Meraki Network Automation

Meraki Automation Programmer's Guide

Albertsons Group

April 24, 2018

Linus Vidal
linus.vidal@dimensiondata.com

Dimension Data Contact Details

We welcome any enquiries regarding this document, its content, structure or scope. Please contact:

Linus Vidal - Sr. Consultant,

Dimension Data North America, Inc.
5000 Hopyard Road,
Suite 450,
Pleasanton, CA 94588

✉ linus.vidal@dimensiondata.com

Please quote reference in any correspondence or order.

This document is valid until May 24, 2018 and is subject to the confidentiality and terms and conditions as per the formally agreed contract between Dimension Data North America, Inc., ("Dimension Data") and Albertsons Group ("Albertsons") or Dimension Data's standard terms and conditions (which are available on request) if no formal contract has been agreed.

Albertsons Meraki Automation Programmer's Guide

Document Configuration Management

Document Identification

File Name	Albertsons-Meraki-Automation-Programmers-Guide-v1.1.docx
Version	Version 1.1
Sensitivity Classification	Company Confidential - Client / Vendor Information
Document Owner	Linus Vidal

Preparation

Action	Name	Role / Function	Date
Prepared by:	Linus Vidal		April 24, 2018
Reviewed/Approved by:	Safeway Automation Team		April 24, 2018

Release

	Date Released	Change Notice	Remarks
1.1	April 24, 2018		1st Draft

Albertsons Meraki Automation Programmer's Guide

Contents

1. Design Overview 5

2. Automation Scripts Functions 6
2.1. Overview 6
2.2. CLI 6
2.3. Automation Layer 6
2.4. API Layer 7

3. Directory Structure of Scripts 8
3.1. Top level directories 8
3.2. “code” directory contents 8
3.3. Data Directories 10
3.4. “data” directory 10
3.5. “runtime” sub-directory contents 11
3.6. “doc” sub-directory contents 11

4. Store IP subnet generation and templates 12
4.1. Overview 12
4.2. VLAN generation Overview 13
4.3. Subnet generation process 13
4.4. Template Specialization 16

5. Convert and Validation logic 23
5.1. Schema Validation 23
5.2. “l3fwrules” validation 24
5.3. “s2svpnrules” validation 25

6. Increase the log messages and debugging scripts 27
6.1. Increase the log messages verbosity on Automation Server 27
6.2. Debugging on local laptop using Pycharm 27
6.3. Common production issues and possible remedies 28

1. Design Overview

This document has the details for design and functionality of Meraki automation scripts written in Python 3.

The system provides a simple “cli” interface that allows deploying networks (cloning stores), deploying stores (setup of vlan, static routes, vpn and l3fwrules) as well as updates of l3fwrules and s2svpnrules to store lists.

The system uses templates for vlans and l3fwrules, obtains subnet’s Safeway’s proprietary Netx conversion generates actual absolute vlans and l3 firewalls rules.

The documentation for the conversion process and logic is contained in the relevant files. (netx.py, vlans.py and vlans_handlers, and firewall_handlers.py) and is outside the scope of this document.

Additionally, the system provides “csv” to “json” conversion and validation (schema and firewall semantics) for the required files in order to sanitize input data.

Another aspect is keeping extensive logs for support, traceability, auditability and future enhancements.

At this point the system does not support multi-users, but the basic design was done considering future need for this enhancement.

2. Automation Scripts Functions

2.1. Overview

A “cli” controller controls and dispatches commands to the lower layers.

The structure is such that handlers take care of manipulating the data and calling API drivers. These API drivers essentially interface directly to Meraki-API and provide logging and isolation. An automation layer contains handlers for the various modules. The Automation layer is mostly concerned about generating proper API-Calls with proper inputs. It is also responsible for bulk and aggregated operations.

2.2. CLI

This is a layer which receives “cli” commands, does some initial parsing and passes valid commands down to the automation layer.

Note: A text UI (tui.py) framework was also developed and could be incorporated into the system if needed.

2.3. Automation Layer

This layer composed of several handlers is mainly concerned with bulk-updates (multi-store) and template specialization using Safeway’s proprietary Netx Logic for the stores (l3 and Vlan’s)

- Bulk update
- Firewall handler (l3rwrules)
- Network handler
  Access to the network/store layer, mostly bulk support
- Static route handler
- Store Orchestration
- Vlan Handler
- Vpn firewall handler (s2svpnrules)
- Vpn handler

2.4. API Layer

This layer contains the drivers to interface to external API's. It is mostly composed of Meraki API wrappers, but it also contains the Men and Mice interface and the Netx (Safeway's proprietary logic for store subnet generation)

In order to improve error logging and error logging capabilities the open-source Meraki-API driver is hot-patched (python monkey patching). This allows for the automation scripts to be run in dry-mode, which means that scripts and input data can be exercised without incurring delays due to real Meraki-API calls and being totally transparent to the Meraki network.

Below are the modules supported by the API layer:

- network
- netx
- static route
- vlans
- vpn

3. Directory Structure of Scripts

The directory structure for the automation scripts is described in this section.

3.1. Top level directories

Code contains following top level directories

- code directory
- data directory
- templates

3.2. “code” directory contents

Code directory contains following python scripts

- cli.py – cli handler.
- global_var.py – caters for global flags used in debugging. (see debugging hints for more info)
- “get, deploy, select, convert” - linux scripts to simply use of the cli for the get, deploy, select and convert functions.
- “VERSION.sh” – displays the git branch label for the current deployment
- tui.py – text ui handler framework, not yet integrated into the system.

Code contains following subdirectories.

- Code base directory
  - automation
  - api
  - utils
  - GUI
  - menAndMice

The details for each directory are given below

3.2.1. “automation” sub-directory contents

- bulk_update.py – handler for bulk updates and gets for l3 and vpn firewall

- firewall_handler.py – handler for the l3 firewall. Includes template (/templates/l3fwrules_template_xxx.json) specialization to create proper meraki api calls.
- network_handler – supports bulk cloning of stores/networks. (used by deploy networks)
- static_rouge – support for adding static routes
- store_orchestration – aggregation of multiple handlers, namely vlan_handler, static_route, l3fwrules and vpn_handler to support a single cli deploy stores bulk command.
- vlan_handler – specializes jinja_vlan_template using Safeway's Netx logic in order to create absolute value vlans for a give store/network.
- vpn_firewall_handler – support for getting and setting a vpn firewall for a given org.
- vpn_handler – implements setup of vpn for a given org.

3.2.2. "api" sub-directory contents

The directory "api" contains all the API drivers afore-mentioned.

- men_and_mice.py
- network.py
- vlans.py
- devices.py
- meraki.py
- netx.py
- vpn.py
- firewall.py
- meraki_patch.py
- static_route.py
- vpn_firewall.py

Please note that meraki.py is simply a copy of a Meraki release.(This is done so that Meraki updates do not impact the code, and also that there more control over the stability of the code, at any point a new version can be pulled in, but it is advisable that full regression testing should be done)

3.2.3. "utils" sub-directory contents

This directory contains general use utilities such as:

logging,

csv conversion and validation,

json,

jinja support,

global variables.

3.2.4. "GUI" sub-directory contents

This directory contains a flask python web-server, which was integrated for GUI testing. No longer supported, however it could be useful in for REST-API support development.

3.2.5. "menAndMice" sub-directory content

This directory contains Python 2.7 SOAP based men and mice client. Tested but not operational. Will need run as an external service periodically in order to provide full Men and Mice integration.

3.3. Data Directories

There are three data directories as shown below.

- templates
- data

3.3.1. "templates" directory

This directory contains the in-use definitions for store-lists, org-lists, l3fwrules, s2svpn rules.

Please note the "valid inputs. json" file contains a list of valid schemas that is used in the convert utility ("csv-to-json")

3.4. "data" directory

This directory contains run time data and logs generated by the scripts. This is useful for debugging and auditing.

3.4.1. "config" sub-directory contents

This directory contains system level configuration.

At this point the main function for this settings file is to support new Orgs.

Modify the "safeway-config.json" file in two places:

a) In the networks section as per example below:

```
"network": [
  {
      "org_name": "AutomationTestOrg_DONOTDELETE",
      "org_id" : 686798943174000795,
      "clone_store": "0_Base_Config_v1"
  },
```

b) In the vpn/hubnetworks section as per example below:

```
"vpn":
    {
      "hubnetworks" : [
        {
          "org_name": "Store_QA_Org",
          "id": [
            "N_686798943174004323",
            "N_686798943174004324"
          ]
        },
```

3.5. "runtime" sub-directory contents

This directory contains run time transient settings. Useful for debugging and monitoring.

3.6. "doc" sub-directory contents

Used for loose storage of development relevant documentation.

4. Store IP subnet generation and templates

4.1. Overview

The Meraki API for vlan creation and updates, uses a format, which requires absolute values which are specific for each store.

Additionally, the Meraki API for l3fwrules updates also requires absolute values.

One of the purposes of the automation scripts is to automatically generate the above files/formats based on templates and store name. This allows for bulk deployments and consistency across the network.

The subnet generation logic is as follows:

Stores map to device names, which translates into an IP address. Based on this IP the netx logic, explained below, creates eight subnets.

A men and mice funnel file, explained below, contains the last octet and mask, which together with the vlan_netx file provides all the information required for sub-netting. We call this process store-subnetting.

Template use is as follows:

The store subnetting is used to specialise a VLAN jinja template, which produces the required vlans_generated_<xx>, used to call the Meraki API

So, with inputs netx + funnel and using /config/jinja_vlans_template we obtain /data/…./vlans_generated which is used by the Meraki API to update and or create the vlans.

For l3fwrules the process is very similar but uses an /templates/l3fwrules_template_<xxx> provided and constantly updated by the Safeway Security Team. This template follows the Meraki standard for firewall templating. This choice of template format was done for the sake of consistency and the ability to easily compare Safeway Templates against Meraki deployed templates. This template is specialized by the automation scripts code instead of jinja, as jinja was not suitable for this task. The firewall_handler.py contains the code that specializes this template.

4.2. VLAN generation Overview

Store names will always be in the format <ABC>_<nnnn>, where ABC is the Safeway Division (e.g. SHA, JEW, etc.), this is always an uppercase non-numeric A-Z character and nnnn is the actual store four-digit number 0000-9999.

The store numbers are unique and each store has a network controller cc<nnnn> associated with it.

For example, the SHA_0012 store will have a cc0012 controller which should be visible on the Safeway network.

The VLAN generation is done using the following steps:

- From the Men and Mice external utility a vlans_funnel.csv file is obtained. This file is patched using the config/vlan_funnels.patch.csv, which contains internal, across store constant addresses in the 192.168 subnet.
- The first column is the VLAN number, the second line which follows the format

  10.x.a.nn/nn is the subnetting.

  10.x.a is translated into a select for the subnet “a” (three octets subnet created by the netx function)
- Conversely 10.x.b. selects subnet “b”, and so forth up to 10.x.h which selects subnet “h”.
- The last octet is exactly that the last octet (4th octet, plus mask)

4.3. Subnet generation process

This entails three steps which are detailed below:

- Netx IP subnet generation
- Funnel generation
- Template Specialization

4.3.1. Netx IP subnet generation

The vlan_netx.json describes 8 subnets for a given store.

It is easier to understand the logic by example:

```
How to generate netx by example
a) ping cc8501
```

```
b) Pinging cc8501 [10.218.31.5] with 32 bytes of data:
c) netx["upper"] = {1:10, 2:218, 3:28, 4:0}   (31-3 = 28 for 3rd octet)
d) netx["lower"] = {1:10, 2:154, 3:28, 4:0}   (218-0x40=154 for 2nd
octet)
3:28} (copy 1,2,3 octets from netx["upper"]
f) next{"b"] = {1:10. 2:218. 3:29} (copy 1,2 octets from netx["a"] and
3rd octet = netx["a"][3] + 1
g) next{"c"] = {1:10. 2:218. 3:30} (copy 1,2 octets from netx["b"] and
3rd octet = netx["b"][3] + 1
e) next{"d"] = {1:10. 2:218. 3:31} (copy 1,2 octets from netx["c"] and
3rd octet = netx["c"][3] + 1

e) next{"e"] = {1:10. 2:154. 3:28} (copy 1,2,3 octets from netx["lower"]
f) next{"f"] = {1:10. 2:154. 3:29} (copy 1,2 octets from netx["a"] and
3rd octet = netx["f"][3] + 1
g) next{"g"] = {1:10. 2:154. 3:30} (copy 1,2 octets from netx["b"] and
3rd octet = netx["g"][3] + 1
e) next{"h"] = {1:10. 2:154. 3:31} (copy 1,2 octets from netx["c"] and
3rd octet = netx["b"][3] + 1

e) next{"a"] = {1:10. 2:218.
So for
ip = 10.218.31.5
netx = {
    "upper": "10.218.28.0",
    "lower": "10.154.28.0",
    "a": "10.218.28",
    "b": "10.218.29",
    "c": "10.218.30",
    "d": "10.218.31",
    "e": "10.154.28",
    "f": "10.154.29",
    "g": "10.154.30",
    "h": "10.154.31"

}
```

Below the generated vlans_netx.json

```
{
    "upper": "10.195.200.0",
    "lower": "10.131.200.0",
    "a": "10.195.200",
    "b": "10.195.201",
    "c": "10.195.202",
    "d": "10.195.203",
    "e": "10.131.200",
    "f": "10.131.201",
    "g": "10.131.202",
    "h": "10.131.203"

}
```

4.3.2. "vlans_funnel" generation

The vlans_funnel.csv (copied from ./menAndMice/funnel.csv is patched with ./config/vlans_funnel_patch.csv and is ultimately an old-style template. The first column is the vlan number, the second column contains the 3 octet subnet reference and the

actual last octet + mask. (e.g. 10.x.a.16/27, translates to use netx subnet “a” with the last octet as 16 and mask 27)

The netx logic explained below produces 8 subnets , named “a”, “b“, “c’, “d’, “e”, ”f”, ”g”, ”h”, for a given absolute fixed IP.

Therefore, the netx logic and the funnel-vlan logic above provide subnets which are used in VLAN and Firewall Rule generation.

Below a sample vlans_funnel.csv obtained from Men and Mice.

```
Vlan,Subnet,Description
1,10.x.a.16/27,Network Management
4,10.x.a.32/27,Network Management
6,10.x.a.64/27,Extranet Vendors
7,10.x.a.96/27,Pharmacy
8,10.x.a.128/27,Backstage Users
14,10.x.a.248/29,Store Content Engine
16,10.x.b.0/24,Store Wireless
19,10.x.c.0/24,General Store LAN
24,10.x.d.0/24,POS LAN
35,10.x.e.0/27,Digital Signage
40,10.x.e.32/27,Thin Wireless Mgmt.
45,10.x.h.192/26,Cisco Wireless Management
60,10.x.h.128/27,Vendor VLAN II (2nd Subnet)
70,10.x.e.64/26,New Pharmacy
75,10.x.e.128/25,Macauthenticated Wireless Clients
79,10.x.g.192/26,VOIP Clients
80,10.x.f.0/24,Secure Wireless
81,10.x.g.160/28,Field Service Wireless
82,10.x.g.176/28,iPad quarantine vlan
85,10.x.h.0/26,Retail Cluster Management VLAN
95,10.x.g.128/27,Printer VLAN
```

The above vlans_funnel.csv file is patched with the patch below:

vlans_funnel_patch has vlans where subnets do not change per store

```
992,"192.168.192.0/24","Digital Signage - I"
995,"192.168.1.0/24","Guest WIFI"
996,"192.168.100.0/24","Cache VPN"
997,"192.168.101.0/24","Cache Internet"
```

```
Vlan,Subnet,Description
1,10.x.a.16/27,Network Management
4,10.x.a.32/27,Network Management
6,10.x.a.64/27,Extranet Vendors
7,10.x.a.96/27,Pharmacy
8,10.x.a.128/27,Backstage Users
14,10.x.a.248/29,Store Content Engine
16,10.x.b.0/24,Store Wireless
19,10.x.c.0/24,General Store LAN
24,10.x.d.0/24,POS LAN
35,10.x.e.0/27,Digital Signage
40,10.x.e.32/27,Thin Wireless Mgmt.
45,10.x.h.192/26,Cisco Wireless Management
60,10.x.h.128/27,Vendor VLAN II (2nd Subnet)
70,10.x.e.,New Pharmacy64/26
75,10.x.e.128/25,Macauthenticated Wireless Clients
79,10.x.g.192/26,VOIP Clients
80,10.x.f.0/24,Secure Wireless
81,10.x.g.160/28,Field Service Wireless
82,10.x.g.176/28,iPad quarantine vlan
85,10.x.h.0/26,Retail Cluster Management VLAN
95,10.x.g.128/27,Printer VLAN
96,10.x.g.178/27,Printer VLAN
992,"192.168.192.0/24","Digital Signage - I"
995,"192.168.1.0/24","Guest WIFI"
996,"192.168.100.0/24","Cache VPN"
997,"192.168.101.0/24","Cache Internet"
```

4.4. Template Specialization

The vlan generation is a two-step template specialization process.

The first step uses netx as input data which is applied to the vlans_funnel_netx and creates a vlan_funnels_table.

For vlan generation the second step uses vlan_funnels_table and applies it to jinja_vlans_template and obtains vlan_generated_<netid>

For L3fwrules generation the second step uses vlan_funnels_table and applies it to l3fwrules_template_<desc> and obtains l3fwrules_deploy_<store-number>

4.4.1. Process Steps

- Create vlans_funnel_subnet:

  A vlans_netx subnet file, described in the previous chapter is used to specialize a vlans_funnel file, described in previous chapter, and it produces a vlans_funnel_subnet file as depicted below:

  Vlan,Subnet,Description
  1,10.218.28.16/27,Network Management

- Create vlans_funnel_table:

  vlans_funnel_subnet is used to create vlans_funnel_table. which provides a simple json subnet vlan lookup file as depicted below:

  { "1": "10.218.28.16/27", "4": "10.218.28.32/27", "6": "10.218.28.64/27",... }

- Vlan generation - create vlans_generated_<netid>:

  This is the file which is used for the Meraki API call.

  Using the jinja the /config/jinja_vlans_template.json and the vlans_funnel_table the vlans_generated_<netid> is created

- L3 firewall generation - create l3fwrules_deploy_<store-number>:

  This is the file which is used for the Meraki API call.

  Using the logic in /automation/firewall_handler and a selected templated provided in /templates/l3fwrules_template_<desc> and the vlans_funnel_table the l3fwrules_deploy_<store-number>.

4.4.2. Create vlans_funnel_subnet

/data/../vlan_funnel_netx is created from vlans_funnel by simply removing irrelevant information. A snapshot of vlans_funnel_netx is depicted below.

```
Vlan,Subnet,Description
1,a.16/27,Network Management
4,a.32/27,Network Management
6,a.64/27,Extranet Vendors
7,a.96/27,Pharmacy
```

By specializing vlans_funnel_netx, vlans_funnel_subnet is obtained.

```
Vlan,Subnet,Description
1,10.195.200.16/27,Network Management
4,10.195.200.32/27,Network Management
6,10.195.200.64/27,Extranet Vendors
7,10.195.200.96/27,Pharmacy
8,10.195.200.128/27,Backstage Users
14,10.195.200.248/29,Store Content Engine
16,10.195.201.0/24,Store Wireless
19,10.195.202.0/24,General Store LAN
24,10.195.203.0/24,POS LAN
35,10.131.200.0/27,Digital Signage
40,10.131.200.32/27,Thin Wireless Mgmt.
45,10.131.203.192/26,Cisco Wireless Management
60,10.131.203.128/27,Vendor VLAN II (2nd Subnet)
70,10.131.200.64/26,New Pharmacy
75,10.131.200.128/25,Macauthenticated Wireless Clients
79,10.131.202.192/26,VOIP Clients
80,10.131.201.0/24,Secure Wireless
81,10.131.202.160/28,Field Service Wireless
82,10.131.202.176/28,iPad quarantine vlan
85,10.131.203.0/26,Retail Cluster Management VLAN
95,10.131.202.128/27,Printer VLAN
992,192.168.192.0/24,Digital Signage - I
995,192.168.1.0/24,Guest WIFI
996,192.168.100.0/24,Cache VPN
997,192.168.101.0/24,Cache Internet
```

4.4.3. Create vlans_funnel_table

The file “vlans_funnel_subnet”, which is a list, is transformed into a table

Below vlans_funnel_table (note that the description info field is being dropped and the description is being picked up from the jinja template file), this maps vlans to respective subnets.


```
{
    "1": "10.195.200.16/27",
    "4": "10.195.200.32/27",
    "6": "10.195.200.64/27",
    "7": "10.195.200.96/27",
    "8": "10.195.200.128/27",
    "14": "10.195.200.248/29",
    "16": "10.195.201.0/24",
    "19": "10.195.202.0/24",
    "24": "10.195.203.0/24",
    "35": "10.131.200.0/27",
    "40": "10.131.200.32/27",
    "45": "10.131.203.192/26",
    "60": "10.131.203.128/27",
    "70": "10.131.200.64/26",
    "75": "10.131.200.128/25",
    "79": "10.131.202.192/26",
    "80": "10.131.201.0/24",
    "81": "10.131.202.160/28",
    "82": "10.131.202.176/28",
    "85": "10.131.203.0/26",
    "95": "10.131.202.128/27",
    "992": "192.168.192.0/24",
    "995": "192.168.1.0/24",
    "996": "192.168.100.0/24",
    "997": "192.168.101.0/24"
}
```

4.4.4. Vlan generation - create vlans_generated_<netid>

Also note that the jinja template uses the 3 octets for the appliance-IP and this is generated by the code on the flight, so there is no file mapping for this.

The network ID is also picked up on the flight and uses the correct Meraki-netid for that particular store.

So now we have to generate a proper specialized version of the vlans which can be applied to Meraki. The template for Meraki Vlans is in /config/jinja_vlans_template.json

```
{
  "id": 19,
  "networkId": "{{networkid}}",
  "name": "generalstorelan",
  "applianceIp": "{{vlan[19]['octets']}}.1",
  "subnet": "{{vlan[19]['subnet']}}",
  "dnsNameservers": "upstream_dns",
  "fixedIpAssignments": {},
  "reservedIpRanges": []
},
{
  "id": 24,
  "networkId": "{{networkid}}",
  "name": "storelan",
  "applianceIp": "{{vlan[24]['octets']}}.1",
  "subnet": "{{vlan[24]['subnet']}}",
  "dnsNameservers": "upstream_dns",
  "fixedIpAssignments": {},
  "reservedIpRanges": []
},
```

The final output is vlans_generated_N_686798943174007640.json and is used against the Meraki API

```
[
    {
        "id": 4,
        "networkId": "N_686798943174007640",
```

```
            "name": "networkmgmt",
            "applianceIp": "10.195.200.33",
            "subnet": "10.195.200.32/27",
            "dnsNameservers": "upstream_dns",
            "fixedIpAssignments": {},
            "reservedIpRanges": []
        },
        {
            "id": 6,
            "networkId": "N_686798943174007640",
            "name": "extranetvendor",
            "applianceIp": "10.195.200.65",
            "subnet": "10.195.200.64/27",
            "dnsNameservers": "upstream_dns",
            "fixedIpAssignments": {},
            "reservedIpRanges": []
        },
        {
            "id": 7,
            "networkId": "N_686798943174007640",
            "name": "pharmacy",
            "applianceIp": "10.195.200.97",
            "subnet": "10.195.200.96/27",
            "dnsNameservers": "upstream_dns",
            "fixedIpAssignments": {},
            "reservedIpRanges": []
         }
    ]
```


4.4.5. L3 firewall generation - create l3fwrules_deploy_<store-number>

Applying the vlans_funnel_table to the selected l3fwrules_template_<desc>, generates the l3fwrules_deploy depicted below:


```
[
{
"comment": "413",
"policy": "allow",
```

```
"protocol": "tcp",
"srcPort": "Any",
"srcCidr": "10.218.31.23/32",
"destPort": "58019",
"destCidr": "10.154.28.126/32",
"syslogEnabled": false
},
```

5. Convert and Validation logic

The CLI “convert csv-to-json” and “convert json-to-csv” was developed to convert “csv” format file to “json” format file as accepted by Meraki API.

Conversion is supported for following file types.

- l3fwrules
- s2svpnrules
- store-list
- org-name

There is no validation built-in in convert json-to-csv.

Validation is built-in into convert csv-to-json.

There are to levels of validation, schema validation which applies to all the files types and firewall validation.

5.1. Schema Validation

For every file there is a schema defined in /templates/valid_inputs.json
This allows for validation-based file name standards, required fields and field content type. For new file types this schema file should be amended.

A snapshot of valid_inputs.json is depicted below:

For every file there is an entry in the schema defined in /templates/valid_inputs.json

This allows for validation-based file name standards, required fields and field content

```
[
{
 "l3fwrules" : {
        "fname_pattern" : "l3fwrules_template_",
        "json_schema" : {
            "type": "object",
                "properties": {
                    "comment": {"type": "string"},
                    "policy": {"enum": ["allow", "deny"]},
                    "protocol": {"enum": ["udp", "tcp", "any", "icmp",
"Any"]},
                    "srcPort": {"type": "string"},
                    "srcCidr": {"type": "string"},
```

```
                        "destPort": {"type": "string"},
                        "destCidr": {"type": "string"},
                        "syslogEnabled": {"type": "string"}
                    }
                }
            },
        ]
```

5.2. “l3fwrules” validation

Follows validation done on l3fwrule fields

- “syslogEnabled”

Forces it to be always lower case “false” and “true”

- “syslogEnabled”

  Forces it to be always lower case “false” and “true”
- “srcCidr”, “destCidr”, "comment", "srcPort", "destPort"

  Remove “\n” (spurious fake carriage-return inserted by Excel)
- "protocol", "srcCidr", "destCidr", "comment", "srcPort", "destPort"

  Changes “Any” to “any”
- "destPort", "srcPort"

  Limit field to be either:
  - “any”
  - Numeric range number1-number2, where number1 and number are in the range of 1-65535 (e.g 1-24000)
  - A list of comma separated ports, where ports are in the range of 1-65535. (e.g. 1, 65000, 234)

- “destCidr”

  Limit vlan and IP to be:

  A list containing valid vlans templates and IPs.

  Valid vlan templates should be in the format VLAN(nnn).zzz, where “nnn” is a non-zero number and “zzz” can be “*” or non-zero number.

  Valid IPs should be in one of the formats below:

  o1.o2.o3,o4/nn,

  o1.o2.o3.o4 mask m1.m2.m3.m4

  “any”

  Where nn is within the range of 1-32 and o1, o2, o3, o4 and

  m1, m2, m3, m4 are in the range 0-255

  Also, destCidr is not allowed to be blank.

- “srcCidr”

  Limit vlan and IP to be:

  A list containing valid vlans templates and IPs.

  Valid vlan templates should be in the format VLAN(nnn).zzz, where “nnn” is a non-zero number and “zzz” can be “*” or non-zero number.

  Valid IPs should be in one of the formats below:

  o1.o2.o3,o4/nn,

  o1.o2.o3.o4 mask m1.m2.m3.m4

  “any”

  Where nn is within the range of 1-32 and o1 is 192 and o2 is 168.

5.3. “s2svpnrules” validation

- “destCidr”, “srcCidr”

Limit vlan and IP to be:

Limit IP to be:

A list containing valid IPs.

Valid IPs should be in one of the formats below:

o1.o2.o3,o4/nn,

o1.o2.o3.o4 mask m1.m2.m3.m4

“any”

Where nn is within the range of 1-32 and o1, o2, o3, o4 and m1, m2, m3, m4 are in the range 0-255

Also, destCidr is not allowed to be blank.

6. Increase the log messages and debugging scripts

6.1. Increase the log messages verbosity on Automation Server

When running in production log messages a kept to a minimum. In the case of errors where the messages are not sufficient it is possible to increase the verbosity by changing a variable in the global_vars.py

Simply change the line

```
log_verbose = False
to
log_verbose = True
```

Do not forget to change it back once the problem has been resolved.

6.2. Debugging on local laptop using Pycharm

Given that debug capabilities on the automation server are very limited, it might be necessary to resort to Pycharm in order to seek a faster resolution/debug to the problem.

The steps should be the following:

- An existing Pycharm environment should be already setup. It is outside the scope of this document how to setup such an environment.
- The version of code in nsmk-qa/code should be the same as the one which is being used in production.

These are the steps to obtain a sync-up version on your environment.

On the server run ./VERSION.sh

(same as doing a git describe –tags)This will provide which version is running on the server.

<VERSION_IN_PRODUCTION> e.g. P25

On your machine clone the same version: git clone -b <VERSION_IN_PRODUCTION> https://github.com/<repo> code

Now you are ready to start debugging the code.

If the problem is a crash, simply put a breakpoint just before the crash and attempt to find the problem that way.

6.3. Common production issues and possible remedies

It is always advisable to deploy l3 firewall and s2svpn rules against a test org.

- Deploy l3fwrules fails
    a. The networks/stores have been corrupted.
    b. Delete and re-deploy networks and stores again.
    c. An actual problem with the l3fwrules_template.json file.
    d. Ensure the converted csv-to-json is passing without errors, watch for the actual error messages from the convert script. (A message will point you to a line that can be offset but one of two depending if you are using csv or excel source input)
- Deploy s2sfwrules fails
    a. An actual problem with the l3fwrules_template.json file.
    Ensure the convert csv-to-json is passing without errors, watch for the actual error messages from the
    b. Script. (The message will point to a line that can be offset but one of two depending if you are using csv or excel)