 5.6/api* does not give query parameter information. Query parameters are defined in the API documentation. For example, the login method needs to have the *authenticationkey* query parameter to succeed.

The execution of ***GET 5.6/api*** with ***‘Accept: application/xml’*** returns the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<api>

…

<entry_point href="http://localhost:8082/5.6/elements/host" method="GET" rel="host"/>

<entry_point href="http://localhost:8082/5.6/login" method="POST" rel="login"/>

…

</api>
```

The execution of ***GET 5.6/api*** with ***‘Accept: application/json’*** returns the following:

```
{

"entry_point":

[

…

{

"href": "http://localhost:8082/5.6/elements/host",

"method": "GET",

"rel": "host"

},

{

"href": "http://localhost:8082/5.6/login",

"method": "POST",

"rel": "login"

},

…

]

}
```

Data Elements

Elements can be retrieved 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/elements as a content description.

By default, all the attributes/elements from the data element input (XML or JSON) that are not supported are ignored. For this reason, it is strongly recommended to first retrieve an existing element in XML or JSON and then create a new one 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 (http://en.wikipedia.org/wiki/Json). JSON is based on key/value arrays.

For 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:8082/5.6/elements/host/86",
               "method": "GET",
               "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.

XML

The API also supports the XML format. As a standard (http://en.wikipedia.org/wiki/XML), 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.

For 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:8082/5.6/elements/host/86" method="GET" 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).

Collections

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 5.6/elements*, so it is possible to search all elements using the *192.168.** IP address pattern with the following query:

```
GET 5.6/elements?filter=192.168.*
```

It is also 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 5.6/elements/host?filter=host
```

For information on searching for unused elements or duplicated IP address elements, see Specific Searches.

Retrieving a Resource

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

For example, after having retrieved the API’s URL for hosts, GET 5.6/elements/host returns the following:

```
{
   "result":
   [
       {
           "href": "http://localhost:8082/5.6/elements/host/86",
           "name": "Your-Freedom Servers",
           "type": "host"
                },
        {
           "href": "http://localhost:8082/5.6/elements/host/39",
           "name": "DHCP Broadcast Destination",
           "type": "host"
       },
       …
    ]
}
```

The HTTP request lists all the defined hosts with their API’s URLs. If a specific host is needed, it is preferable to search for the host by its name to get the same kind of result but only including the particular host.

For example, GET 5.6/elements/host/39 returns a 200 HTTP response status code and the specified XML/JSON description.

Note - 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.

For example, for a host, the POST 5.6/elements/host with the following returns a 201 HTTP response status code and the created element API's URL in the HTTP header:

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

Note - 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 this is the last current version of the element.

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 one 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 URL.

Note - 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 URL, you can execute a DELETE operation on it. If the execution succeeds, it returns a 204 HTTP response status code.

Other Actions

Specific Searches

The API makes it possible to execute specific searches. For example, you can search for unused elements or duplicate IP addresses.

Searching for Unused Elements

```
GET 5.6/elements/search_unused
```

This operation executes a search for all unused elements. It is possible to also filter this search by a specific name, comment, or IP address with the query parameter ***filter***.

Searching for Duplicate IP Addresses

```
GET 5.6/elements/search_duplicate
```

This operation executes a search for all duplicated IP address elements. It is also possible to filter this search by a specific name, comment, or IP address with the query parameter ***filter***.

System Information

```
GET 5.6/system
```

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

Examples

In the following examples, the SMC API is simply configured on port 8082, without any host name restrictions, and reached from the same machine that the Management Server is running on. Hence, the SMC API entry-point URI is `http://localhost:8082/api`. An SMC API Client element with the appropriate permissions and an authentication key equal to `sqfTm8UCd6havtycRP7P0001` has been defined in the Management Client.

Unless otherwise specified, all examples use JSON representations. The example elements (for example, Helsinki FW and HQ Policy) take their names and properties from the elements that are used in the standard demo environment of the SMC.

There are several python example scripts in the `samples` directory. Explanations of these samples are provided below.

Version-Specific Entry-Point

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 the supported API versions and their entry-point URIs. At the time of writing, only the 5.6 version of the SMC API is supported, so the following is returned:

```
GET http://localhost:8082/api

Status Code: 200 OK

{

  "version": [

    {

      "href": "http://localhost:8082/5.6/api",

      "method": "GET",

      "rel": "5.6"

    }

  ]

}
```

Global Services and Element URIs

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

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:8082/5.6/api
Status Code: 200 OK
{
  "entry_point": [
    {
      "href": "http://localhost:8082/5.6/elements",
      "method": "GET",
      "rel": "elements"
    },
…
    {
      "href": "http://localhost:8082/5.6/elements/host",
      "method": "GET",
      "rel": "host"
    },
…
    {
      "href": "http://localhost:8082/5.6/login",
      "method": "POST",
      "rel": "login"
    }
  ]
}
```

Logging in

Login is performed with a POST request on the `login` service URI. The API Client authentication key must be specified in the query parameter named `authenticationkey`.

```
POST http://localhost:8082/5.6/login?authenticationkey=sqfTm8UCd6havtycRP7P0001

Status Code: 201 Created
```

Listing the Hosts Collection

After login, you can get a list of all the defined hosts with the following request:

```
GET http://localhost:8082/5.6/elements/host
```

This request returns a 200 HTTP response status code and the following:

```
{
        "result":
        [
         {
          "href": "http://localhost:8082/5.6/elements/host/40",
          "name": "DHCP Broadcast Originator",
          "type": "host"
         },
         {
          "href": "http://localhost:8082/5.6/elements/host/43",
          "name": "IPv6 Unspecified Address",
          "type": "host"
         },
         …
        ]
}
```

Filtering Elements

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

```
GET http://localhost:8082/5.6/elements/host?filter=Your-Freedom Servers
```

This request returns a 200 HTTP response status code and the following:

```
{
    "result":
    [
        {
            "href": "http://localhost:8082/5.6/elements/host/86",
            "name": "Your-Freedom Servers",
            "type": "host"
        }
    ]
}
```

Creating a New Host

After login, creating a new Host in the JSON format with the following request:

```
POST http://localhost:8082/5.6/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"]
}
```

Returns a 201 HTTP response status code and the following in the HTTP header:

```
Location: http://localhost:8082/5.6/elements/host/1704
```

See `createHostThenDeleteIt.py` JSON or XML samples.

Attempting to Create an Element with an Existing Name

If you try to create an element with an already existing name, you will 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 using the filtering feature:

```
GET http://localhost:8082/5.6/elements/host?filter=mySrc1
```

After the element is found, using the following request:

```
GET http://localhost:8082/5.6/elements/host/1704
```

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

```
Etag: MTcwNDMxMTEzNzQwNDMwNzM1NDQ=
        {
            "address": "192.168.0.13",
            "comment": "My SMC API's my Src Host 2",
            "key": 1704,
            "link":
            [
                {
                    "href": "http://localhost:8082/5.6/elements/host/1704",
                    "method": "GET",
                    "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).

```
PUT http://localhost:8082/5.6/elements/host/1704
```

Using *Etag: MTcwNDMxMTEzNzQwNDMwNzM1NDQ*= 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:8082/5.6/elements/host/1704
```

See `updateHostThenDeleteIt.py` JSON or XML samples.

Deleting a Host

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

```
GET http://localhost:8082/5.6/elements/host?filter=mySrc1
```

```
http://localhost:8082/5.6/elements/host/1704
```

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

```
DELETE http://localhost:8082/5.6/elements/host/1704
```

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

Modifying a Rule in a Policy

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

```
GET http://localhost:8082/5.6/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:8082/5.6/elements/fw_policy/56
```

```
GET http://localhost:8082/5.6/elements/fw_policy/56
```

- *fw_ipv4_access_rules:* A special link to the Firewall Policy that retrieves all Firewall IPv4 Access Rules in the current Firewall Policy.
- *fw_ipv6_access_rules:* A special link to the Firewall Policy that retrieves all Firewall IPv6 Access Rules in the current Firewall Policy.

- *fw_ipv4_nat_rules:* A special link to the Firewall Policy that retrieves all Firewall IPv4 NAT Rules in the current Firewall Policy.
- *fw_ipv6_nat_rules:* A special link to the Firewall Policy that retrieves all Firewall IPv6 NAT Rules in the current Firewall Policy.

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

```
{
        "href": "http://localhost:8082/5.6/elements/fw_policy/56/fw_ipv4_access_rule/514",
        "name": "Rule @514.0",
        "type": "fw_ipv4_access_rule"
}
```

```
GET http://localhost:8082/5.6/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:8082/5.6/elements/fw_policy/56/fw_ipv4_access_rule/514",
                    "method": "GET",
                    "rel": "self",
                    "type": "fw_ipv4_access_rule"
                }
            ],
            "parent_policy": "http://localhost:8082/5.6/elements/fw_policy/56",
            "rank": 4,
            "read_only": false,
            "system": false,
            "tag": "514.0"
}
```

The result has *Etag: MjU0Mzk4MTEzMDYyMzMyMzYxMTg*= 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:8082/5.6/elements/fw_policy/56/fw_ipv4_access_rule/514
```

The new JSON content with the updated comment and *Etag: MjU0Mzk4MTEzMDYyMzMyMzYxMTg=* as the HTTP request header returns a 200 HTTP response status code and the following in the HTTP response header:

```
http://localhost:8082/5.6/elements/fw_policy/56/fw_ipv4_access_rule/514
```

See `addRuleAndUpload.py` JSON or XML samples.

Performing and Following Up on a Policy Upload

There are two ways of uploading a policy: from the policy and from the engine.

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

```
GET http://localhost:8082/5.6/elements?filter=Helsinki FW
```

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

```
        "link":
        [
        {
        "href": "http://localhost:8082/5.6/elements/fw_cluster/1563/refresh",
        "method": "POST",
        "rel": "refresh"
        },
        {
        "href": "http://localhost:8082/5.6/elements/fw_cluster/1563/upload",
        "method": "POST",
        "rel": "upload"
        },
        …
]
```

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

```
POST http://localhost:8082/5.6/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:8082/5.6/elements/fw_cluster/1563/upload/NWYyMDBiOTA4ZTY3NDM0ZTotNzg
        yM2JmMmI6MTNmZWMxMGI3ZGY6LTdmZDA=",
            "href": "http://localhost:8082/5.6/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:8082/5.6/elements/fw_cluster/1563/upload/NWYyMDBiOTA4ZTY3NDM0ZTotNzgyM2JmMmI6
MTNmZWMxMGI3ZGY6LTdmZDA=
```

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 proceed to a policy refresh. As you can see from the engine links, the verb ‘refresh’ is also available on the engine:

```
POST http://localhost:8082/5.6/elements/fw_cluster/1563/refresh
```

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

See `addRuleAndUpload.py` JSON or XML samples.

Search Filtering by Group Type

It is possible to filter searches by group type. The following 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 and IPS rules.

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

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

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 on host, router, and network.

Retrieving Routing/Antispoofing Information

To retrieve the complete (static/dynamic) routing/antispoofing 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:8082/5.6/elements/fw_cluster/1563/routing/887",
        "method": "GET",
        "rel": "routing",
        "type": "routing"
        },
        {
        "href": "http://localhost:8082/5.6/elements/fw_cluster/1563/antispoofing/990",
        "method": "GET",
        "rel": "antispoofing",
        "type": "antispoofing"
        },
        …
],
```

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

```
GET http://localhost:8082/5.6/elements/fw_cluster/1563/routing/887
```

This returns a 200 HTTP response status code and the following:

```
{
"href": "http://localhost:8082/5.6/elements/fw_cluster/1563",
"ip": "10.8.0.21",
"key": 887,
"level": "engine_cluster",
"link":
[
{
"href": "http://localhost:8082/5.6/elements/fw_cluster/1563/routing/887",
"method": "GET",
"rel": "self",
"type": "routing"
}
],
"name": "Helsinki FW",
"read_only": false,
"routing_node":
[
{
"exclude_from_ip_counting": false,
"href": "http://localhost:8082/5.6/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:8082/5.6/elements/fw_cluster/1563/antispoofing/990
```

This returns a 200 HTTP response status code and the following:

```
…
"auto_generated": "true",
"href": "http://localhost:8082/5.6/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:8082/5.6/elements/fw_cluster/1563",
"ip": "10.8.0.21",
"key": 990,
"level": "engine_cluster",
"link":
[
{
"href": "http://localhost:8082/5.6/elements/fw_cluster/1563/antispoofing/990",
"method": "GET",
"rel": "self",
"type": "antispoofing"
}
],
"name": "Helsinki FW",
"read_only": false,
"system": false,
"validity": "enable"
}
```

Stonesoft Corporation

Itälahdenkatu 22A
FI-00210 Helsinki
Finland

Tel. +358 9 476 711
Fax +358 9 4767 1349

STONESOFT
A McAfee Group Company

Stonesoft Inc.

1050 Crown Pointe Parkway
Suite 900
Atlanta, GA 30338
USA

Tel. +1 770 668 1125
Fax +1 770 668 1131