mxfirewallcontrol.py

A multi-organization, multi-network Meraki MX Layer 3 firewall control script in Python 3

mxfirewallcontrol.py is a script to view, create backups for and make changes to Meraki MX Layer 3 firewall rulesets across multiple organizations, networks and templates. It can be used both as a command-line utility and as a back-end process to create custom management portals.

This manual is updated to reflect changes in the script. The version of this manual is 1.1.

You can get the latest version of this manual here: [insert link]

Installation

To run the script, Python 3 and the Requests module will need to be installed. More information on installing these components for your operating system can be found here:

https://www.python.org/downloads/

http://docs.python-requests.org/en/master/user/install/

The script was developed and tested using Python 3.5.3 on Windows 10.

The Meraki Dashboard API

The script uses the Meraki Dashboard API to get information from and make changes to the configuration of elements in the Meraki cloud. You will need a Meraki Dashboard API key in order to run the script. More information on the Dashboard API, how to enable it for an organization and how to create a key can be found here:

https://documentation.meraki.com/zGeneral_Administration/Other_Topics/The_Cisco_Meraki_Dashboard_API

The script requires organization-level access privileges to run. Dashboard API keys have the same privilege level as the administrator account they are tied to. An explanation of different Dashboard administrator privilege levels can be found here:

https://documentation.meraki.com/zGeneral_Administration/Managing_Dashboard_Access/Managing_Dashboard_Administrators_and_Permissions

Script usage

To run the script, run the command "`python mxfirewallcontrol.py`" in the directory where the script resides, followed by the correct command-line arguments:

```
python mxfirewallcontrol.py -k <key> -o <org> [-f <filter>] [-c <command>] [-m <mode>]
```

Elements marked *<value>* represent values that need to be entered. Arguments in brackets *[]* are optional. Here is an example of running the script:

```
python mxfirewallcontrol.py -k 1234 -o "Meraki Inc" -c create-backup
```

That command would run the script with API key `1234`, find an organization named `Meraki Inc` and create a local backup copy of the MX Layer 3 firewall rules for all of its configuration templates. You can find more information on the available command line arguments in the *Command line arguments* section of this manual.

Running the script with no arguments will print the help text.

Using the script as a utility VS as a headless backend process

You can use the script both as a command line utility and as a backend process for creating custom management portals. The script supports two ways of input for large blocks of data:

- Input files
- JSON formatted strings as command line arguments

Depending on your intended use of the script, one or the other method may suit your purpose better. If using the script as a command line utility, a firewall ruleset is probably easier to define in an input file using a text editor. More on these input methods in the *Command line arguments* section of this manual, under *Issuing commands*.

Command line arguments

The script includes both mandatory and optional arguments. If mandatory arguments are omitted, the script will only display the help text, without attempting any other operations. If optional arguments are omitted, the script will execute according to their default values.

Arguments may require values that include spaces. Operating systems can typically pass those by using single or double quotes (`'` or `"`). For example by writing `-o "Meraki Inc"` you can define an organization name that includes a space.

JSON values can include the double quote character. To pass that in a command line argument that includes both spaces and double quotes, you will need to either use the single quote character to enclose the string, or to escape the double quote character according to how your operating system requires. For example in Windows you need to use double double quotes:

```
-c "append:""port"":""any"",""srcCidr"":""any"", [Input omitted]"
```

The mandatory arguments are listed in the table below:

`-k <key>`	Your Meraki Dashboard API key
`-o <org>`	The name of the Meraki dashboard organization you want to process. Enter `/all` for all

Note that if you want to process multiple organizations, you can enter the value `/all` instead of an organization name. In this case, all of the organizations accessible by your administrator account will be processed. Both `/all` and organization names can be combined with filters to limit the scope of networks and templates to be processed. More on filters below.

In addition to the mandatory arguments, you can provide optional arguments to define the operations carried out by the script. If no optional arguments are given, the script will read the MX Layer 3 firewall rules for all templates in the specified organizations and print them.

The three optional arguments are listed in the table below:

`-f <filter>`	Define a subset of networks or templates to be processed. To use multiple filters, separate them with commas. A network/template needs to satisfy all filters to be processed.
`-c <command>`	Specify the operation to be carried out.
`-m <mode>`	Define whether commands that modify firewall rulesets will upload changes to cloud.

Using filters

The argument `-f <filter>` defines which networks and templates in the target organizations will be selected for processing. If the argument is omitted, it defaults to `-f type:template`. You can use commas (,) to specify multiple filters.

The available types of filters are listed in the table below:

`name:<name>`	Network/template name must match the value specified in `<name>`. Use `*` for wildcard. The wildcard character only allowed in the beginning or end of a string.
`tag:<tag>`	To be selected, a network needs to have a network tag matching `<tag>`. This filter is not compatible with filters `type:template` and `type:any`.
`type:network`	Process only non-template networks.
`type:template`	Process only configuration templates. This is the default filter. Cannot be combined with filter `tag:<tag>`.
`type:any`	Process both networks and config templates. Cannot be combined with filter `tag:<tag>`.

An example of using multiple filters:

```
python mxfirewallcontrol.py -k 1234 -o /all –f “type:network,tag:branch”
```

The name filter supports wildcard searches. The wildcard character is the asterisk `*`, which can be used at the beginning and/or the end of a string to match. Examples of valid use of the wildcard character:

`name:*security`	Matches all elements with a name that ends with “security”
`name:security*`	Matches all elements with a name that starts with “security”
`name:*security*`	Matches all elements with a name that includes “security”
`name:adv*,name:*security`	Matches all elements with a name that starts with “adv” and ends with “security”

Issuing commands

The argument `-c <command>` defines which operation will be carried out by the script. One command can be entered at a time. If the argument is omitted, it defaults to `-c print`.

In the Meraki Dashboard, MX Layer 3 firewall configuration is expressed as rulesets. The ruleset is composed of multiple firewall rules, which are applied to traffic in a sequence. You can find the sequence number of a rule by viewing the ruleset in Dashboard, or by using the `print` and `create-backup` functions of this script.

When referring to rules in a ruleset by sequence number, you can use positive or negative sequence numbers. A positive number indicates counting from the start of the ruleset. A negative number indicates counting from the end of the ruleset. The sequence number of the first rule in a ruleset is 1. The sequence number of the last rule in a ruleset is -1.

When inserting rules, if the index provided is out of range, the rules will be inserted to the end or beginning of the ruleset, depending on the direction of counting. When removing rules, an invalid index for a network or template will cause that item to be skipped without processing.

The following table lists the valid options for the argument `-c <command>`:

`print`	Do not make changes, just print the ruleset to screen. This is the default command.
`create-backup`	Save rulesets in a local folder. The name of the folder created is `mxfirewallctl_backup_<timestamp>`. The `<timestamp>` is expressed as YYMMDD_HHmmSS. The filenames created are in the format `<org name>__<net name>.txt`. The file naming format is the same as the one used by `load-folder`.
`load-folder:<folder>`	Replace rulesets in scope by the ones contained as text files in folder `<folder>`. This function will look for filenames with a naming format of `<org name>__<net name>.txt`. The intent of this command is to reupload files created by `create-backup` easily.
`append:<rules>`	Add `<rules>` to the end of ruleset. The rules are entered as a single JSON formatted string.
`append-file:<filename>`	The ruleset defined in `<filename>` will be appended to existing rulesets in scope.
`insert:<num>:<rules>`	Insert `<rules>` as rules starting with line number `<num>`. The rules are entered as a single JSON formatted string. `<num>` can be a

	positive or negative integer and indicates the sequence position where the rules will be inserted.
`insert-file:<num>:<filename>`	Insert rules defined in `<filename>` into the rulesets in scope, in sequence position starting with line number `<num>`. A positive `<num>` indicates counting from the start of the ruleset. A negative `<num>` indicates counting from the end of the ruleset.
`replace:<rules>`	Replace rulesets in scope by the one given as a single JSON formatted string.
`replace-file:<filename>`	Replace rulesets in scope by the one defined in file `<filename>`.
`remove:<num>`	Remove rule with sequence number `<num>` from rulesets in scope. Positive and negative rule counting supported.
`remove-marked:<label>`	Remove rules which have a comment field that includes the character sequence `<label>`. The intent of this command is to clean up rulesets easily from lines added to tackle a temporary need or incident.
`remove-all`	Remove all rules from rulesets in scope.
`default-allow`	Check if the last rules in the rulesets in scope are “deny any” and remove them if such rules are found.
`default-deny`	Add a “deny any” rule to the end of the rulesets in scope.

Using modes

The argument `-m <mode>` can be used to specify whether commands that can affect device configuration in cloud, such as append, insert, will commit changes and whether they create a backup first. If the argument is omitted, it defaults to `-m simulation`.

Valid options for argument `-m <mode>` are:

`simulation`	Print changes for review, without applying to cloud. This is the default mode.
`commit`	Create backup and apply changes to cloud.
`commit-no-backup`	Apply changes to cloud without creating a backup.

Expressing rulesets

Firewall rulesets can be defined for processing either as input files or as JSON formatted strings. To read more about which commands use which form, see section *Issuing commands* above.

Input files are text files, with one rule per line in JSON format. You must insert a new line character after the last rule definition in the file, for example by pressing Enter in your text editor. An input file that defines a ruleset with two firewall rules can look something like this:

```
{"protocol":"any", "srcPort":"Any", "srcCidr":"10.1.1.1", "destPort":"Any",
     "destCidr":"any", "policy":"deny", "syslogEnabled":false, "comment":"Line 1"}
{"protocol":"any", "srcPort":"Any", "srcCidr":"10.2.2.2", "destPort":"Any",
     "destCidr":"any", "policy":"deny", "syslogEnabled":false, "comment":"Line 2"}
```

In the example above, the whole section between *braces {}* should be written as a single line of the input file. Note that while all other items are type *string* and should be enclosed in double quotes, *syslogEnabled* is type *boolean* and should be written in lower case, without quotes.

The keyword “any” can be written in any combination on uppercase and lowercase letters.

A Windows command that replaces a ruleset with two lines defined as a command line argument could look like this:

```
python mxfirewallcontrol.py -k 1234 -o "Meraki Inc" -c replace:"[{""protocol"":""any"",
      ""srcPort"":""Any"", ""srcCidr"":""10.0.0.1"", ""destPort"":""Any"",
      ""destCidr"":""any"", ""policy"":""deny"", ""syslogEnabled"":false,
      ""comment"":""Line 3""},{""protocol"":""any"", ""srcPort"":""Any"",
      ""srcCidr"":""10.0.0.2"", ""destPort"":""Any"", ""destCidr"":""any"",
      ""policy"":""deny"", ""syslogEnabled"":false, ""comment"":""Line 4""}]" -m commit
```

Enter the whole command as a single line.

Note that backups created by this script may include the “allow any” final rule that is automatically appended by Dashboard. The script removes these “allow any” rules before sending the ruleset to Dashboard, so you do not need to worry about these being duplicated.

Troubleshooting

Common issues that can result in errors or warnings when executing the script:

- The administrator account linked to your API key does not have access to the organization you want to modify
- The administrator account linked to your API key has only network level access to the organization you want to modify
- The organization you want to modify does not have API access enabled
- A network or template in the filter scope does not include security appliance configuration
- The workstation or server running a script does not have connectivity to Dashboard, due to firewalling or routing issues
- You are applying a *tag* filter to configuration templates
- Rulesets are entered in incorrect form
- Input files are missing or the script has insufficient privileges to read or write to disk