

Provisioning Tools

Ericsson Service-Aware Policy Controller

User Guide

Copyright

© Ericsson AB 2018. All rights reserved. No part of this document may be reproduced in any form without the written permission of the copyright owner.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

Trademark List

All trademarks mentioned herein are the property of their respective owners. These are shown in the document [Trademark Information](#).



Contents

1	Provisioning Tools Introduction	1
2	Provisioning Tools	2
3	Provisioning with Resty	3
3.1	Resty Usage Reference	3
3.2	Initialize Base URI	3
3.3	PUT	5
3.4	GET	5
3.5	DELETE	5
3.6	Operation Results	6
3.7	Prettify JSON Data	7
3.8	Edit Input JSON Data on an Editor	7
3.9	Reuse Base URI for Massive Provisioning	8
4	Provisioning with Postman	10
4.1	Installation	10
4.2	Authentication	10
4.3	PUT	10
4.4	GET	11
4.5	DELETE	12
4.6	Operation Results	13
4.7	TLS Certificate	13
5	Massive Data Retrieval	14
6	Operation and Maintenance	15
6.1	Security	15
6.2	Troubleshooting	15





1 Provisioning Tools Introduction

This instruction provides a guideline for the provisioning tools supplied with the SAPC.

To provision the SAPC, the following conditions must be fulfilled:

- The SAPC product software has been installed.
- The user must have basic knowledge of HTTP, TLS certificates, REST APIs, and the JSON format.



2 Provisioning Tools

The SAPC provides a REST API for provisioning data, refer to [Provisioning REST API](#). To ease provisioning operations, the SAPC provides an HTTPS CLI client named `resty`, although any other HTTPS client can be used.

`resty` is a tiny script wrapping `curl`. It provides a simple, concise shell interface for interacting with REST services. Since it is implemented as functions in the shell and not in its own separate command environment, all the powerful shell tools, such as `perl`, `awk`, `grep`, and `sed`, among others, are accessible. More information about `resty`, its use, and some examples are provided in subsequent sections. For general information on `resty` visit <https://github.com/micha/resty>.

`curl` is an open source command-line tool for transferring data with URL syntax, and the base for `resty`. Although it can also be used for provisioning the SAPC, Ericsson recommends `resty` because it provides a more user-friendly interface towards the operators. For more information on `curl` visit <https://curl.haxx.se/>.

Other tools that can also be used for provisioning are included in later sections.



3 Provisioning with Resty

3.1 Resty Usage Reference

resty is available and must be executed from a System Controller (SC) in the node. The [System Administrator Guide](#) contains information about how to connect to the SC.

Note: In geographical redundant deployments, do the provisioning only on the active node.

This is the basic usage reference for resty. More details about specific operations are provided in next sections.

```
resty [-v]                # prints the base URI
resty <remote> [OPTIONS]  # sets the base URI

GET [path] [OPTIONS]      # GET request
DELETE [path] [OPTIONS]   # DELETE request
PUT [path] [data] [OPTIONS] # PUT request

Options:

-Q          Don't URL encode the path.
-W          Don't write to history file
            (only when sourcing script).
-V          Edit the input data interactively in 'vi'.
            (PUT, PATCH, and POST requests only, with
            data piped to stdin).
-Z          Raw output. This disables any processing
            of HTML in the response.
-v          Verbose output. When used with the resty
            command itself this prints the saved curl
            options along with the current URI base.
            Otherwise this is passed to curl for
            verbose curl output.
--dry-run   Just output the curl command.
<curl opt> Any curl options will be passed
            down to curl.
```

3.2 Initialize Base URI

Initialize resty following these steps:



Steps

1. Set the base URI. The base URI is the URI used as a foundation for subsequent requests. Specifically, it is a URI that contains the * character one or more times. The * are replaced with the path parameter in the GET, PUT, or DELETE request. Otherwise, if no * character is included, it assumes that the provided URI is the prefix for the rest of provisioning URIs. When successful, `resty` prints the base URI that is used from that moment on for the provisioning operations. The `resty` command for initializing the base URI is:

```
sapcprov@SC-1:~> resty https://<$VIPP>:<$PORTP>/provisioning/v1  
-H "Content-Type: application/json" -k -u sapcprov:<password>  
  
https://<>10.95.95.9:8443/provisioning/v1*
```

Note: `sapcprov` user is used for provisioning operations. Execute the command `su sapcprov` in order to change from `sapcadmin` user to `sapcprov` before running `resty` commands.

As a facility, the system sets the following environment variables when logged in on an SC processor. They define the IP and TCP port, respectively, where the SAPC listens to incoming REST API requests:

- `VIPP`: Provisioning VIP.
- `PORTP`: Provisioning TCP port.

`resty` uses these environment variables, as in the previous example, at initialization.

Besides the base URI, the example shows some `curl` CLI parameters to specify a custom header (`-H`), an indication to accept insecure connections (`-k`) and user authentication details (`-u`).

2. Make HTTPS requests. The HTTP verbs (GET, PUT, and DELETE) are used from the CLI, and their first argument is always an optional URI path. This path must always start with a / character. If the path parameter is not provided on the command line, `resty` uses the last path it was provided with. This "last path" is stored in an environment variable (`$_resty_path`), so each terminal basically has its own "last path". `resty` always URL encodes the path, except for slashes. Slashes in path elements need to be manually encoded as `%2F`. This means that the `?`, `=`, `&`, and some other problematic characters are encoded. To disable this behavior, use the `-Q` option.

Note: For a complete reference of all available resources, refer to the Provisioning REST API.



3.3 PUT

The PUT method executes HTTP PUT requests. There are several ways to provide the request body data:

- Directly on the command line, as a parameter in JSON format. For example:

```
sapcprov@SC-1:~> PUT /contents/100_http
'{"contentName": "100_http", "pccRuleName": "100",
"pccRuleType": 0}'

{}
```

- From a file, piping its content. For example, if the file `contents_100.json` contains the data to provision in JSON format:

```
sapcprov@SC-1:~> PUT /contents/100_http < contents_100.json

{}
```

- Piping the output from another program. For example, if the file `contents_100.json` contains the data to provision in JSON format, but we want to replace all occurrences of "http" to "https":

```
sapcprov@SC-1:~> sed 's/http/https/' contents_100.json | PUT /
contents/100_https

{}
```

3.4 GET

The GET method executes HTTP GET requests. For example:

```
sapcprov@SC-1:~> GET /contents/100_http

{"contentName": "100_http", "pccRuleName": "100", "pccRuleType": 0}
```

This returns the previous PCC rule as a raw JSON. To transform the output to a more human readable format, see [Prettify JSON Data](#) on page 7.

3.5 DELETE

The DELETE method executes HTTP DELETE requests. For example:

```
sapcprov@SC-1:~> DELETE /contents/100_http
```

This deletes the previous PCC rule. When the execution is successful, `resty` prints no output. To verify manually the status of the node, check the return code of the operation as indicated in [Operation Results](#) on page 6.



3.6 Operation Results

Successful requests (with 2xx status code responses) return zero. For example:

```
sapcprov@SC-1:~> GET /contents/100_http
{"contentName":"100_http","pccRuleName":"100","pccRuleType":0}

sapcprov@SC-1:~> echo $?

0
```

Otherwise, the first digit of the response status is returned (that is, 4 for 4xx, 5 for 5xx, and so on). This is because the exit status is an 8-bit integer (it cannot be greater than 255). For example:

```
sapcprov@SC-1:~> GET /contents/missing
{"error":{"code":"404","description":"Not found."}}

sapcprov@SC-1:~> echo $?

4
```

To obtain the exact status code, use the `-v` option from `curl`. For example:

```
sapcprov@SC-1:~> GET /contents/missing -v

* <url> malformed
* Closing connection -1
* Hostname was NOT found in DNS cache
*   Trying 10.95.95.9...
* Connected to 10.95.95.9 (10.95.95.9) port 8443 (#0)
* Server auth using Basic with user 'sapcprov'
> GET /provisioning/v1/contents/missing HTTP/1.1
> Authorization: Basic c2FwY3Byb3Y6cGFzc3dvcmQ=
> User-Agent: curl/7.49.1
> Host: 10.95.95.9:8443
> Accept: */*
> Content-Type: application/json
>
< HTTP/1.1 404 Not Found
< Connection: Keep-Alive
< Content-Length: 51
< Content-Type: application/json
< Date: Thu, 18 Aug 2016 14:42:59 GMT
<
{ [51 bytes data]
* Connection #0 to host 10.95.95.9 left intact
{"error":{"code":"404","description":"Not found."}}
```

As a reference, these are the possible status codes returned by `resty`:



Table 1 Resty Status Codes

Status code	Description	Example
0	Success	N/A
4	Client Error	400 - Bad Request. The input JSON data is not valid
5	Server Error	500 - Internal Server Error. The server encountered an unexpected condition which prevented it from fulfilling the request

3.7 Prettify JSON Data

resty output is a raw JSON. For example:

```
sapcprov@SC-1:~> GET /contents/100_http
```

```
{"contentName": "100_http", "pccRuleName": "100", "pccRuleType": 0}
```

Although this is a valid JSON, it is not very readable, so the SAPC provides an alias called `pp` that eases reading the JSON output:

```
sapcprov@SC-1:~> alias pp
```

```
alias pp='python -m json.tool'
```

`pp`, short for "pretty-print", indents and prettifies JSON data. For example:

```
sapcprov@SC-1:~> GET /contents/100_http | pp
```

```
{
  "contentName": "100_http",
  "pccRuleName": "100",
  "pccRuleType": 0
}
```

3.8 Edit Input JSON Data on an Editor

By using the `-V` option, data is edited in `vim`, so the resulting data of a GET can be piped and used in a PUT request. For example:

```
sapcprov@SC-1:~> GET /contents/100_http | PUT -V
```

This fetches the data from the output of a GET request and allows editing it, then performs a PUT with the edited JSON.



3.9 Reuse Base URI for Massive Provisioning

The base URI defined in section [Initialize Base URI](#) on page 3 is reused as long as the user remains in the shell and the base URI is not changed. This can be used for massive provisioning, where multiple requests are executed against the same base URI.

To provision multiple resources to the same subscriber, set the base URI temporarily to also include the subscriber, for example:

```
sapcprov@SC-1:~> resty https://<$VIP>:<$PORT>/provisioning/v1/  
subscribers/<subscriberId1> -H "Content-Type: application/json" -k  
-u sapcprov:<password>
```

```
https://<>10.95.95.9:8443/provisioning/v1/subscriber/  
<subscriberId1>*
```

```
sapcprov@SC-1:~> PUT / <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /dataplan/<dataplanName> <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /static-qualification <data>
```

```
{}
```

```
sapcprov@SC-1:~> ... # Rest of provisioning for subscriber 1
```

```
sapcprov@SC-1:~> resty https://<$VIP>:<$PORT>/provisioning/v1/  
subscribers/<subscriberId2> -H "Content-Type: application/json" -k  
-u sapcprov:<password>
```

```
https://<>10.95.95.9:8443/provisioning/v1/subscribers/  
<subscriberId2>*
```

```
sapcprov@SC-1:~> PUT / <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /dataplan/<dataplanName> <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /static-qualification <data>
```

```
{}
```

```
sapcprov@SC-1:~> ... # Rest of provisioning for subscriber 2
```



Compare the URIs in the previous example with the URIs in case only the base URI is used:

```
sapcprov@SC-1:~> resty https://<$VIP>:<$PORT>/provisioning/v1 -H
"Content-Type: application/json" -k -u sapcprov:<password>
```

```
https://<>10.95.95.9:8443/provisioning/v1*
```

```
sapcprov@SC-1:~> PUT /subscribers/<subscriberId1> <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /subscribers/<subscriberId1>/dataplan/
<dataplanName> <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /subscribers/<subscriberId1>/static-
qualification <data>
```

```
{}
```

```
sapcprov@SC-1:~> ... # Rest of provisioning for subscriber 1
```

```
sapcprov@SC-1:~> PUT /subscribers/<subscriberId2> <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /subscribers/<subscriberId2>/dataplan/
<dataplanName> <data>
```

```
{}
```

```
sapcprov@SC-1:~> PUT /subscribers/<subscriberId2>/static-
qualification <data>
```

```
{}
```

```
sapcprov@SC-1:~> ... # Rest of provisioning for subscriber 2
```

Note: Every method invocation produces a separate TCP connection. Previous TCP connections are not reused.



4 Provisioning with Postman

Postman is a Chrome plug-in that helps working efficiently with APIs. It can be used from any host that can run the Chrome browser and for which there is network connectivity towards the SAPC's Provisioning VIP. For more information, visit <https://www.getpostman.com/>.

Note: The description about how to use Postman and the screen captures have been obtained with Postman v4.6.2 and Chrome v52.0.2743.116 on Windows 7. Other versions or platforms may vary.

4.1 Installation

Postman is a Chrome App. This means that Postman runs on the Chrome browser. To use Postman, Google Chrome must be installed first. For more information, visit <http://www.google.com/chrome/>.

Once Chrome is installed, go to the Chrome Webstore at <https://chrome.google.com/webstore/category/apps> to find Postman and click **Add to Chrome** to install it on Chrome.

The download may take a few minutes, depending on the internet connection. Once Postman is downloaded, start it via `chrome://apps` in the browser or via the Chrome App Launcher in the dock or taskbar.

4.2 Authentication

Postman needs the proper credentials to authenticate the user against the REST API server. To provide the credentials, introduce the user and password using **Basic Auth** in the **Authorization** tab when executing any request:

The screenshot shows the Postman interface with the 'Authorization' tab selected. The 'Type' dropdown is set to 'Basic Auth'. The 'Username' field contains 'sapcprov' and the 'Password' field contains a masked password '*****'. There are 'Clear' and 'Update Request' buttons. A 'Generate Code' button is in the top right. A 'Show Password' checkbox is at the bottom left. A note on the right states: 'The authorization header will be generated and added as a custom header'. A 'Save helper data to request' checkbox is also present.

Figure 1 User Authentication for Requests

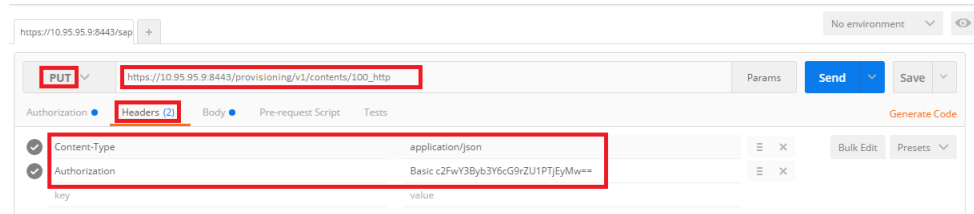
4.3 PUT

The **PUT** option executes HTTP PUT requests. To execute a PUT request:



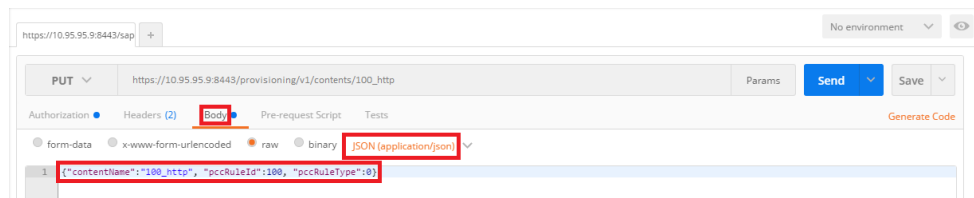
Steps

1. Provide a "Content-Type: application/json" header, for example:

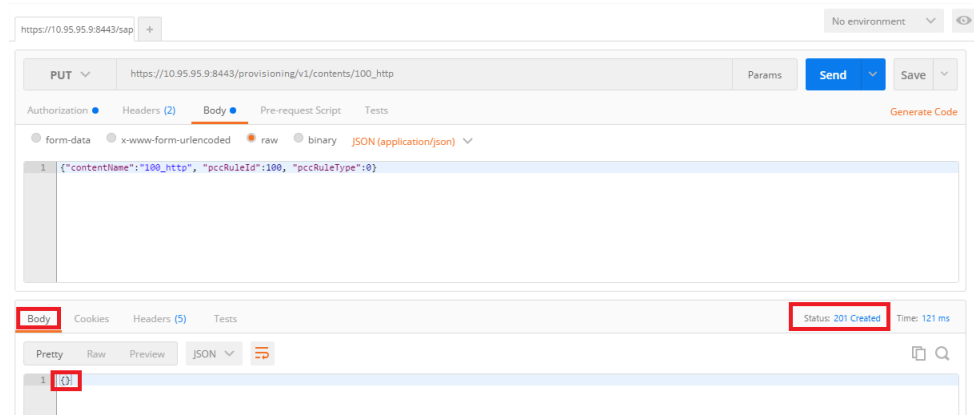


Note: The Authorization header is automatically added when the user provisions the authorization data as defined in [Authentication](#) on page 10.

2. Provide a request body with the data to provision, for example:



3. Click **Send** to execute the request.
4. If the execution succeeds, a valid HTTP response is obtained, for example:



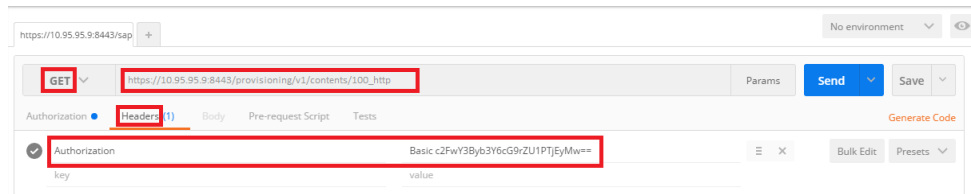
4.4

GET

The **GET** option executes HTTP GET requests. To execute a GET request:

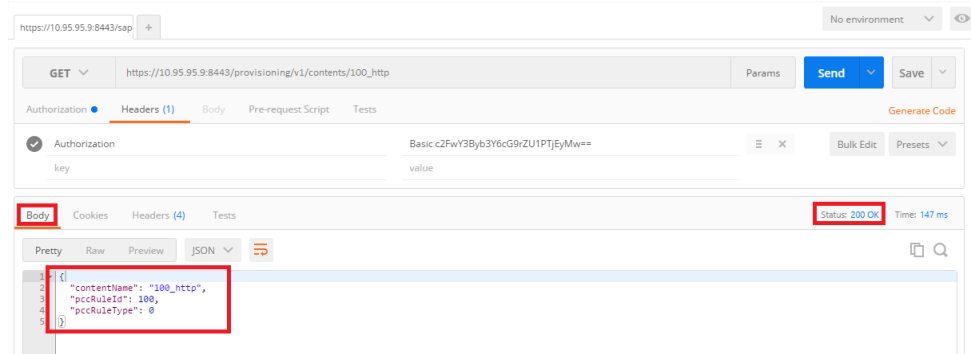
Steps

1. Do not add any extra header, the REST API server assumes that the output format is JSON, for example:



Note: The Authorization header is automatically added when the user provisions the authorization data as defined in [Authentication](#) on page 10.

2. Click **Send** to execute the request.
3. If the execution succeeds, a valid HTTP response is obtained, for example:



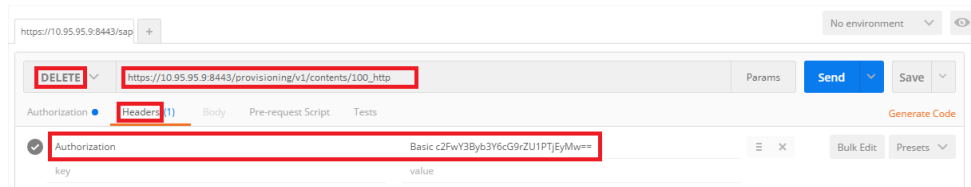
4.5

DELETE

The **DELETE** option executes HTTP DELETE requests. To execute a DELETE request:

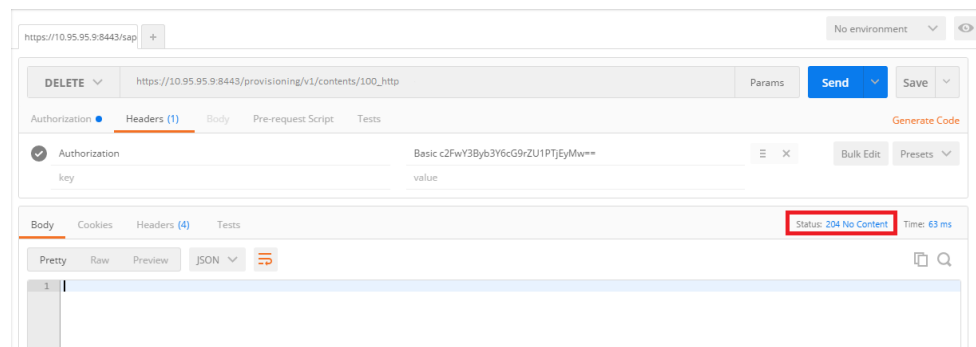
Steps

1. Do not add any extra header, for example:



Note: The Authorization header is automatically added when the user provisions the authorization data as defined in [Authentication](#) on page 10

2. Click **Send** to execute the request.
3. If the execution succeeds, a valid HTTP response is obtained, for example:



4.6 Operation Results

The status code of the HTTP response is the same as defined in [Operation Results](#) on page 6. It can be found in the main screen:

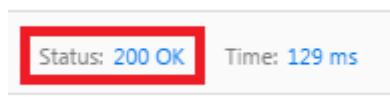


Figure 2 Status Code of the Response

4.7 TLS Certificate

Postman needs that the TLS certificate used by the REST API server is installed in Chrome. Otherwise, Postman shows the following error when executing a request:

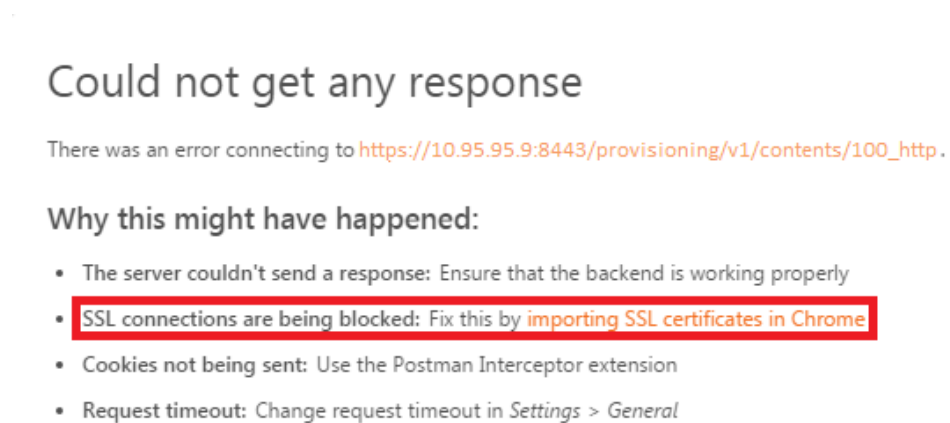


Figure 3 Error Message when the TLS Certificate is not Installed in Chrome

If you get the previous error message and have not installed the certificate in Chrome, follow the instructions in the link to install it manually.



5 Massive Data Retrieval

The SAPC supports to retrieve some data collections in a single GET response (for details of supported collections see [Provisioning REST API](#)). The request timeout in the SAPC rest server is 180 seconds. To be able to get the request, configure the request timeout in the rest HTTP client accordingly.

When the SAPC is not able to return the collection data inside the HTTP GET response (HTTP status code 200 OK), the SAPC dumps it to a file. The file is available on the path returned in the GET response containing HTTP status 202 (Accepted).

To retrieve the file containing the results, follow the procedure explained in [Fetch File in Logical File System](#), connecting to the SFTP server using the sapcadmin user.

Highly time consuming data collections are:

`subscribers`



6 Operation and Maintenance

6.1 Security

Provisioning is not supported through 'plain text' HTTP. HTTPS is required instead. Therefore, an HTTPS client shall be used for the provisioning.

TLS certificates are used to establish secure encrypted connections. The TLS connection protects sensitive data, so that even if the connection is eavesdropped it is impractical to try brute force attacks to break it owing to the computing power and time required. The HTTPS client used for provisioning the SAPC has to have the valid TLS certificates installed or accept self-signed TLS certificates. For more information about TLS certificates, refer to [Security Management Guide](#).

Note: When provisioning the SAPC from an SC, as the user is already in the internal network, the provisioning operations can be considered secure, so the HTTPS client can accept self-signed TLS certificates safely.

Note: When provisioning the SAPC from an external node, HTTPS clients without the valid TLS certificates may be used if configured to accept self-signed TLS certificates, although this option is not recommended. Use that option under your own responsibility, as it poses a severe security risk. Provisioning from an SC or with an HTTPS client with valid TLS certificates installed is recommended, instead.

For more information about the recommended tools and procedures for provisioning, see [Provisioning with Resty](#) on page 3.

6.2 Troubleshooting

Some typical errors and possible solutions are described in next table.

Table 2 Provisioning Tools Problems and Possible Causes and Solutions

Problem	Possible Cause	Possible Solution
Requests provide an empty return or hang in resty	<ul style="list-style-type: none"> —The \$VIPP variable, the \$PORTP variable, or both, defined automatically when resty is installed, are empty or not set —The TLS certificate is not accepted by the HTTPS client 	<ul style="list-style-type: none"> —Make sure that you are logged in on an SC with the correct user. For more details, refer to the System Administrator Guide, section <code>ssh</code>. —Make sure that you are using the default shell provided by the SAPC in the SC.



Problem	Possible Cause	Possible Solution
		<ul style="list-style-type: none">—Make sure that the environment variables have not been overwritten.—Make sure that <code>resty</code> is using a valid TLS certificate or that it is accepting insecure connections with the <code>-k</code> parameter.
Error 400 - Bad Request	Invalid input JSON	Double check that the input JSON provided is valid.
Error 401 - Unauthorized	Invalid credentials	<p>Make sure that the HTTPS client is sending the correct user and password.</p> <ul style="list-style-type: none">—If using <code>resty</code>, this is done with the <code>-u <user>:<password></code> parameter.—If using another HTTPS client, double check the authentication information for that specific client.
Error 404 - Not Found	REST resource not found	Double check that the provisioning URL and the path are correct.
Error 415 - Unsupported Media Type	Content Type is not correct	<p>Make sure that the "Content-Type: application/json" header is included in the request.</p> <ul style="list-style-type: none">—If using <code>resty</code>, this is done with the <code>-H "Content-Type: application/json"</code> parameter—If using another HTTPS client, double check the information for that specific client about how to include the "Content-Type: application/json" header to the request.