usagestats.py

A user group network usage report script in Python 3

usagestats.py is a script to run comparison reports on network traffic volume generated by different user groups across multiple organizations and networks. It can be used to run a quick report over a certain period of time or can be scheduled to generate a regular report.

This manual is updated to reflect changes in the script. The version of this manual is 1.0.

You can download the latest version of the manual here: https://github.com/meraki/automation-scripts/blob/master/usagestats_manual.pdf

Prerequisites

To run the script, you need to install Python 3 and the Requests module first. 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/

To install the script itself, copy the script and the sample init file into a directory on your workstation or server from Github:

https://github.com/meraki/automation-scripts/blob/master/usagestats.py

https://github.com/meraki/automation-scripts/blob/master/usagestats_initconfig.txt

The script was developed and tested using Python 3.6.4 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

How the script works

Operation basics

The script works by sending calls to the Meraki cloud using the Dashboard API to get per-client usage information. After it receives this information, the script figures out which group the client device belongs to by examining its IP address or VLAN ID. Usage data is summed up by group and reported to the user as command line output or an email.

The API call used is described here: https://dashboard.meraki.com/api_docs#list-the-clients-of-a-device-up-to-a-maximum-of-a-month-ago. The call supports requesting all usage data for the last X seconds, up to 30 days in the past.

The script calculates data for a specific day by requesting data for the moment the day began and the moment the day ended. After that, it subtracts usage volume since the day ended from usage volume since the day started.

The results of these calculations are stored in a SQLite3 database, as one record per network per group per day. The database file is used to:

- Extend reporting beyond the maximum of 30 days supported by Dashboard
- Make visualizing the data with third party interfaces easier. For example, you can use any database viewer that supports SQLite3 to create graphic representations of the usage data.

Accuracy

There are multiple factors that can influence the accuracy of data generated by this script:

- The method used to extract usage data from dashboard introduces some inaccuracy. High network usage by the clients tracked at the exact moment that the script is run, can cause traffic volumes to appear higher than they are in reality. This can be mitigated by running the *sync* operation often, which can reduce the risk of this type of random event influencing a large set of data.
- The daily volume calculation algorithm uses floating point operations.
- The database might not contain data for the full time range that a report is run for. Make sure to run the *sync* operation at least once every 29 days, in order to prevent time gaps in the database. It is recommended to *sync* the database more often than that, for example once a week. Running *sync* more often than once per day offers no benefit.

- The script uses a client's last known IP address or VLAN to classify which user group it belongs to. Running the *sync* operation frequently can reduce the risk of misclassifying a client that has, for some reason, acquired an IP address of a group it does not usually belong to, over a longer period of time.
- Selecting the wrong devices to pull information from can result in double-reporting of clients, which can significantly influence reporting results. If you are interested in WAN usage, the best devices to pull data from are WAN edge devices, like MX security appliances. If you are interested in LAN traffic volumes as well, wireless access points or a core switch can be the most suitable devices. Use the *filter* parameter when creating a database to define which devices data should be pulled from. More on this parameter under *Using the script > Using filters* in this manual.

Using the script

To run the script, open a command line window to the directory where you stored the script and run the following command:

```
python usagestats.py -k <key> [-d <database file name> -c <command> -i <initfile> -g
          <groups> -f <filter>] [-u <user> -p <pass> -r <recipient> -s <server>]
```

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

```
python usagestats.py –k 1234 –d mydatabase
```

This command runs the script using configuration and data stored in a database with filename *mydatabase*. It first *syncs* the database and after that runs a report with the default reporting settings (*report:last-week*).

It is recommended that you set the script to run as a scheduled task. If you want to have a monthly report of network usage per user group, you could set two scheduled tasks: One task would run the script every Saturday evening to *sync* the database while another would run it again with different command line parameters to generate and email a usage report every 2nd day of every month.

Using command line parameters

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.

The mandatory arguments are listed in the table below:

`-k <key>`	Your Meraki Dashboard API key

For the script to run successfully, you will also need to define a SQLite3 database file to store configuration and usage data. You can define the database file either as a command line option, using optional argument `–d`, or as a record in an init config file. More on these options below. To start a new database, just enter a filename that is not in use and the script will create the right database file for you.

The script's optional arguments are listed in the table below:

`-d <database file name>`	Specify the database file to be used. If omitted, the script will try to read a database file definition in the *init config file*.*
`-c <command>`	Specify the operation to be carried out.
`-i <initfile>`	Specify the init config file to be used to load initial configuration.
`-g <groups>`	Specify the Client Device Groups to be tracked. If omitted, a single group will be tracked, containing all clients. *+
`-f <filter>`	Specify the organizations, networks and devices usage data will be pulled from. If this parameter is omitted, the script will process all MX and Z appliances in all networks, across all organizations accessible by the admin account running the script.*+
`-u <user>`	Email account (username) that will be used to send email reports. This parameter is optional, but needed to send emails.
`-p <pass>`	Password for the email account that will be used to send email reports. This parameter is optional, but needed to send emails.
`-r <recipient>`	Recipient email address to send email reports to. This parameter is optional, but needed to send emails.
`-s <server>`	Email server to use for emailing reports. If omitted, Gmail will be used.

* Asterisk: This parameter can be defined in an *init config file*.

+ Plus sign: This parameter will be stored in the *database file*. You will need to run the script with the command line option `-c dbreconfigure` to change these values on an existing database.

Using an init config file

An *init config file* is a text file that can be used to define the initial parameters to use when creating a new database. Its purpose is to make entering *group* and *filter* definitions easier. The script will store parameters given in the *init config file* into the database, so the file is not needed after the database is created.

To write an init config file, create a new text document with a text editor. The init config file can have two sections, one defining groups and a second one defining other options. Each section spans multiple lines in the file and must have a section label as its first line. You can add comments by entering lines that have a hash (#) as the first character. The init config file format looks like this:

```
#This is a comment line
[GROUPS]
groupname=Corporate
subnet=10.0.0.0/8

groupname=Guest
subnet=192.168.0.0/16
```

```
[OPTIONS]
filter=dtag:statsmachine
```

You can find an example of an init config file here:

https://github.com/meraki/automation-scripts/blob/master/usagestats_initconfig.txt

Defining groups

Groups are collections of subnets or VLANs that you want to track network usage for. Each group can contain multiple subnets and VLANs. For each subnet or VLAN that you want to track, you will need to define either its IPv4 subnet or VLAN ID or VLAN name. Only one of those is required. You do not need to define all three.

You can define *groups* as either a command line parameter or an *init config file* section. You only need to use one or the other method, not both. After a database file is created for your reporting project, group definitions will be stored in the database, so you no longer need to define them when using an existing database file.

The script will attempt to match client devices to *groups* in order. If a match is found, no further processing will be done for that client. For example: your organization uses the 10.0.0.0/8 network for corporate traffic and sales employees use the 10.10.0.0/16 block across all offices. You want to track how much traffic sales employees generate in relation to all other users. To do this you need to define two *groups*:

- The first one will be called "Sales" and have a subnet of 10.10.0.0/16
- The second will be called "Rest" and have a subnet of 10.0.0.0/8

Since the script will only match a client to one *group*, data for clients belonging to sales employees will only appear in the first *group*.

To define groups as a command line parameter:

Use parameter `-g` to enter the group definitions as a single string. The string should have the following format:

```
<group name>=<subnet or vlan>,<subnet or vlan>;<group name>=<subnet or vlan>
```

You define enter a name for the group, followed by the equals sign "=", followed by *subnet* or *vlan* definitions separated by commas. Separate groups by using a semicolon. If your group definition contains spaces, enclose the whole string in quotes or double quotes, depending on your operating system.

Subnet or VLAN definitions can have the following form:

- `sub:<subnet>`, for example `sub:10.0.0.0/8`
- `vid:<vlan id>`, for example `vid:10`
- `vname:<vlan name>`, for example `vname:corp`

Read on for a complete example.

To define *groups* as an *init config file* section:

- Enter the section label `[GROUPS]`

- Enter a *groupname* record: `groupname=<name>`
- Enter the subnets, VLAN IDs and VLAN names needed to define the group. The records to do this are: `subnet=<ipv4 subnet>`, `vlanid=<VLAN ID>` and `vlanname=<VLAN name>`

Example: You want to compare two groups, one for corporate traffic and one for contractor traffic. Corporate users include Sales (VLAN 10 with subnet 10.10.0.0/16 and VLAN name "sales") and Engineering (VLAN 20 with subnet 10.20.0.0/16 and VLAN name "engineering"). Contractors use VLAN 30 with subnet 10.30.0.0/16 and VLAN name "contractors".

Here is a way you could define these two groups as an *init config file* section:

```
[GROUPS]
groupname=Corporate
#This is for Sales users
subnet=10.10.0.0/16
#This is for engineering users
vlanid=20

groupname=Contractors
#This is for contractors
vlanname=contractors
```

Here is same configuration as a command line parameter:

```
-g "Corporate=sub:10.10.0.0/16,vid:20;Contractors=vname:contractors"
```

Using filters

You can use filters to select which organizations, networks and devices the script will pull data from. By default the script will pull data from all MX and Z appliances in all networks and all organizations the administrator running the script has access to. You can enter multiple filters by separating them with commas. The available filters are:

- `org:<organization name>`
- `net:<network name>`
- `tag:<network tag>`
- `dtag:<device tag>`
- `dtype:<device type>`

The valid options for filter `dtype` are:

- `dtype:mr`
- `dtype:ms`
- `dtype:mx`
- `dtype:all`

To have the script pull data from device types other than MX and Z appliances, you will need to explicitly override the default value of the `dtype` filter, which is `dtype:mx`.

Example: Set a filter to pull data from all devices with the tag "core-switch" in the organization "Meraki Inc":

```
-f "org:Meraki Inc,dtype:all,dtag:core-switch"
```

Issuing commands

You can use the argument `-c <command>` to activate the different functions of this script. The available commands are:

`-c sync`	Executes a database sync without producing any report output.
`-c report:<time>`	Executes a database sync and runs a report for the time range defined in <time>. The report can be sent as an email or displayed on screen.
`-c report-offline:<time>`	Runs a report for the time range defined in <time> without executing a database sync first. The report can be sent as an email or displayed on screen.
`-c dbdump`	Displays the raw contents of the database.
`-c dbreconfigure`	Overwrites the configuration stored in the database with a new one.

Syncing the database

It is important that the *database sync* operation is executed frequently, so that the contents of the local database are complete and as accurate as possible. It is recommended to set the sync operation to be executed as a scheduled task, at least once a week.

To trigger a *database sync*, run the script with the argument `-c sync`. Example:

```
python usagestats.py -k 1234 -d mydatabase -c sync
```

The command line argument `-c report:<time>` will also execute a database sync before running a report.

Running reports

Use the command line arguments `-c report:<time>` and `-c report-offline:<time>` to run reports on the database. By default, the report will be displayed on screen. You can also have the report sent by email, as described later in this section.

The difference between `-c report:<time>` and `-c report-offline:<time>` is that `report-offline` will not attempt to connect to the Meraki cloud to *sync* the database before running the report.

Parameter `<time>` has multiple valid forms:

`<integer>`	Run report for the last `<integer>` days.
`last-week`	Run report for the last complete week (Monday-Sunday).
`last-month`	Run report for the last complete month (1st-last of month).
`<start date> to <end date>`	Run report between two dates given in ISO format.

Examples:

```
python usagestats.py -k 1234 -d mydatabase -c report:10
python usagestats.py -k 1234 -d mydatabase -c report:last-week
python usagestats.py -k 1234 -d mydatabase -c report:last-month
python usagestats.py -k 1234 -d mydatabase -c "report:2018-01-01 to 2018-02-10"
```

By default, the report will be displayed on screen. To have it sent via email instead, you will need to provide the following additional command line arguments:

`-u <user>`	Email account (username) that will be used to send email reports. This parameter is optional, but needed to send emails.
`-p <pass>`	Password for the email account that will be used to send email reports. This parameter is optional, but needed to send emails.
`-r <recipient>`	Recipient email address to send email reports to. This parameter is optional, but needed to send emails.

The following argument is also required for sending emails, unless you are using a Gmail account:

`-s <server>`	Email server to use for emailing reports. If omitted, Gmail will be used.

An example of running a report and sending it via a Gmail account:

```
python usagestats.py -k 1234 -d mydatabase -c report:last-month -u
          merakireports@gmail.com -p Password123 -r merakireports@gmail.com
```

Other functions

dbdump: This function will display the current contents of a database file in raw form, without executing a sync first. It is intended as a tool for testing, troubleshooting and debugging the script. An example of running dbdump:

```
python usagestats.py -k 1234 -d mydatabase -c dbdump
```

dbreconfigure: This function will drop the configuration stored in a database file and attempt to read a new one via command line parameters or an *init config file*. The function will not affect existing client usage data stored in the database. Be careful when using this function, as improper use may cause contents of the database to become meaningless. An example of running dbreconfigure:

```
python usagestats.py -k 1234 -d mydatabase -i new_configuration.txt -c dbreconfigure
```

Troubleshooting

Some situations which may result in the script failing to execute, display warnings, or produce inaccurate results include:

- When defining groups, for every subnet you will need to give either its IPv4 subnet, or its VLAN ID, or its VLAN name. If you enter multiple of these for the same subnet, the script will throw a warning about not being able to resolve some of the parameters. This is not an error and the script can continue normally.

- The Meraki Dashboard API has a maximum request rate of 5 requests/second. To optimize performance, The script tries to reach as close to this maximum rate as possible, without exceeding it. If you are consistently getting error messages from the script, writing *"Unable to contact Meraki cloud"*, you may be exceeding the maximum request rate. To solve this issue, try adjusting the global variable `API_EXEC_DELAY` at the beginning of the script to a higher value. The default value is 0.21 (seconds).
- The script includes database version checking. If you create a database with one version of the script and later download a newer version of the script that needs a different database format, you may get an error message with the text *"Database version not compatible. Please start a new database"*. You can check if the database format has changed between two versions of the script by comparing the global variable `DB_VERSION`.
- If reports seem to be inaccurate with usage values being too low, the *sync* operation may not be executed often enough. *Sync* needs to be executed at least once per 29 days, preferably at least once a week as a scheduled task.