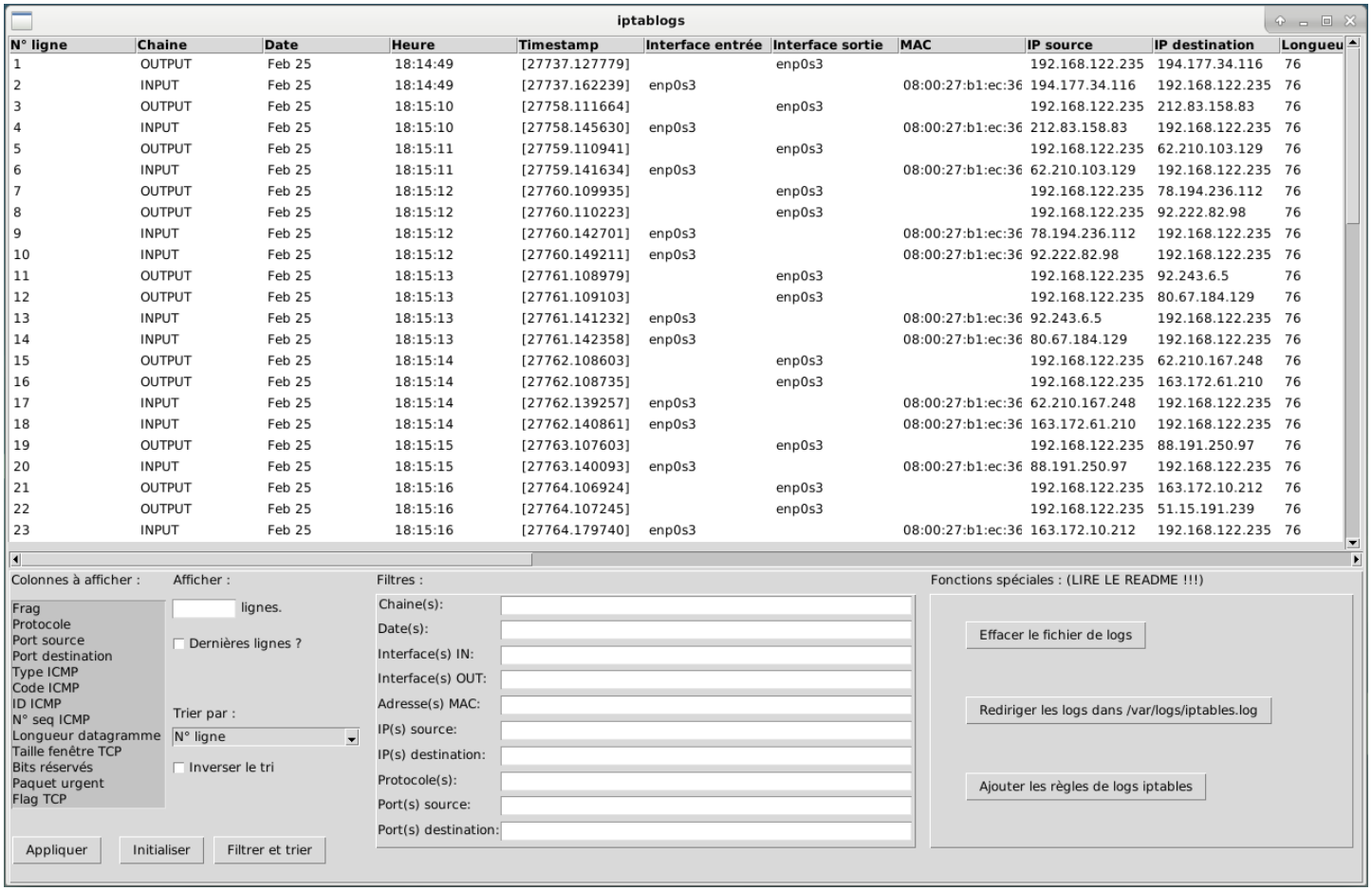
External documentation for iptablogs :

Table des matières

Introduction :

This file presents an in-depth look at the script/program iptablogs. It is essentially the iptablogs.py file cut in chunks with more detailed informations on how it works and, I hope, a clearer presentation. The code presented in this support file is not meant to be used as is. If you want to copy bits of the code (or the whole), do it directly from iptablogs.py.

As stated in the readme.md file, this script/program is written is Python 3.7 and needs to be run with root privileges. If you use another version of Python, it might result in displaying issues (observed on Python 3.5).



Shebang:

Line 1 is the shebang. It specifies where the interpreter is located (you'll need to change that if you use another version of python :

```
#!/usr/bin/env python3
```

Encoding:

Line 2 is the encoding (UTF-8) :

```
# -*-coding:Utf-8 -*
```

Modules :

Line 4 to 8 are the modules used :

```
from tkinter import *
from tkinter import ttk
from re import search
from os import getuid
from subprocess import getoutput
```

The GUI relies on *tkinter* and *ttk* modules.
The function *search* from the module *re* is used to split the raw lines into usable data.
The function *getuid* from the module *os* is used to get the uid of the user (in order to know if it's run as root).
The function *getoutput* from the module *subprocess* is used to launch external commands (to create the config file in /etc/rsyslog.d/ and to add iptables rules), it returns the standard output.

Global variable :

dico_colonnes is a dictionnary used multiple times as a base to make the link between the variable names and the corresponding formated column names of the table. It's easier to iterate the dictionary rather than writing all the variable names each time we want to call them. See for instance :

```
dico_colonnes = {
    "numero_ligne": "N° ligne",
    "chaine": "Chaine",
    "date": "Date",
    "heure": "Heure",
    "timestamp": "Timestamp",
    "interface_in": "Interface entrée",
    "interface_out": "Interface sortie",
    "adresse_mac": "MAC",
    "ip_source": "IP source",
    "ip_destination": "IP destination",
    "longueur_trame": "Longueur trame",
    "type_service": "TOS",
    "priorite_tos": "PREC",
    "time_to_live": "TTL",
    "id_paquet": "ID",
    "flag_fragment": "Frag",
    "protocole": "Protocole",
    "port_source": "Port source",
    "port_destination": "Port destination",
    "icmp_type": "Type ICMP",
    "icmp_code": "Code ICMP",
    "icmp_id": "ID ICMP",
    "icmp_n_sequence": "N° seq ICMP",
    "longueur_datagramme": "Longueur datagramme",
    "fenetre_tcp": "Taille fenêtre TCP",
    "bits_reserves": "Bits réservés",
    "paquet_urgent": "Paquet urgent",
    "tcp_flag": "Flag TCP"
}
```

Checking if the program is run as root :

The program needs to be run with root privileges as it won't be able to read the log file otherwise (and perform optional actions like adding rules to iptables). The function *verif_sudo* checks if it is run as root. It then instantiate the object *interface* of the *Interface* class and calls the method *mainloop* from *interface.fenetre* object (see Interface Class for more details).

```
def verif_sudo():
    """This function checks if the program is run with root privileges"""
    if getuid() != 0:
        print("Ce programme nécessite d'être exécuté avec les droits root.")
    else:
        interface = Interface()
        interface.fenetre.mainloop()
```

If the program is not run as root, it just prints a warning message in the standard output. This is the first function called.

Error windows :

The program calls the function *appeler_fenetre_erreur* each time an error is raised.

It instantiates an object of the class *Toplevel* and displays the error message along with a description.

```
def appeler_fenetre_erreur(err, msg="Erreur inconnue"):
    """This function is called when an error is raised.
    It creates a simple error window which takes as arguments the raised error and an optional
message.
    """
    fenetre_erreur = Toplevel()
    fenetre_erreur.title("Erreur")
    label_erreur = Label(fenetre_erreur, text="{}\n{}".format(msg, err))
    label_erreur.pack()
```

Stdout windows :

The following function is similar to the one above but is only used to display potential messages from the standard output (only used with the "special functions").

```
def appeler_fenetre_stdout(msg_stdout):
    """This function is called when we want to display messages returned by the standard output.
    It creates a simple window which takes as argument the standard output.
    """
    fenetre_stdout = Toplevel()
    fenetre_stdout.title("stdout")
    label_stdout = Label(fenetre_stdout, text=msg_stdout)
    label_stdout.pack()
```

Interface class :

Except for the above functions and variable, all attributes and methods are defined in the class *Interface*. It starts with a try and ends with and except to catch every unexpected error that might occur.

```
class Interface(object):
    """This class contains almost every attribute and method used in this program."""
    # We want to catch every unpredicted error.
    try:
```

[...]

```
    except Exception as err_generale:
    appeler_fenetre_erreur(err_generale)
```

Interface __init__ method :

This method initializes the interface and some of the attributes that will be used later.

```
def __init__(self):
    # The list liste_lignes_log_initiale will hold the lines read from the log file. Because it's
going to be used in a method of this class and also because we don't want to use a global variable,
it's registered here as an attribute of the Interface class object :
    self.liste_lignes_log_initiale = []

    # We also declare the dico_filtre attribute.
    self.dico_filtre = None
```

The interface is built using the *tk* module. You can adjust the window resolution by changing the value in *fenetre.geometry("1280x800")*. Note : the position and size of most of the elements are defined by absolute values, so, changing that might need a bit of tweaking.

```
    # Building the window :
    self.fenetre = Tk()
    self.fenetre.title("iptablogs")
    self.fenetre.geometry("1280x800")
```

Then we create a *Treeview* object from the ttk module, it will be used to display the values of the log lines in a cleaner way. The "browse" selectmode means the elements can't be edited by clicking them. Defining the Treeview columns can be done using a list (*dico_colonnes.keys()*) :

```
    # Building the table as a child of fenetre (the browse mode means read only) :
    self.tableau = ttk.Treeview(self.fenetre, height=23, selectmode="browse")

    # Hiding the first column (it's a treeview control, but we don't really use it as is) :
    self.tableau["show"] = "headings"

    # Defining columns we want to display (we use the keys from dico_colonnes):
    self.tableau["columns"] = dico_colonnes.keys()

    # Displaying the columns names (we browse dico_colonnes and display the formated names
corresponding to each key). We use i as index :
    i = 0
    for cle_colonne_tab, valeur_colonne_tab in dico_colonnes.items():
        self.tableau.column(i, width=120, minwidth=40, stretch=0)
        self.tableau.heading(i, text=valeur_colonne_tab, anchor="w")
        i += 1

    # Scrollbars to adjust the table view :
    self.defil_hori = ttk.Scrollbar(self.fenetre, orient="horizontal", command=self.tableau.xview)
    self.defil_hori.place(x=1, y=485, height=20, width=1278)
    self.defil_verti = ttk.Scrollbar(self.fenetre, orient="vertical", command=self.tableau.yview)
    self.defil_verti.place(x=1260, y=0, height=485, width=20)
    self.tableau.configure(xscrollcommand=self.defil_hori.set)
    self.tableau.configure(yscrollcommand=self.defil_verti.set)
    self.tableau.pack(side=TOP, fill=X)
```

The second part of the interface is defined here. It uses a *Frame* object type from the *ttk* module. Its main purpose here is to allow to place the elements using relative coordinates instead of absolute ones :

```
    # Second part of the interface :
    # Frame containing the sorting/filtering options :
    self.trame_options = ttk.Frame(self.fenetre, borderwidth=1, relief="groove")
    self.trame_options.place(height=295, width=1278, x=1, y=505)
```

To select the columns the user wants to display, I use a *Listbox* type object from *tk*. The selectmode "extended" allows to select several elements in the list. This list is fed using *dico_colonnes.values()* because we want to display the formated column names.

```
    # Options to select the columns to display :
    self.label_colonnes_a_afficher = ttk.Label(self.trame_options, text="Colonnes à afficher :")
    self.label_colonnes_a_afficher.place(relx=0.0001, rely=0.001)
    self.liste_colonnes_a_afficher = Listbox(self.trame_options, width=18, height=13,
selectmode="extended")
    self.liste_colonnes_a_afficher.place(relx=0.0001, rely=0.09)
    # We fill the listbox with the values from dico_colonnes :
    for valeur_colonne_a_afficher in dico_colonnes.values():
        self.liste_colonnes_a_afficher.insert("end", valeur_colonne_a_afficher)
```

The button *bouton_filtre_colonnes_a_afficher* triggers the method *filtrer_colonnes()*.

```
    # This button triggers the method filrer_colonnes to narrow down the columns :
    self.bouton_filtre_colonnes_a_afficher = Button(self.trame_options, text="Appliquer",
                                                    command=self.filtrer_colonnes)
    self.bouton_filtre_colonnes_a_afficher.place(relx=0.0008, rely=0.85)
```

The number of lines the user want to display can be typed in the *Entry* type object *inpbox_nb_lignes_a_afficher* from *tk* :

```
    # Options to limit the number of lines to be displayed :
    self.label_nb_lignes_a_afficher = ttk.Label(self.trame_options, text="Afficher :")
    self.label_nb_lignes_a_afficher.place(relx=0.12, rely=0.001)
    self.label_nb_lignes_a_afficher2 = ttk.Label(self.trame_options, text="lignes.")
    self.label_nb_lignes_a_afficher2.place(relx=0.17, rely=0.09)
    self.inpbox_nb_lignes_a_afficher = ttk.Entry(self.trame_options, width=7)
    self.inpbox_nb_lignes_a_afficher.place(relx=0.12, rely=0.09)
```

If the user wants to display the X first lines or the X last lines, it can be done by checking the *Checkbutton* type object from *ttk* named *checkbox_dernieres_nb_lignes_a_afficher*.

```
    # checkbox_dernier will get the value from "checkbox_dernieres_nb_lignes_a_afficher" as an
integer.
    self.checkbox_dernier = IntVar()
    self.checkbox_dernieres_nb_lignes_a_afficher = ttk.Checkbutton(self.trame_options,
                                                                   text="Dernières lignes ?",
                                                                   variable=self.checkbox_dernier)
    self.checkbox_dernieres_nb_lignes_a_afficher.place(relx=0.12, rely=0.2)
```

Next are the sorting options. I used once again a checkbox named *checkbox_inverser_tri*, and also a *Combobox* type object from *ttk* named *combobox_tri*. It is fed using *dico_colonnes.values()*, because once again, we want to display the formated names (you can notice that I removed the option to sort by date because it would require a dedicated method and quite a bit of resources and it would be the same as filtering by line anyway) :

```
    # Sorting options :
    self.label_tri = ttk.Label(self.trame_options, text="Trier par :")
    self.label_tri.place(relx=0.12, rely=0.43)
    # combobox_tri_valeur_selectionnee will get the selected value from the combobox "combobox_tri"
to know which column the user wants to sort by :
    self.combobox_tri_valeur_selectionnee = StringVar()
    # liste_attributs_tri contains every value from dico_colonnes except Date :
    self.liste_attributs_tri = [x for x in list(dico_colonnes.values()) if x != "Date"]
    self.combobox_tri = ttk.Combobox(self.trame_options, values=self.liste_attributs_tri,
state="readonly",
                                     textvariable=self.combobox_tri_valeur_selectionnee, width=20)
    # We set the combobox_tri default selected item as the line number :
    self.combobox_tri.current(0)
    self.combobox_tri.place(relx=0.12, rely=0.5)
    # checkbox_var_tri will get the value from checkbox_inverser_tri to know if the user wants to
```

```
reverse the
    # sorting :
    self.checkbox_var_tri = IntVar()
    self.checkbox_inverser_tri = ttk.Checkbutton(self.trame_options, text="Inverser le tri",
                                                 variable=self.checkbox_var_tri)
    self.checkbox_inverser_tri.place(relx=0.12, rely=0.6)
```

Then we have the button *bouton_tri* which triggers the method remplir_tableau which is defined here. This button launches the filtering and the sorting.

```
    self.bouton_tri = Button(self.trame_options, text="Filtrer et trier",
command=self.remplir_tableau)
    self.bouton_tri.place(relx=0.15, rely=0.85)
```

The next button *bouton_initialiser* is the most important one, because it triggers the method *initialiser()* which reads the log file the first time. It can be used serveral times if the use wants to refresh the content :

```
    # Button to launch the first read of the log and display the list :
    self.bouton_initialiser = Button(self.trame_options, text="Initialiser",
command=self.initialiser)
    self.bouton_initialiser.place(relx=0.08, rely=0.85)
```

The following lines are essentially *Label* and *Entry* type objects. They are used to set the desired filters. Note that they are defined as children of the *Frame* type object named *trame_filtres*, this is important because the filtering method will use the *winfo_children()* method to browse the *Entry* objects with a loop instead of calling them individually with the *get()* method to extract the user's inputs.

```
    # Options to filter the list (note : the input boxes names are the same as the dico_colonnes
keys):
    self.label_filtres = ttk.Label(self.trame_options, text="Filtres :")
    self.label_filtres.place(relx=0.27, rely=0)
    self.trame_filtres = ttk.Frame(self.trame_options, borderwidth=1, relief="groove")
    self.trame_filtres.place(height=240, width=510, relx=0.27, rely=0.07)
    self.label_filtre_chaine = ttk.Label(self.trame_filtres, text="Chaine(s):")
    self.label_filtre_chaine.place(relx=0.001, rely=0.005)
    self.inpbox_filtre_chaine = ttk.Entry(self.trame_filtres, width=48, name="chaine")
    self.inpbox_filtre_chaine.place(relx=0.23, rely=0.005)
    self.label_filtre_date = ttk.Label(self.trame_filtres, text="Date(s):")
    self.label_filtre_date.place(relx=0, rely=0.1)
    self.inpbox_filtre_date = ttk.Entry(self.trame_filtres, width=48, name="date")
    self.inpbox_filtre_date.place(relx=0.23, rely=0.1)
    self.label_filtre_interface_in = ttk.Label(self.trame_filtres, text="Interface(s) IN:")
    self.label_filtre_interface_in.place(relx=0, rely=0.2)
    self.inpbox_filtre_interface_in = ttk.Entry(self.trame_filtres, width=48, name="interface_in")
    self.inpbox_filtre_interface_in.place(relx=0.23, rely=0.2)
    self.label_filtre_interface_out = ttk.Label(self.trame_filtres, text="Interface(s) OUT:")
    self.label_filtre_interface_out.place(relx=0, rely=0.3)
    self.inpbox_filtre_interface_out = ttk.Entry(self.trame_filtres, width=48,
name="interface_out")
    self.inpbox_filtre_interface_out.place(relx=0.23, rely=0.3)
    self.label_filtre_adresse_mac = ttk.Label(self.trame_filtres, text="Adresse(s) MAC:")
    self.label_filtre_adresse_mac.place(relx=0, rely=0.4)
    self.inpbox_filtre_adresse_mac = ttk.Entry(self.trame_filtres, width=48, name="adresse_mac")
    self.inpbox_filtre_adresse_mac.place(relx=0.23, rely=0.4)
    self.label_filtre_ip_source = ttk.Label(self.trame_filtres, text="IP(s) source:")
    self.label_filtre_ip_source.place(relx=0, rely=0.5)
    self.inpbox_filtre_ip_source = ttk.Entry(self.trame_filtres, width=48, name="ip_source")
    self.inpbox_filtre_ip_source.place(relx=0.23, rely=0.5)
    self.label_filtre_ip_destination = ttk.Label(self.trame_filtres, text="IP(s) destination:")
    self.label_filtre_ip_destination.place(relx=0, rely=0.6)
    self.inpbox_filtre_ip_destination = ttk.Entry(self.trame_filtres, width=48,
name="ip_destination")
```

```
    self.inpbox_filtre_ip_destination.place(relx=0.23, rely=0.6)
    self.label_filtre_protocole = ttk.Label(self.trame_filtres, text="Protocole(s):")
    self.label_filtre_protocole.place(relx=0, rely=0.7)
    self.inpbox_filtre_protocole = ttk.Entry(self.trame_filtres, width=48, name="protocole")
    self.inpbox_filtre_protocole.place(relx=0.23, rely=0.7)
    self.label_filtre_port_source = ttk.Label(self.trame_filtres, text="Port(s) source:")
    self.label_filtre_port_source.place(relx=0, rely=0.8)
    self.inpbox_filtre_port_source = ttk.Entry(self.trame_filtres, width=48, name="port_source")
    self.inpbox_filtre_port_source.place(relx=0.23, rely=0.8)
    self.label_filtre_port_destination = ttk.Label(self.trame_filtres, text="Port(s) destination:")
    self.label_filtre_port_destination.place(relx=0, rely=0.9)
    self.inpbox_filtre_port_destination = ttk.Entry(self.trame_filtres, width=48,
name="port_destination")
    self.inpbox_filtre_port_destination.place(relx=0.23, rely=0.9)
```

In the last frame of the interface are the "special" functions. They are designed to reduce the actions the user has to do outside of the interface as much as possible. The first button calls the method *effacer_fichier_log*, it removes the log file */var/log/iptables.log* and restarts the *rsyslog* service (if we only remove the file, it won't be rebuilt) and it can be useful to reduce the time needed to read the log file if you have a very large one.
The second button calls the method *rediriger_les_logs_iptables*, it redirects the logs to the file /var/log/iptables.log.
The third button calls the method ajouter_regles_log_iptables, it creates rules in iptables to log the drops (or the opposite if the default rule is ACCEPT).

```
    # Controls of the "special" functions :
    self.label_fonctions_speciales = ttk.Label(self.trame_options,
                                               text="Fonctions spéciales : (LIRE LE README !!!)")
    self.label_fonctions_speciales.place(relx=0.68, rely=0)
    self.trame_fonctions_speciales = ttk.Frame(self.trame_options, borderwidth=1, relief="groove")
    self.trame_fonctions_speciales.place(height=240, width=400, relx=0.68, rely=0.07)
    # This button triggers the method effacer_fichier_log :
    self.bouton_effacer_log = Button(self.trame_fonctions_speciales, text="Effacer le fichier de
logs",
                                     command=self.effacer_fichier_log)
    self.bouton_effacer_log.place(relx=0.08, rely=0.1)
    # This button triggers the method rediriger_les_logs_iptables :
    self.bouton_rediriger_logs = Button(self.trame_fonctions_speciales,
                                        text="Rediriger les logs dans /var/logs/iptables.log",
                                        command=self.rediriger_les_logs_iptables)
    self.bouton_rediriger_logs.place(relx=0.08, rely=0.4)
    # This button triggers the method ajouter_regles_log_iptables :
    self.bouton_ajouter_regles_iptables = Button(self.trame_fonctions_speciales,
                                                 text="Ajouter les règles de logs iptables",
                                                 command=self.ajouter_regles_log_iptables)
    self.bouton_ajouter_regles_iptables.place(relx=0.08, rely=0.7)
```

Lignelog class :

This class is used to store the data cut out of each line in separate attributes (each object of this class represents a line of the log file). To instantiate an object of this class, we need the raw line to cut (*ligne_a_decouper* as string), and the line number (*n_ligne* as integer). To search for each attribute, I use the *search* function (from *re* module) to look for a specific regular expression. Once it finds a match, I use the *groups()* function to return the value. You can notice that some of the values are converted to integers, it makes more sense to sort them in numerical order instead of alphabetical.

```
class Lignelog:
    """Classe representing a line in the logs."""

    def __init__(self, ligne_a_decouper, n_ligne):
        """Initialization of an object Lignelog.
        It takes as arguments the line itself (str) and the line number (int).
        It makes use of regular expressions to extract the data we're looking for from each string.
```

```
        If a regex fails it will return either an attribute error (if it's a NoneType) or an
IndexError (if it can't return the group).
        As you'll notice, the attribute names are the same as the keys in dico_colonnes.
    """
        self.numero_ligne = n_ligne
        try:
            self.chaine = search("(?<= \\[netfilter-)(OUTPUT|INPUT|FORWARD)(?=\\] )",
                                 ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.chaine = ""
        try:
            self.date = search("^([A-Z][a-z]+\\s((\\s\\d)|(\\d{2})))(?= )",
ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.date = ""
        try:
            self.heure = search("(?<= )(\\d{2}:\\d{2}:\\d{2})(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.heure = ""
        try:
            self.timestamp = search("(?<= )(\\[\\s*\\d*\\.\\d*\\])(?= )",
ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.timestamp = ""
        try:
            self.interface_in = search("(?<= IN=)(\\S*)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.interface_in = ""
        try:
            self.interface_out = search("(?<= OUT=)(\\S*)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.interface_out = ""
        try:
            self.adresse_mac = search("(?<= MAC=)((([a-z]|[0-9]){0,2}:){5,13}([a-z]|[0-9]){0,2})(?=
)",
                                      ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.adresse_mac = ""
        try:
            self.ip_source = search("(?<= SRC=)((([0-9]{0,3}.){4}([0-9]{0,3}))?)(?= )",
                                    ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.ip_source = ""
        try:
            self.ip_destination = search("(?<= DST=)((([0-9]{0,3}.){4}([0-9]{0,3}))?)(?= )",
                                         ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.ip_destination = ""
        try:
            self.longueur_trame = int(search("(?<= LEN=)(\\d*)(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.longueur_trame = ""
        try:
            self.type_service = search("(?<= TOS=)(\\S*)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.type_service = ""
        try:
            self.priorite_tos = search("(?<= PREC=)(\\S*)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.priorite_tos = ""
        try:
            self.time_to_live = int(search("(?<= TTL=)(\\d*)(?= )", ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.time_to_live = ""
        try:
            self.id_paquet = int(search("(?<= ID=)(\\d*)(?= )", ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
```

```
            self.id_paquet = ""
        try:
            self.flag_fragment = search("(?<= )(DF|CE|MF)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.flag_fragment = ""
        try:
            self.protocole = search("(?<= PROTO=)(\\S*)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.protocole = ""
        try:
            self.port_source = int(search("(?<= SPT=)(\\d*)(?= )", ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.port_source = int()
        try:
            self.port_destination = int(search("(?<= DPT=)(\\d*)(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.port_destination = int()
        try:
            self.icmp_type = int(search("(?<= TYPE=)(\\d{0,2})(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.icmp_type = int()
        try:
            self.icmp_code = int(search("(?<= CODE=)(\\d{0,2})(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.icmp_code = int()
        try:
            self.icmp_id = int(search("(?: PROTO=ICMP.*)(?<= ID=)(\\d*)(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.icmp_id = int()
        try:
            self.icmp_n_sequence = int(search("(?<= SEQ=)(\\d*)(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.icmp_n_sequence = int()
        try:
            self.longueur_datagramme = int(search("(?: PROTO=UDP.*)(?<= LEN=)(\\d*)(?= )",
                                                  ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.longueur_datagramme = int()
        try:
            self.fenetre_tcp = int(search("(?<= WINDOW=)(\\d*)(?= )",
ligne_a_decouper).groups()[0])
        except (AttributeError, IndexError):
            self.fenetre_tcp = int()
        try:
            self.bits_reserves = search("(?<= RES=)(\\S*)(?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.bits_reserves = ""
        try:
            self.paquet_urgent = search("(?<= URGP=)[0|1](?= )", ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.paquet_urgent = ""
        try:
            self.tcp_flag = search("(?<= )(((ACK).?|(PSH).?|(FIN).?|(RST).?|(SYN).?)+?)(?= )",
                                   ligne_a_decouper).groups()[0]
        except (AttributeError, IndexError):
            self.tcp_flag = ""
```

Lignelimitee class :

The objects from this class have the same attribute names as *Lignelog* objects but they aren't created the same way and their values aren't strings or integers but lists. This class is used later on to compare each line to the filters the user specified.

```
class Lignelimitee:
    """The objects from this class have similar attributes to those of Lignelog.
    It is used to create similar objects as Lignelog but the attributes values are lists based on
the filters
    the user specified. We can't use the class Lignelog because its __init__ would search for
regular
    expressions again in a string. Instead we browse the dictionary dico_colonnes to create an
attribute name,
    the values are provided by the liste_arguments variable.
    """

    def __init__(self, liste_arguments):
        index_dico_attributs = 0
        for cle_dico_attribut in dico_colonnes.keys():
            setattr(self, cle_dico_attribut, liste_arguments[index_dico_attributs])
            index_dico_attributs += 1
```

init_dico_filtres() method :

This method browse the *trame_filtres* frame children looking for data to filter by. It returns a dictionary whose keys names are from dico_colonnes.key() and whose values are lists of values entered in *Entry* fields. The lists are built using the split method with the argument ", ", this means each value must be separated by a comma followed by a space.

```
def init_dico_filtres(self):
    """This method initializes the variable dico_filtre_tmp.
    It's a dictionary whose keys correspond to the table columns, that's why it uses
dico_colonnes.keys(). To each key corresponds a value (list) that will be used to filter the lines
that will be displayed.
    """
    # Once again, we rely on dico_colonnes. We create a dictionary with the same keys as
dico_colonnes, and for
    # each key an empty list :
    dico_filtre_tmp = {}
    for index_dico in dico_colonnes.keys():
        dico_filtre_tmp[index_dico] = []
    # Then we browse the frame containing the filters input boxes :
    for index_trame in self.trame_filtres.winfo_children():
        valeurs = None
        # We check if the object is a ttk.Entry class object and if it's not empty :
        if isinstance(index_trame, ttk.Entry) and len(index_trame.get()) > 0:
            # We store its name in nom_attribut :
            nom_attribut = index_trame.winfo_name()
            # If the input box names are "port_source" or "port_destination", we convert them into
integers
            # before adding them to the list "valeurs".
            if nom_attribut == ("port_source" or "port_destination"):
                try:
                    valeurs = [int(nombre) for nombre in ((index_trame.get()).split(", "))]
                except ValueError as erreur_valeur:
                    # If we try to convert an alpha character into an integer, we call the function
                    # appeler_fenetre_erreur to warn the user that the entry is incorrect.
                    appeler_fenetre_erreur(erreur_valeur,
                                           "Le port source ou destination doivent être des valeurs
numériques")
            else:
                # Same as above, except we don't convert into integer this time.
                valeurs = (index_trame.get()).split(", ")
            if valeurs is not None:
                # If valeurs isn't empty, we add its content to the value corresponding to the
correct key in
                # dico_filtre_tmp (as a reminder, each key is associated with a list, that's why we
use
                # extend().
                dico_filtre_tmp[nom_attribut].extend(valeurs)
```

```
    # We then return the dictionary dico_filtre_tmp
    return dico_filtre_tmp
```

initialiser() method :

This method first calls the method *lire_fichier_log()* to feed the attribute *liste_lignes_log_initiale* (a list made of Lignelog objects). This method must be called at least once because all sorting/filtering will be made based on it. Lastly, it call the method *remplir_tableau()* to fill the table named *tableau* (to display the values).

```
def initialiser(self):
    """This method reads the log file then fills the table.
    It calls the methods lire_fichier_log() to fill the list "liste_lignes_log_initiale", that one
will be used to sort and filter the results.
    It then calls remplir_tableau() to fill the table.
    initialiser can be used several times to read again the log file but it's only mandatory once
to be able to sort and filter the results.
    """
    self.liste_lignes_log_initiale = self.lire_fichier_log()
    self.remplir_tableau()
```

filtrer_colonnes() method :

This short method only redefines the columns the user wants to display through the option *displaycolumns* of the Treeview widget. It gets the selected items using the *curselection()* method of the object *liste_colonnes_a_afficher*.

```
def filtrer_colonnes(self):
    """The purpose of this method is to narrow down the columns that are displayed.
    It gets the column names selected in the listbox "list_colonnes_a_afficher".
    """
    self.tableau["displaycolumns"] = self.liste_colonnes_a_afficher.curselection()
```

lire_fichier_log() method :

This method reads the log file /var/log/iptables.log. For each line, it creates an object *Lignelog* with the raw line and the line number as arguments. It adds the line returned by the previous method to the list *liste_ligne* and returns it.

```
def lire_fichier_log(self):
    """This method reads the log file and returns a list of objects Lignelog."""
    liste_lignes = []
    try:
        with open("/var/log/iptables.log", "r") as fichier:
            lignes = fichier.readlines()
            num_ligne = 0
            for ligne in lignes:
                num_ligne += 1
                ligne_coupee = self.Lignelog(ligne, num_ligne)
                liste_lignes.append(ligne_coupee)
    # If the file is not found, we warn the use and advise him to read the readme.
    except FileNotFoundError as erreur:
        appeler_fenetre_erreur(erreur, "Fichier log introuvable, veuillez lire le README.")
    # It then returns the list of objects Lignelog :
    return liste_lignes
```

remplir_tableau() method :

This method clears the table, then calls *trier_resultats(lignes_du_log)* to sort the values from *liste_lignes_log_initiale*. Then, it fills the table line by line.

```
def remplir_tableau(self):
    """This method clears the treeview object "tableau", then inserts the new fresh lines in it.
    It calls the method trier_resultats to gather the list of lines to insert.
    """
    # First we check if the list of log lines is not empty :
    if len(self.liste_lignes_log_initiale) > 0:
        # We delete the contents of the table
        self.tableau.delete(*self.tableau.get_children())
        # We call the function trier_resultats. It returns a sorted and filtered list :
        liste_triee_a_inserer_dans_tableau = self.trier_resultats(self.liste_lignes_log_initiale)
        # For each entry in the list, we add its values in succession :
        for ligne_liste_triee in liste_triee_a_inserer_dans_tableau:
            self.tableau.insert("", "end", values=(list(ligne_liste_triee.__dict__.values())))
    else:
        # If the list is empty, we advise the user to click on "Initialiser".
        appeler_fenetre_erreur("", "Rien à trier, veuillez d'abord cliquer sur 'Initialiser'.")
```

trier_resultats(lignes_du_log) method :

This method calls the methods *init_dico_filtres()*, *filtrer_liste(liste_a_filtrer, liste_filtres)*, *limiter_resultats(liste_filtree)*, *tri_liste(liste_a_trier, colonne_tri, inverse)*, and returns a filtered and sorted list of lines.

```
def trier_resultats(self, lignes_du_log):
    """This method is invoked by the method remplir_tableau().
    Its purpose is mainly to trigger other functions to sort and filter results.
    """
    # cle is the column that we want to sort our table by. Given that combobox_tri displays
formated names, we
    # use a condition to get the key name corresponding to the formated name) :
    cle = "".join(x for x, y in dico_colonnes.items() if y == self.combobox_tri.get())
    # v_inverser_tri is a boolean, its value is True if the user checked the box to reverse the
sorting :
    v_inverser_tri = self.checkbox_var_tri.get()
    # We call init_dico_filtres to return a dictionary with the filters the user created :
    self.dico_filtre = self.init_dico_filtres()
    # We call the method filtrer_liste with as arguments the initial list of lines from the log
file and the
    # filters the user specified in the form of a dictionary. It then return a filtered list :
    liste_filtree_tri = self.filtrer_liste(lignes_du_log, self.dico_filtre)
    # Next, we call the method limiter_resultats to narrow down the number of lines we want to
display :
    liste_limitee_tri = self.limiter_resultats(liste_filtree_tri)
    # Finally, we call the method tri_list to sort the lines and return liste_triee that will be
used to fill
    # the table :
    liste_triee = self.tri_liste(liste_limitee_tri, cle, v_inverser_tri)
    return liste_triee
```

filtrer_liste(liste_a_filtrer, liste_filtres) method :

It is a static method (it doesn't use any attribute of the class as you can see from the absence of *self* in its definition).

```
@staticmethod
def filtrer_liste(liste_a_filtrer, liste_filtres):
    """This method's purpose is to filter the results.
    It takes as arguments the list of lines to filter and the list of filters.
    It then compares each line "attribute" to the filters' ones. If it finds a match, the line is
added to the
    list liste_filtree_generee that is returned.
    """
    # The list is created empty :
    liste_filtree_generee = []
    # We take each line in liste_a_filtrer :
    for ligne_a_filtrer in liste_a_filtrer:
        present = True
        for cle_dico in dico_colonnes.keys():
            # For each key in dico_colonnes, we get the corresponding value in liste_filtres :
            attribut_filtre = liste_filtres[cle_dico]
            # Then we check if the value of the attribute of the line is in the list
attribut_filtre :
            attribut_ligne = getattr(ligne_a_filtrer, cle_dico)
            if len(attribut_filtre) > 0 and attribut_ligne not in attribut_filtre:
                # If the value is not empty but doesn't match, the boolean "present" becomes False.
                present = False
        # If none of the matching tests set the "present" boolean to False, then the line is added
to
        # liste_filtree_generee :
        if present is True:
            liste_filtree_generee.append(ligne_a_filtrer)
    # We then return the filtered list :
    return liste_filtree_generee
```

limiter_resultats(liste_filtree) method :

This method returns a list of lines limited by the number provided by the user, from the beginning or from the end depending on the user's preference (reflected by the value of the checkbox *checkbox_dernier*). The number of lines is specified in the *Entry* object *inpbox_nb_lignes_a_afficher*.

```
def limiter_resultats(self, liste_filtree):
    """This method limits the number of lines to display in the table.
    It takes as argument the list of lines to narrow down and returns a list limited by the
specified number.
    """
    # We fetch the number of elements to display. If the input box is empty or if the number is
bigger than the
    # number of elements in the list (the number of lines in the log file,
nombre_elements_dans_liste), then we
    # keep all of them. This number is stored in the variable nombre_elements_a_afficher.
    nombre_elements_dans_liste = len(liste_filtree)
    liste_limitee = []
    try:
        entree_nb_elements_a_afficher = self.inpbox_nb_lignes_a_afficher.get()
        if len(entree_nb_elements_a_afficher) > 0 and int(entree_nb_elements_a_afficher) <= \
                nombre_elements_dans_liste:
            nombre_elements_a_afficher = int(entree_nb_elements_a_afficher)
        else:
            nombre_elements_a_afficher = nombre_elements_dans_liste
    except ValueError as err_nombre_lignes:
        # If the user has entered an invalid character, we display all of the lines but warn the
user :
        appeler_fenetre_erreur(err_nombre_lignes, "Vous avez saisi un nombre lignes incorrect !")
        nombre_elements_a_afficher = nombre_elements_dans_liste
    nombre_elements_affiches = 0
```

```
    # We check the status of the checkbox "checkbox_dernier". If it's checked, we create a list of
X lasts lines
    # of the log file, else we create a list of X firsts ones (where X is the number of lines to
display).
    if self.checkbox_dernier.get():
        nombre_elements_affiches = nombre_elements_dans_liste - nombre_elements_a_afficher
        while nombre_elements_affiches < nombre_elements_dans_liste:
            dico_attrib = vars(liste_filtree[nombre_elements_affiches])
            liste_limitee.insert(nombre_elements_affiches,
self.Lignelimitee(list(dico_attrib.values())))
            nombre_elements_affiches += 1
    else:
        while nombre_elements_affiches < nombre_elements_a_afficher:
            dico_attrib = vars(liste_filtree[nombre_elements_affiches])
            liste_limitee.insert(nombre_elements_affiches,
self.Lignelimitee(list(dico_attrib.values())))
            nombre_elements_affiches += 1
    return liste_limitee
```

tri_liste(liste_a_trier, colonne_tri, inverse) method :

This static method simply uses a lambda function to sort by the user defined column and order.

```
@staticmethod
def tri_liste(liste_a_trier, colonne_tri, inverse):
    """This method sorts the provided list.
    It takes as arguments the list of lines to sort, the column to sort by (str), and
    a boolean to reverse the sorting if needed.
    """
    liste_a_trier.sort(key=lambda ligne_tri: getattr(ligne_tri, colonne_tri), reverse=inverse)
    return liste_a_trier
```

Special functions :

ajouter_regles_log_iptables() method :

This static method adds iptables rules to log with the needed prefixes. It repeats theses two actions for each default chain (INPUT, OUTPUT, FORWARD) : First it checks if the log rule has already been added or not. If not, it adds the log rule to the corresponding chain. If any message is returned in the standard output, the message is passed as argument to the function *appeler_fenetre_stdout to display it in a popup,* else we call *appeler_fenetre_stdout* just to tell the user that the function worked properly.

```
@staticmethod
def ajouter_regles_log_iptables():
    """This method adds iptables rules to log with the needed prefixes.
    If one of the commands returns an error, it is displayed using appeler_fenetre_stdout method.
    """
    sortie_std = ""
    # We check if the log rule exist in each chain. If not, we add the log rule to the chain.
    for chaine in ["INPUT", "OUTPUT", "FORWARD"]:
        chaine_a_executer = "iptables -nL {0} --line-numbers | grep \"\\[netfilter-
{0}\\]\"".format(chaine)
        if not getoutput(chaine_a_executer):
            sortie_std += getoutput("iptables -A {0} -j LOG"
                                    " --log-prefix=\"[netfilter-{0}] \" ".format(chaine))
    # We instantiate a popup to display the result :
    if sortie_std:
        appeler_fenetre_stdout(sortie_std)
    else:
        appeler_fenetre_stdout("Règles ajoutées avec succès")
```

rediriger_les_logs_iptables() method :

This static method creates the conf file in /etc/rsyslog.d/ to redirect the log messages containing the log prefix specified in the iptables rules (starting with [netfilter-). It then restarts the rsyslog service (or else it wouldn't create the log file). If we catch a message in the standard output, we display it using the *appeler_fenetre_stdout* method or we just display a success message using the same method.

```
@staticmethod
def rediriger_les_logs_iptables():
    """This method allows the logs to be redirected to var/log/iptables.log
    It creates a file in /etc/rsyslog.d/ to filter the kernel logs and redirects every line
containing "[netfilter-"
    """
    sortie_std = getoutput(
        "echo ':msg,contains,\"[netfilter-\" /var/log/iptables.log' > /etc/rsyslog.d/1-
iptables.conf")
    sortie_std += getoutput("service rsyslog restart")
    if sortie_std != "":
        appeler_fenetre_stdout(sortie_std)
    else:
        appeler_fenetre_stdout("Commande exécutée avec succès")
```

effacer_fichier_log() method :

This static method simply sends *rm* and *restart* shell commands to remove the log file and restart the rsyslog service (or else the file wouldn't be recreated). This method can be useful when the log file reaches a big size, slowing down the scripts. If we catch a message in the standard output, we display it using the *appeler_fenetre_stdout* method or we just display a success message using the same method.

```
@staticmethod
def effacer_fichier_log():
    """This method simply removes the log file and restarts the rsyslog service."""
    sortie_std = getoutput("rm /var/log/iptables.log")
    sortie_std += getoutput("service rsyslog restart")
    if sortie_std != "":
        appeler_fenetre_stdout(sortie_std)
    else:
        appeler_fenetre_stdout("Commande exécutée avec succès")
```

Starting the script :

At the very end is a call to the function *verif_sudo()* which is the first function called at the start of the script :

```
verif_sudo()
```