Project 1 – Parsing a log file

Often, system administrators need to analyze log files to check for unusual activities, such as a spikes in traffic during a DDOS attack. Unfortunately, there are many different kinds of devices/programs that produce different kinds of log files in many different formats, often with a variable of number of fields, so there is no one tool that understands all the different formats. Consequently, sysadmins often need to write scripts to parse (break into its individual fields) log files for analysis. In this assignment you will write a script to parse a log file produced by the **tcpdump** utility. Once you have the required individual fields, your program will then perform some basic analysis tasks and output a file with a summary report of traffic for different choices of source and destination IPs.

General Instructions

The script you're creating should be able to take a varying number of calling arguments as shown below:

>>./project1.py file_name [opt] source_IP [opt] dest_IP

The **file_name** argument, which stands for the name of the file you are trying to parse is mandatory. However, both **source_IP** and **dest_IP** are optional. If only one IP is provided, your script should assume it is a **source_IP**. Your script should also be able to be called with an option **-s**, which tells your script that the parsed results should be sorted according to the size of the transmitted packets. This option can be placed in any order during the script call. For example, all the calls below should be considered valid:

>>./project1.py tcpdump_file -s 192.168.255.255

>>./project1.py tcpdump_file 192.0.0.1 -s 192.168.255.255

>>./project1.py -s tcpdump_file

Specific Instructions

For this project, you will need to create 5 (five) different functions. In what follows, you can find a the names you should give to these functions, as well as a description of what each function should do:

a) **check_ips()**: this function checks if the provided IPs are valid IPs. A valid IP consists of a string containing four numbers between **0** and **255** separated by dots. For instance, **192.165.255.255**, **192.0.0.0**, and **192.128.7.0** are valid IPs, whereas, **192.16.0** and **192.165.17.260** are not. **(1.5 marks)**

b) **check_options()**: This function checks if the user has entered the option **-s** (sort) or not, as one of the arguments of the script call. Note that this option can be in any position during the

function call. For example, **./project1.py -s file_name source_ip dest_ip**, **./project1.py file_name source_ip -s dest_ip**, and **./project1.py file_name source_ip dest_ip -s**, are all valid calls. When this option is present, you should set a global variable called **sort** to **True**. If this option is not present, you should set **sort** to **False**. **(1.5 marks)**

c) **parse_file()**: This function should parse the file indicated by the first argument of the script call. It should create a bi-dimensional list in which each row contains the following information about the TCP or UDP connections: **[source_IP, dest_IP, total_packet_size]**. Note that whenever a pair of **source_IP** and **dest_IP** appears for the second, third, etc. times, you should add the current packet size to the **total_packet_size** for this connection. Lines from the original file that contain non-TCP or non-UDP connections should be ignored. **(2.5 marks)**

OBS: Note that inside the **tcpdump_file,** all IPs are written together with port numbers. You need to create and auxiliar function to remove these port numbers before saving the IPs in your list.

d) **sort_list()**: Create a function that sorts the rows of the list created in part **c)** in descending order according to their **total_packet_size**. This function should be called by your script when the option **-s** is present.

In case you haven't finished part **c)**, you can use the example list provided below. **(1.0 marks)**

example_list = [['192.0.80.1','208.0.0.1',34], ['192.0.80.1','200.0.255.255',224], ['192.24.8.1','108.0.8.8',304], ['192.0.25.1','228.0.38.1',128]]

e) **print_list()**: This function should print the sorted (or not) list in a file called **result.txt**. Make sure that **result.txt** showcases the required information in a visually appealing way, such as in the provided examples. This function will return different results depending on the arguments passed while calling the script: **(1.0 marks)**

a) In case no **source_IP** and no **dest_IP** were called, print the whole list (sorted or not depending on **-s**) of **source_IPs**, **dest_IPs**, and **total_packet_sizes**

b) In case the source_IP was provided, print only the rows of the list containing this **source_IP**. (sorted or not depending on **-s**)

c) In case the both source_IP and dest_IP were called, display on the terminal the value of **total_packet_sizes** assocaited with this pair of Ips. There is no need to create a file **result.txt** in this scenario.

f) Code quality: 2.5 marks for this project will be assigned based on the quality of your code, based on the guidelines provided at the end of this document.

Examples

The files example1.txt, example2.txt, and example3.txt were created using the respective script calls below. Your script should output the exact same information when called with the same list of arguments.

./project1.py tcpdump_file -s > example1.txt

./project1.py tcpdump_file 192.168.0.15 > example2.txt

./project1.py tcpdump_file -s 192.168.0.15 66.185.85.146 > example3.txt

Guidelines

- Make sure to include a comment at the start of your program identifying yourself, the course, the assignment, etc.
- Put a comment above each function to identify what the parameters represent, what the return value (if any) represents, and a one line statement of what the function does.
- Put comments within functions, whenever you are doing something that would not be self-evident to someone reading your program. Don't put a comment on every line - too many are as bad as not enough.
- Put blank lines above and below functions to separate them from each other.
- Don't put in extra blank lines. (Some people put a blank lines between every line of code!)
- Read and understand the specifications. If you do not understand the specifications, ask me for clarification. If you do not implement something required, you will lose marks, even if you didn't understand the requirement - i.e. it is your job to seek clarification.
- Do not change the specifications. If you print something out and the assignment does not tell you to print it out, you are changing the specification and will lose marks.
- Format your output to look, as much as possible, like the sample shown in the specification. The closer it looks, the better your mark will be.
- Get rid of unnecessary (and confusing) duplication. For code, you can do this by factoring out common code and putting it into a function. In a regex, you can always delete {1} because it simply means the character in front of it repeats exactly once, but they always do by default! I also noticed several people were including parentheses in regexes, but they served no purpose.

For example, '([0-9]{1})' is the same as '[0-9]'. Why make it look more complicated than need be?

- Indent the same number of spaces. Always.
- Use three spaces for each indentation group.
- Develop your code anywhere you like, but make sure your code runs under Ubuntu and looks nice in gedit. That's what I use to check it.
- If you develop your code in Windows, don't submit it without testing as above. Transferring source files from Windows to Linux requires transferring them in ASCII mode - and dragging and dropping often transfers files in binary mode. If transferred in binary mode, you will get extra characters in your file and the Python interpreter won't understand, and your program will crash.
- Do not leave external resources "open" when your program terminates (i.e. close all file objects).
- Do not open and close a resource every time you want to write something to it if you are doing so in a tight loop. Open it before the loop, and close it after the loop.
- Make sure your code is efficient. There are multiple ways to accomplish the same result. However, some ways are clearly inefficient. For example, iterating over a list twice or three times, when the task could have been done iterating over it only once is clearly an example of inefficiency.
- Use meaningful names for your variables. For example if you have a variable that stores a set of users, it is better to name it **user_set** instead of **var37**.
- Do not create variables to hold values from the outputs of functions if these values are only used once. For example, given that the output of a function called **func1** needs to be used as an input argument to a function called **func2**, you shoud write your code as:

  func2(func1())

  instead of

  unnecessary_var = func1()
  func2(unnecessary_var)