Project 2 - Data Science applied to Cybersecurity

Raphaël CANIN, Melisa KOCKAN
E4 AIC

Table of Contents :

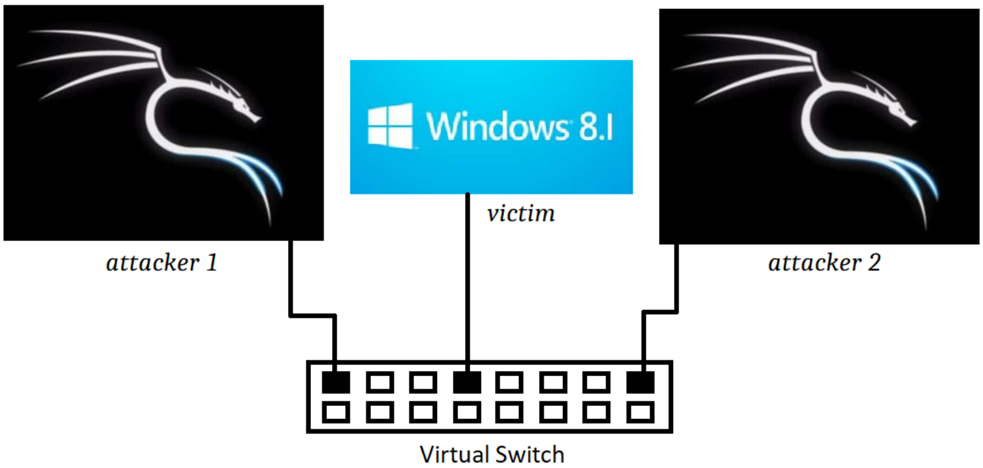
Introduction

The purpose of this lab is to practice Data Science through the scope of Cybersecurity.
In order to do so, we will take the instance of a DDos attack performed by SYN Flooding.
This DDos attack, as known as Distributed Denial of Service attack, consists in sending several SYN in a short amount of time in order to turn a server unavailable.

There are 4 tasks including 3 which consist in setting up the environment, performing the environment and tracking the traffic using ; while the other is focused on using a deep learning model to prevent an attack.

Task 1: Setting up the environment

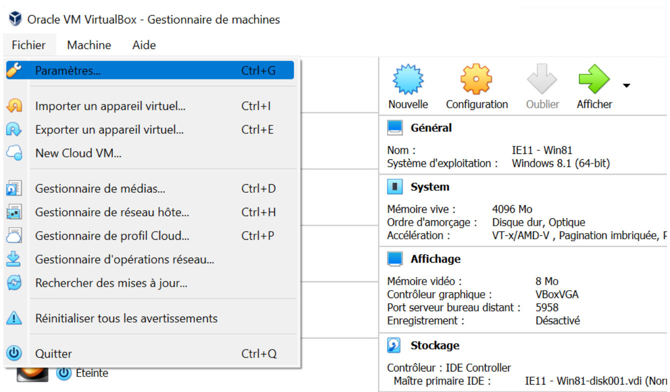
The environment for this lab is described by the figure below :



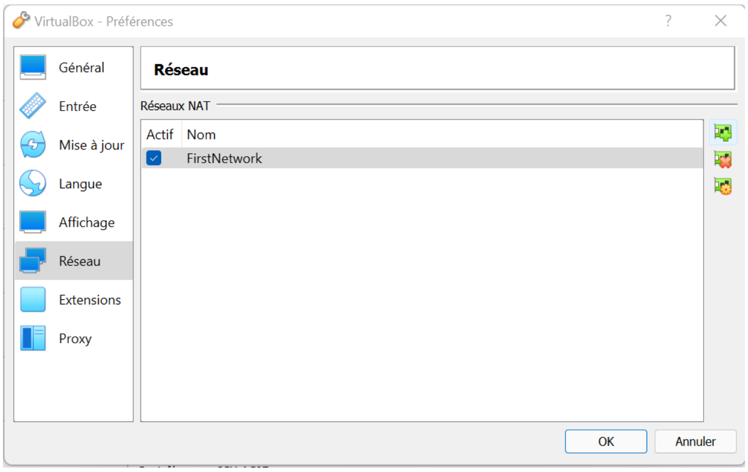
We will be using 2 kali linux machines as attackers, and a Windows 8.1 virtual machine as a victim.

1) Setting up the environment

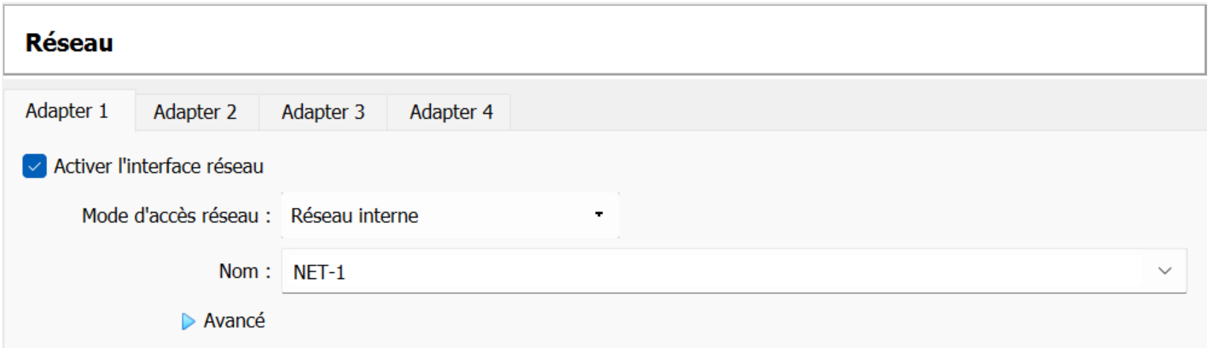
The first step is to create a NAT Network on Virtual Box ; to do so we go to file/settings :



Then we go onto Network, and select the green icon on the right corner “Add a Nat Network”:

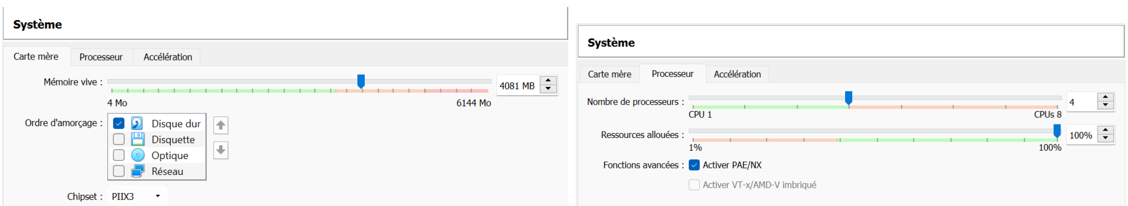


Once the NAT network has been created, we have to select it for the configuration of each of our 3 virtual machines :

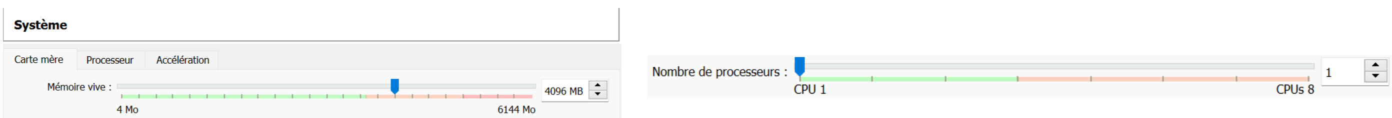


2) Customizing parameters to support simultaneously 3 VM

In order to run simultaneously our 3 Virtual Machines on our own laptop, we had to change some parameters in Virtual box's "System" section. The parameters RAM and the number of processors granted to each VM had to be increased. The RAM and number of processors attributed to each attacker was willingly way greater than the one attributed to the victim.



parameters set for the attackers machine



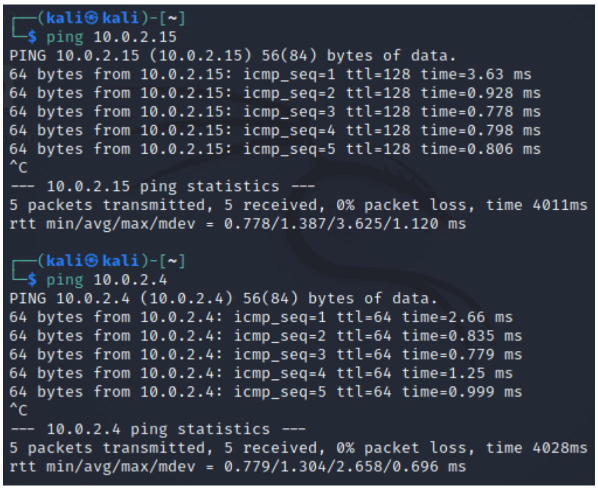
parameters set for the victim's machine

3) Testing connectivity

That being done, using the command ifconfig on kali and ipconfig on linux enabled to know the IP address of each virtual machine. The IP address of attacker 1, attacker 2 and the victim respectively corresponds to : 10.0.2.5 10.0.2.5 and 10.0.2.15.

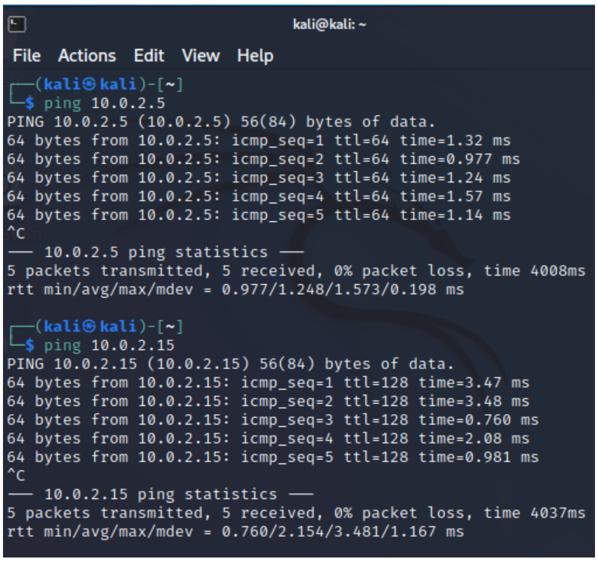
Having the IP address of each virtual machine, the command ping enabled to test their connectivity. However, the Windows victim wasn't reachable when we first tried, we had to disable the Windows Firewall to make it work. Disabling the Windows Firewall was a very simple manipulation done by going into the windows setting ; we had to restart the machine for the modifications to be effective.

Afterward, the connection between each of the machines was correctly established.



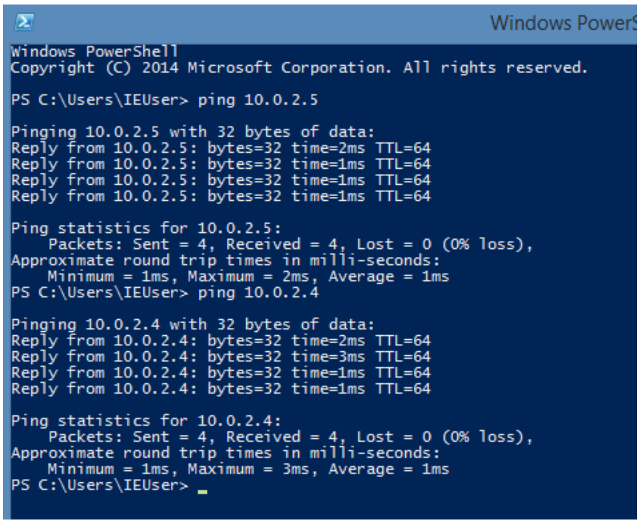
```
┌──(kali㉿kali)-[~]
└─$ ping 10.0.2.15
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=128 time=3.63 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=128 time=0.928 ms
64 bytes from 10.0.2.15: icmp_seq=3 ttl=128 time=0.778 ms
64 bytes from 10.0.2.15: icmp_seq=4 ttl=128 time=0.798 ms
64 bytes from 10.0.2.15: icmp_seq=5 ttl=128 time=0.806 ms
^C
--- 10.0.2.15 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4011ms
rtt min/avg/max/mdev = 0.778/1.387/3.625/1.120 ms

┌──(kali㉿kali)-[~]
└─$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=2.66 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=0.835 ms
64 bytes from 10.0.2.4: icmp_seq=3 ttl=64 time=0.779 ms
64 bytes from 10.0.2.4: icmp_seq=4 ttl=64 time=1.25 ms
64 bytes from 10.0.2.4: icmp_seq=5 ttl=64 time=0.999 ms
^C
--- 10.0.2.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4028ms
rtt min/avg/max/mdev = 0.779/1.304/2.658/0.696 ms
```

Testing connectivity from attacker 1



```
kali@kali: ~
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~]
└─$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_seq=1 ttl=64 time=1.32 ms
64 bytes from 10.0.2.5: icmp_seq=2 ttl=64 time=0.977 ms
64 bytes from 10.0.2.5: icmp_seq=3 ttl=64 time=1.24 ms
64 bytes from 10.0.2.5: icmp_seq=4 ttl=64 time=1.57 ms
64 bytes from 10.0.2.5: icmp_seq=5 ttl=64 time=1.14 ms
^C
— 10.0.2.5 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
rtt min/avg/max/mdev = 0.977/1.248/1.573/0.198 ms

┌──(kali㉿kali)-[~]
└─$ ping 10.0.2.15
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=128 time=3.47 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=128 time=3.48 ms
64 bytes from 10.0.2.15: icmp_seq=3 ttl=128 time=0.760 ms
64 bytes from 10.0.2.15: icmp_seq=4 ttl=128 time=2.08 ms
64 bytes from 10.0.2.15: icmp_seq=5 ttl=128 time=0.981 ms
^C
— 10.0.2.15 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4037ms
rtt min/avg/max/mdev = 0.760/2.154/3.481/1.167 ms
```

Testing connectivity from attacker 2



```
Windows PowerS
Windows PowerShell
Copyright (C) 2014 Microsoft Corporation. All rights reserved.

PS C:\Users\IEUser> ping 10.0.2.5

Pinging 10.0.2.5 with 32 bytes of data:
Reply from 10.0.2.5: bytes=32 time=2ms TTL=64
Reply from 10.0.2.5: bytes=32 time=1ms TTL=64
Reply from 10.0.2.5: bytes=32 time=1ms TTL=64
Reply from 10.0.2.5: bytes=32 time=1ms TTL=64

Ping statistics for 10.0.2.5:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 2ms, Average = 1ms
PS C:\Users\IEUser> ping 10.0.2.4

Pinging 10.0.2.4 with 32 bytes of data:
Reply from 10.0.2.4: bytes=32 time=2ms TTL=64
Reply from 10.0.2.4: bytes=32 time=3ms TTL=64
Reply from 10.0.2.4: bytes=32 time=1ms TTL=64
Reply from 10.0.2.4: bytes=32 time=1ms TTL=64

Ping statistics for 10.0.2.4:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 3ms, Average = 1ms
PS C:\Users\IEUser> _
```

Testing connectivity from the victim

All the ping commands were successful, the environment has been correctly configured.

Task 2: Collecting and processing Data - Simulating DDoS Attacks on virtual machines

The software Metasploit will be used in order to perform the SYN flood attack. Fortunately, this software was already installed on both of the kali linux machines.

However, Wireshark wasn't installed on the Windows Virtual Machines. The installation has been done quickly thanks to https://www.wireshark.org/download.html.

1) Traffic during normal circumstances

Before attacking the victim with a Syn Flood attack, we launched a capture on Wireshark. The traffic obtained was the following one :

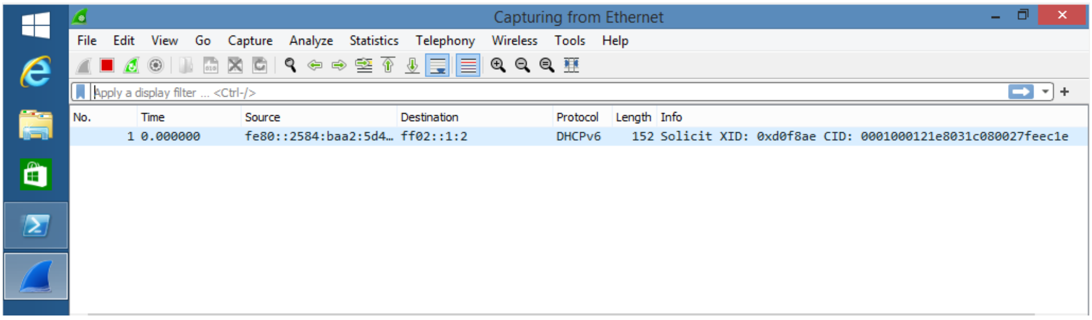
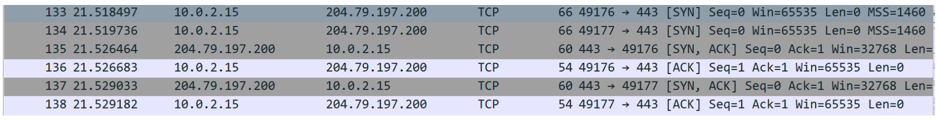


figure 1 - Traffic when nothing is happening

To get the traffic of what corresponds to "normal circumstances", we simply went on https://www.samsung.com/fr/ and tried to purchase the new Samsung Galaxy S22.



133	21.518497	10.0.2.15	204.79.197.200	TCP	66	49176 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460
134	21.519736	10.0.2.15	204.79.197.200	TCP	66	49177 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460
135	21.526464	204.79.197.200	10.0.2.15	TCP	60	443 → 49176 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=
136	21.526683	10.0.2.15	204.79.197.200	TCP	54	49176 → 443 [ACK] Seq=1 Ack=1 Win=65535 Len=0
137	21.529033	204.79.197.200	10.0.2.15	TCP	60	443 → 49177 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=
138	21.529182	10.0.2.15	204.79.197.200	TCP	54	49177 → 443 [ACK] Seq=1 Ack=1 Win=65535 Len=0

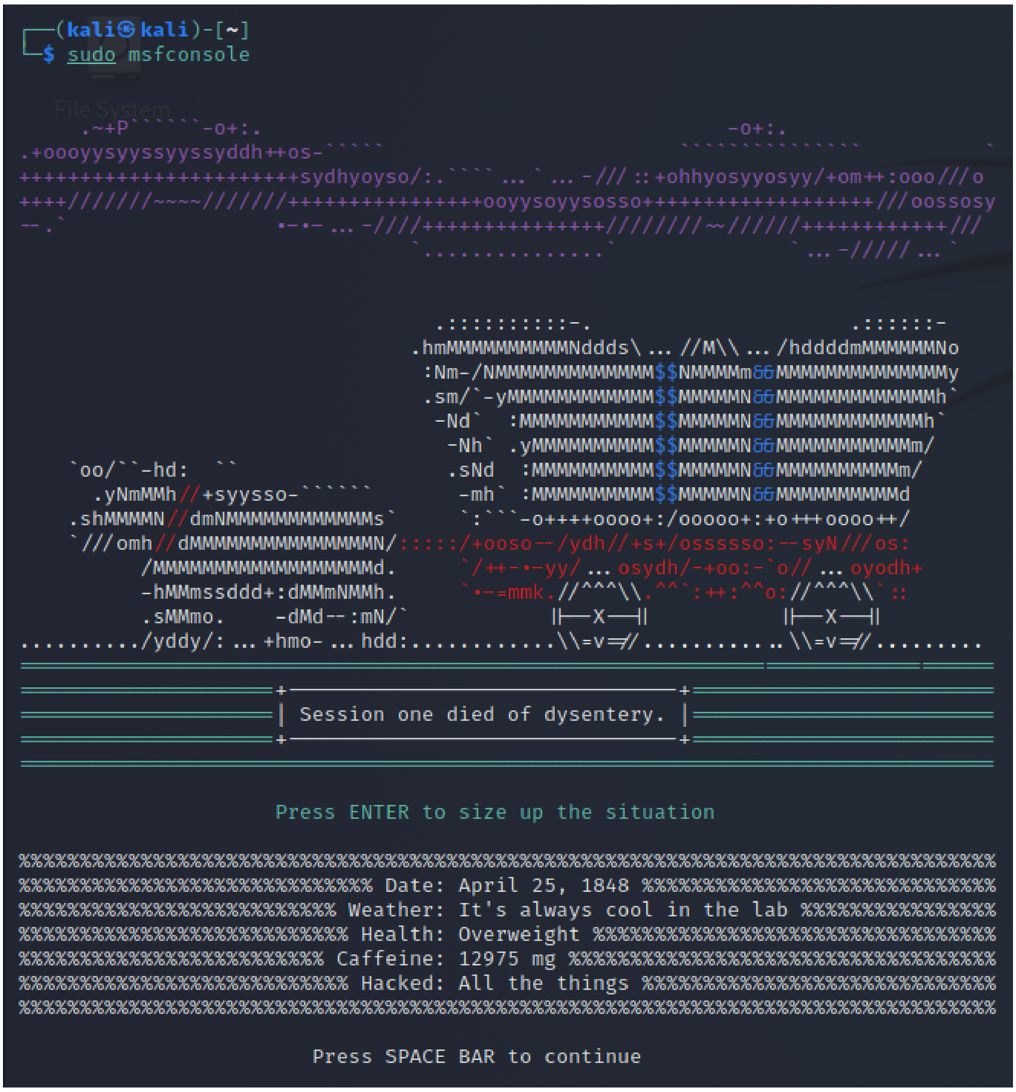
figure 2 - Traffic obtained in normal circumstances

The traffic obtained corresponds to a classical 3-way handshake with SYN, SYN-ACK and ACK.

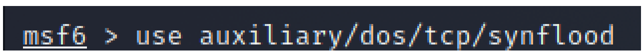
2) Traffic under a SYN Flood attack

During this part, attacker 1 will perform a SYN Flood attack onto the Victim.
To launch metasploit, we used the command **sudo msf console**.
It is important to insist on this command since we had the error “Auxiliary failed: RuntimeError doesn't have permission to capture on that device” at the beginning because we didn’t use the sudo prefix during our first try.

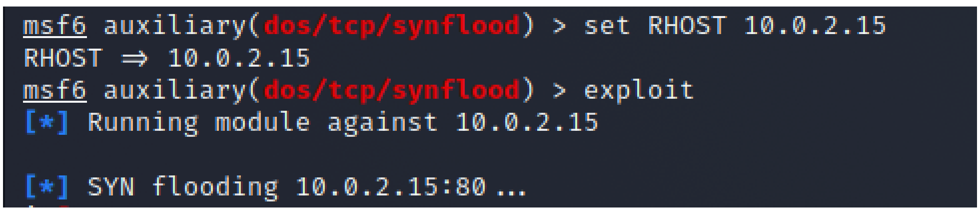
The metasploit console launches and prints an amusing drawing :



On the metasploit console, we specify that we want to use auxiliary/dos/tcp/synflood :



Once we entered this module, we define the host and the port we want to attack :



The Victim’s IP address is 10.0.2.15 and we decided to target the famous HTTP port which is the number 80.

The traffic we observed on Wireshark was the following one :

No.	Time	Source	Destination	Protocol	Length	Info
48960	265.540363	10.0.2.15	142.7.230.241	TCP	54	80 → 12741 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
48961	265.541306	142.7.230.241	10.0.2.15	TCP	60	33281 → 80 [SYN] Seq=0 Win=1365 Len=0
48962	265.541372	10.0.2.15	142.7.230.241	TCP	54	80 → 33281 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
48963	265.542600	142.7.230.241	10.0.2.15	TCP	60	6409 → 80 [SYN] Seq=0 Win=1845 Len=0
48964	265.542679	10.0.2.15	142.7.230.241	TCP	54	80 → 6409 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
48965	265.544210	142.7.230.241	10.0.2.15	TCP	60	[TCP Port numbers reused] 47538 → 80 [SYN] Seq=0 Win=39
48966	265.544276	10.0.2.15	142.7.230.241	TCP	54	80 → 47538 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
48967	265.545093	142.7.230.241	10.0.2.15	TCP	60	[TCP Port numbers reused] 58998 → 80 [SYN] Seq=0 Win=40
48968	265.545142	10.0.2.15	142.7.230.241	TCP	54	80 → 58998 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
48969	265.546041	142.7.230.241	10.0.2.15	TCP	60	23083 → 80 [SYN] Seq=0 Win=2031 Len=0

figure 3 - Traffic during the attack

The traffic observed corresponds to the definition of the SYN Flood attack : the attacker bombards the victim with SYN packets and it makes him unavailable to send ACK packets back. The main difference with the normal traffic that we have seen beforehand is that there is no ACK packets.

The capture of this attack has been saved into a csv file that we visualized using Python.

3) Analyzing the dataset

During this part, we will practice data visualization using the traffic captured under normal circumstances and during an attack :

```
normal.columns
```

```
Index(['No.', 'Time', 'Source', 'Destination', 'Protocol', 'Length', 'Info'], dtype='object')
```

These columns exactly corresponds to the one we had in Wireshark :

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info

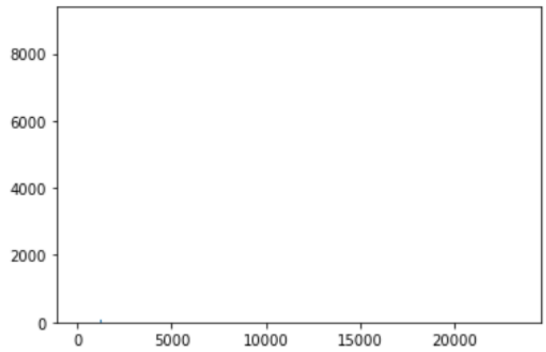
Here is a description of each column :
No - Number identifying the packets
Time - Time at which the packets has been received (in seconds)
Source - IP address of the source of the packet
Destination - IP address of the destination of the packet
Protocol - Protocol used for the packet (Categorical variable)
Length - Length of the packet at stake
Info - String providing information about the packet

The No column is unnecessary since packets are already ordered by their index in the DataFrame and that we are not looking for a history of which packets did what.

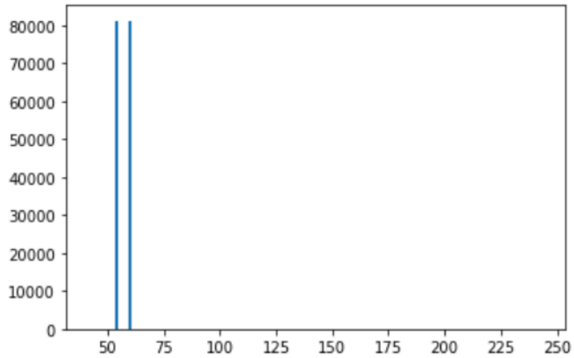
The Info column, although it was providing information about SYN / SYN ACK / ACK must be dropped because it displays way too much information that isn't exploitable in a dataset.

Let’s plot the 3 metrics Length, Protocol, and Source:

Length



Normal Circumstances



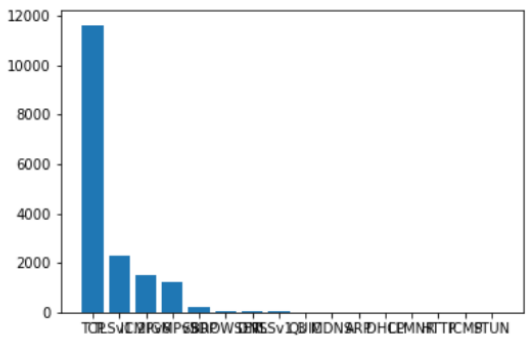
During a DDos

```
count     17021.00000
mean        977.33065
std         769.01787
min          42.00000
25%          75.00000
50%        1514.00000
75%        1514.00000
max       23414.00000
Name: Length, dtype: float64
```

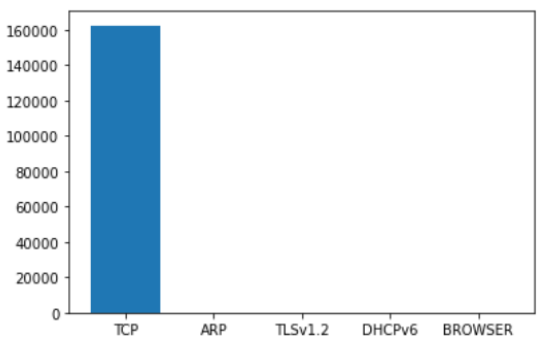
```
count    162501.000000
mean         57.004806
std           3.097377
min          42.000000
25%          54.000000
50%          60.000000
75%          60.000000
max         243.000000
Name: Length, dtype: float64
```

As expected, the length variable drastically changes according to if it is an attack or not. During a DDos, the majority of packets have a very small length compared to normal circumstances.

Protocol



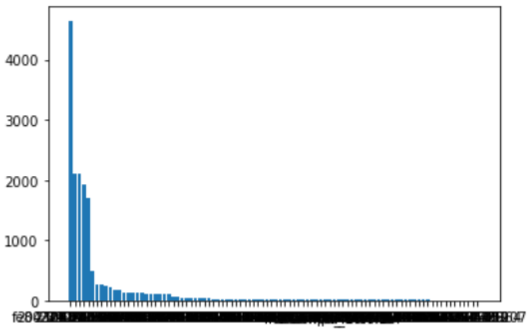
Normal Circumstances



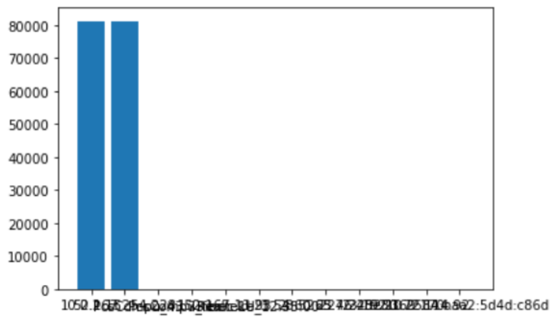
During a DDos

The protocol is also a key information to determine if we are facing an attack or not : during DDos, only a few Protocols are being used compared to under normal circumstances.

Source



Normal Circumstances



During a DDos

Finally, during a DDos attack, the majority of packets comes only from a few IP addresses, compared to under normal Circumstances.

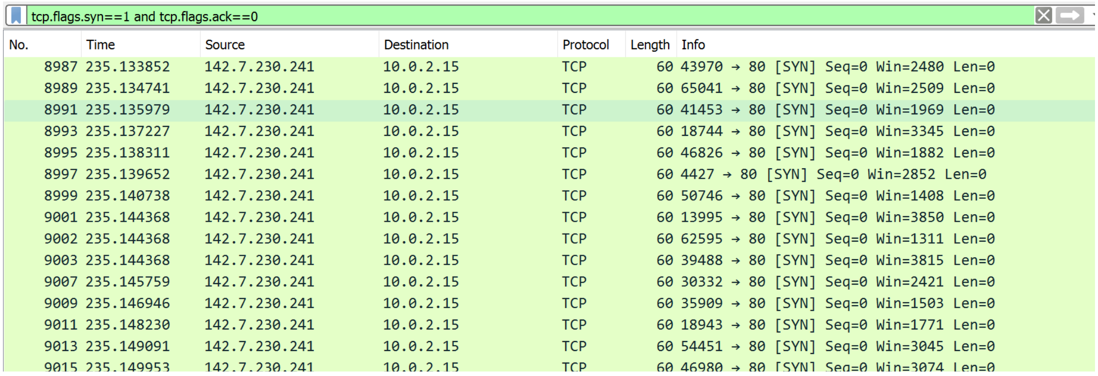
The statistics about Time and Destination have been given in the Jupyter notebook but won’t be described in this report since the assignment asks for 2 or 3 metrics.

Task 3:Detecting DDoS attacks with Wireshark

1) Performing a second attack

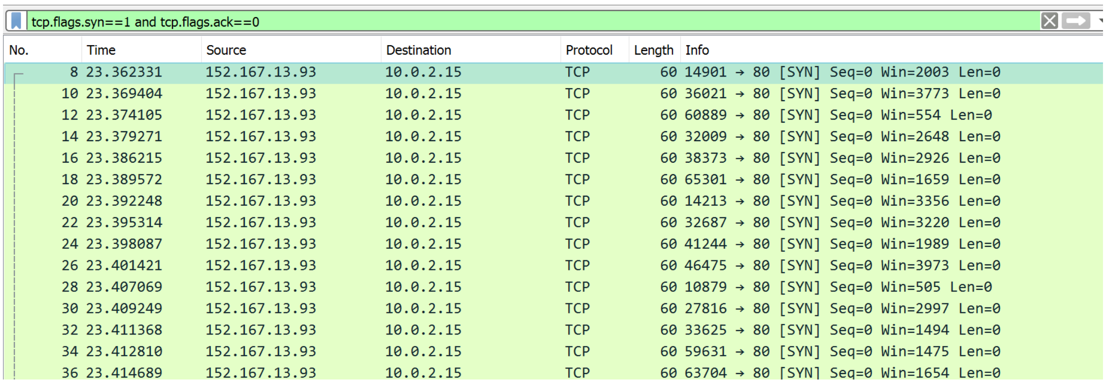
We performed the attack a second time using the same attacker 1 and saved the file as a secondattack.pcapng file.

In order to detect the IP address of the attacker, we selected the packets that were spammed by the attacker. In order to select such packets, we considered the fact that during a syn attack, the attacker sends SYN packets without letting the victim sending ACK packets back.
Hence, we used the following filter on Wireshark : tcp.flags.syn==1 and tcp.flags.ack==0



No.	Time	Source	Destination	Protocol	Length	Info
8987	235.133852	142.7.230.241	10.0.2.15	TCP	60	43970 → 80 [SYN] Seq=0 Win=2480 Len=0
8989	235.134741	142.7.230.241	10.0.2.15	TCP	60	65041 → 80 [SYN] Seq=0 Win=2509 Len=0
8991	235.135979	142.7.230.241	10.0.2.15	TCP	60	41453 → 80 [SYN] Seq=0 Win=1969 Len=0
8993	235.137227	142.7.230.241	10.0.2.15	TCP	60	18744 → 80 [SYN] Seq=0 Win=3345 Len=0
8995	235.138311	142.7.230.241	10.0.2.15	TCP	60	46826 → 80 [SYN] Seq=0 Win=1882 Len=0
8997	235.139652	142.7.230.241	10.0.2.15	TCP	60	4427 → 80 [SYN] Seq=0 Win=2852 Len=0
8999	235.140738	142.7.230.241	10.0.2.15	TCP	60	50746 → 80 [SYN] Seq=0 Win=1408 Len=0
9001	235.144368	142.7.230.241	10.0.2.15	TCP	60	13995 → 80 [SYN] Seq=0 Win=3850 Len=0
9002	235.144368	142.7.230.241	10.0.2.15	TCP	60	62595 → 80 [SYN] Seq=0 Win=1311 Len=0
9003	235.144368	142.7.230.241	10.0.2.15	TCP	60	39488 → 80 [SYN] Seq=0 Win=3815 Len=0
9007	235.145759	142.7.230.241	10.0.2.15	TCP	60	30332 → 80 [SYN] Seq=0 Win=2421 Len=0
9009	235.146946	142.7.230.241	10.0.2.15	TCP	60	35909 → 80 [SYN] Seq=0 Win=1503 Len=0
9011	235.148230	142.7.230.241	10.0.2.15	TCP	60	18943 → 80 [SYN] Seq=0 Win=1771 Len=0
9013	235.149091	142.7.230.241	10.0.2.15	TCP	60	54451 → 80 [SYN] Seq=0 Win=3045 Len=0
9015	235.149953	142.7.230.241	10.0.2.15	TCP	60	46980 → 80 [SYN] Seq=0 Win=3074 Len=0

ip address of attacker 1 during second attack



No.	Time	Source	Destination	Protocol	Length	Info
8	23.362331	152.167.13.93	10.0.2.15	TCP	60	14901 → 80 [SYN] Seq=0 Win=2003 Len=0
10	23.369404	152.167.13.93	10.0.2.15	TCP	60	36021 → 80 [SYN] Seq=0 Win=3773 Len=0
12	23.374105	152.167.13.93	10.0.2.15	TCP	60	60889 → 80 [SYN] Seq=0 Win=554 Len=0
14	23.379271	152.167.13.93	10.0.2.15	TCP	60	32009 → 80 [SYN] Seq=0 Win=2648 Len=0
16	23.386215	152.167.13.93	10.0.2.15	TCP	60	38373 → 80 [SYN] Seq=0 Win=2926 Len=0
18	23.389572	152.167.13.93	10.0.2.15	TCP	60	65301 → 80 [SYN] Seq=0 Win=1659 Len=0
20	23.392248	152.167.13.93	10.0.2.15	TCP	60	14213 → 80 [SYN] Seq=0 Win=3356 Len=0
22	23.395314	152.167.13.93	10.0.2.15	TCP	60	32687 → 80 [SYN] Seq=0 Win=3220 Len=0
24	23.398087	152.167.13.93	10.0.2.15	TCP	60	41244 → 80 [SYN] Seq=0 Win=1989 Len=0
26	23.401421	152.167.13.93	10.0.2.15	TCP	60	46475 → 80 [SYN] Seq=0 Win=3973 Len=0
28	23.407069	152.167.13.93	10.0.2.15	TCP	60	10879 → 80 [SYN] Seq=0 Win=505 Len=0
30	23.409249	152.167.13.93	10.0.2.15	TCP	60	27816 → 80 [SYN] Seq=0 Win=2997 Len=0
32	23.411368	152.167.13.93	10.0.2.15	TCP	60	33625 → 80 [SYN] Seq=0 Win=1494 Len=0
34	23.412810	152.167.13.93	10.0.2.15	TCP	60	59631 → 80 [SYN] Seq=0 Win=1475 Len=0
36	23.414689	152.167.13.93	10.0.2.15	TCP	60	63704 → 80 [SYN] Seq=0 Win=1654 Len=0

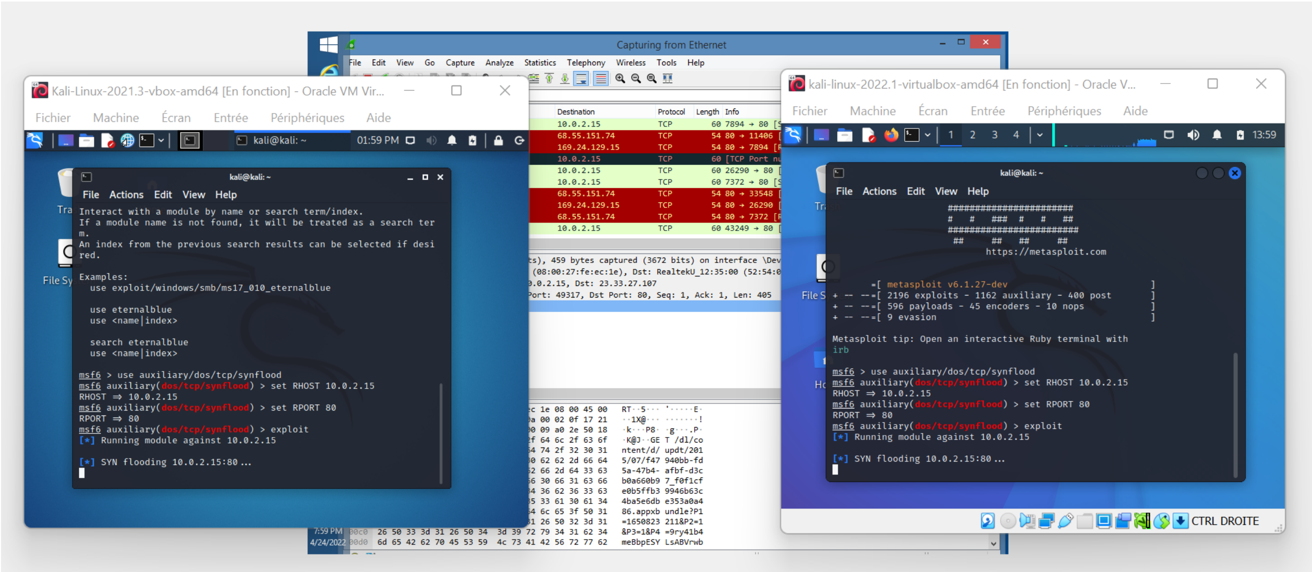
ip address of attacker 1 during first attack

The public IP address of the same attacker changed between the first and second attack because they have been done using different WIFI connections ; however, they correspond to the same attacker.

2) Performing simultaneous attacks

Finally, we performed a last attack combining simultaneously attacker 1 and attacker 2 SYN flood attacks.

We prepared each attack, then we used exploit to start both of them at the same time, which resulted to the following set up :



Using the same filter than above, we have been able to identify the 2 IP addresses of each attackers :

tcp.flags.syn==1 and tcp.flags.ack==0

No.	Time	Source	Destination	Protocol	Length	Info
28347	52.873025	169.24.129.15	10.0.2.15	TCP	60	33439 → 80 [SYN] Seq=0 Win=1278 Len=0
28349	52.873764	68.55.151.74	10.0.2.15	TCP	60	57311 → 80 [SYN] Seq=0 Win=1444 Len=0
28351	52.874419	169.24.129.15	10.0.2.15	TCP	60	52978 → 80 [SYN] Seq=0 Win=1475 Len=0
28353	52.878634	169.24.129.15	10.0.2.15	TCP	60	40439 → 80 [SYN] Seq=0 Win=4022 Len=0
28359	52.878634	169.24.129.15	10.0.2.15	TCP	60	47057 → 80 [SYN] Seq=0 Win=2844 Len=0
28360	52.878634	68.55.151.74	10.0.2.15	TCP	60	[TCP Port numbers reused] 47263 → 80 [SYN] Seq=0
28362	52.878634	68.55.151.74	10.0.2.15	TCP	60	9259 → 80 [SYN] Seq=0 Win=1801 Len=0
28363	52.878634	169.24.129.15	10.0.2.15	TCP	60	57442 → 80 [SYN] Seq=0 Win=3110 Len=0
28371	52.884177	68.55.151.74	10.0.2.15	TCP	60	12655 → 80 [SYN] Seq=0 Win=3916 Len=0
28375	52.884177	68.55.151.74	10.0.2.15	TCP	60	53736 → 80 [SYN] Seq=0 Win=2073 Len=0
28376	52.884177	169.24.129.15	10.0.2.15	TCP	60	7922 → 80 [SYN] Seq=0 Win=3977 Len=0
28380	52.884177	68.55.151.74	10.0.2.15	TCP	60	[TCP Port numbers reused] 1643 → 80 [SYN] Seq=0 W
28381	52.884177	169.24.129.15	10.0.2.15	TCP	60	5439 → 80 [SYN] Seq=0 Win=2878 Len=0
28389	52.886497	68.55.151.74	10.0.2.15	TCP	60	15541 → 80 [SYN] Seq=0 Win=3077 Len=0
28392	52.888971	169.24.129.15	10.0.2.15	TCP	60	2035 → 80 [SYN] Seq=0 Win=2471 Len=0
28393	52.888971	169.24.129.15	10.0.2.15	TCP	60	29340 → 80 [SYN] Seq=0 Win=2947 Len=0
28394	52.888971	68.55.151.74	10.0.2.15	TCP	60	42785 → 80 [SYN] Seq=0 Win=679 Len=0
28398	52.891451	68.55.151.74	10.0.2.15	TCP	60	1587 → 80 [SYN] Seq=0 Win=329 Len=0
28399	52.891451	169.24.129.15	10.0.2.15	TCP	60	37782 → 80 [SYN] Seq=0 Win=2027 Len=0

The IP address of one attacker is 68.55.151.74 and the other 169.24.129.15.
To get the correct IP addresses, it was necessary to go in the middle of the wireshark capture so that we were sure that the attack was already ongoing.

Task 4: Data visualization and exploration - Detecting DDoS attacks using Deep Learning

We first import our train datasets. We will use the *http-dataset.csv* provided by Mr. Bousalem for the normal traffic and our own datasets for the DDoS traffic. *myfirstattack.csv* is a record of an attack using a single Kali while *hugeattack.csv* is an attack using two Kalis.

```
ddos = pd.read_csv('myfirstattack.csv')
ddoshuge = pd.read_csv('hugeattack.csv')
http = pd.read_csv('http-dataset.csv')
```

Merge of the datasets

It will be more convenient to merge the 2 datasets and add a column "Class". 1 will mean DDoS attack while 0 will mean normal traffic.

```
ddos['Class'] = 1 # We add a column with 1 as constant value
ddoshuge['Class'] = 1
http['Class'] = 0 # We add a column with 0 as constant value
```

```
dataset = pd.concat([ddos, ddoshuge, http], axis=0).reset_index()
dataset.drop('index', 1, inplace=True)
```

Our final train dataset looks like :

```
dataset
```

	No.	Time	Source	Destination	Protocol	Length	Info	Class
0	1	0.000000	10.0.2.15	52.167.254.228	TCP	54	49197 > 80 [FIN, ACK] Seq=1 Ack=1 Win=62966 ...	1
1	2	0.002624	52.167.254.228	10.0.2.15	TCP	60	80 > 49197 [ACK] Seq=1 Ack=2 Win=32767 Len=0	1
2	3	0.204789	52.167.254.228	10.0.2.15	TCP	60	80 > 49197 [FIN, ACK] Seq=1 Ack=2 Win=32767 ...	1
3	4	0.204888	10.0.2.15	52.167.254.228	TCP	54	49197 > 80 [ACK] Seq=2 Ack=2 Win=62966 Len=0	1
4	5	22.343940	PcsCompu_43:73:bc	Broadcast	ARP	60	Who has 10.0.2.1? Tell 10.0.2.5	1
...	...	...	...	...	...	...	...	...
343532	24320	860.774820	10.42.0.2	10.42.0.1	TCP	74	80 > 53654 [SYN, ACK] Seq=0 Ack=1 Win=28960 ...	0
343533	24321	860.775115	10.42.0.1	10.42.0.2	TCP	66	53654 > 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0...	0
343534	24322	860.775241	10.42.0.1	10.42.0.2	HTTP	256	GET / HTTP/1.1	0
343535	24323	860.775314	10.42.0.2	10.42.0.1	TCP	66	80 > 53654 [ACK] Seq=1 Ack=191 Win=30080 Len...	0
343536	24324	860.776418	10.42.0.2	10.42.0.1	TCP	7306	80 > 53654 [ACK] Seq=1 Ack=191 Win=30080 Len...	0

343537 rows × 8 columns

Analysis of all the columns

No.

This column is only used to know the order of the packets in Wireshark. We can extract this order from the Time column. Thus, it is not really useful to keep this column.

```
dataset.drop('No.', 1, inplace=True) # Removing column number
```

Time

As explained above, this column can be used to order the data. Furthermore, as we are dealing with DDoS attacks, it can be really useful. Indeed, this kind of attack is meant to completely overwhelm a machine. Thus, having a lot of packets arriving in a small time frame can be a good indicator. We will keep this column. No specific processing is required on this column at this time, it is clean data.

```
: dataset["Time"]
```

```
0           0.000000
1           0.002624
2           0.204789
3           0.204888
4          22.343940
             ...
343532    860.774820
343533    860.775115
343534    860.775241
343535    860.775314
343536    860.776418
Name: Time, Length: 343537, dtype: float64
```

Source

This column allows to know the sender of a request. In the context of a DDoS, many requests will be sent from the same IP address at a short interval of time. That is why it is interesting to keep this column.

```
: dataset["Source"]
```

```
0                 10.0.2.15
1            52.167.254.228
2            52.167.254.228
3                 10.0.2.15
4         PcsCompu_43:73:bc
                ...
343532            10.42.0.2
343533            10.42.0.1
343534            10.42.0.1
343535            10.42.0.2
343536            10.42.0.2
Name: Source, Length: 343537, dtype: object
```

Destination

Idem as source.

```
]: dataset["Destination"]

0           52.167.254.228
1                10.0.2.15
2                10.0.2.15
3           52.167.254.228
4                Broadcast
                      ...
343532           10.42.0.1
343533           10.42.0.2
343534           10.42.0.2
343535           10.42.0.1
343536           10.42.0.1
Name: Destination, Length: 343537, dtype: object
```

Protocol

The protocol used is quite indicative of the traffic that takes place on a machine. Indeed, we notice for example that TCP tends to be the most used protocol for DDoS attacks. Nevertheless, this is a very clean dataset and in reality there will be noise coming from the internet or from other machines communicating with the machine to be analyzed.

We will keep this column.

```
]: dataset["Protocol"]

0          TCP
1          TCP
2          TCP
3          TCP
4          ARP
          ...
343532     TCP
343533     TCP
343534    HTTP
343535     TCP
343536     TCP
Name: Protocol, Length: 343537, dtype: object
```

Length

This column contains the size of the packets. It can be useful so we keep it.

```
5]: dataset['Length'].describe()

count    186825.000000
mean        208.109257
std        1050.053021
min          42.000000
25%          54.000000
50%          60.000000
75%          60.000000
max       11239.000000
Name: Length, dtype: float64
```

Info

This column contains a variety of things. We could tokenize its content but for the moment we will try to do without this column. If our model is not performing well, we will add it if necessary.

We will not use this column for the moment.

```
]: dataset["Info"]
```

```
0           49197  >  80 [FIN, ACK] Seq=1 Ack=1 Win=62966 ...
1              80  >  49197 [ACK] Seq=1 Ack=2 Win=32767 Len=0
2           80  >  49197 [FIN, ACK] Seq=1 Ack=2 Win=32767 ...
3              49197  >  80 [ACK] Seq=2 Ack=2 Win=62966 Len=0
4                             Who has 10.0.2.1? Tell 10.0.2.5
                                                ...
343532      80  >  53654 [SYN, ACK] Seq=0 Ack=1 Win=28960 ...
343533      53654  >  80 [ACK] Seq=1 Ack=1 Win=64256 Len=0...
343534                                        GET / HTTP/1.1
343535      80  >  53654 [ACK] Seq=1 Ack=191 Win=30080 Len...
343536      80  >  53654 [ACK] Seq=1 Ack=191 Win=30080 Len...
Name: Info, Length: 343537, dtype: object
```

```
]: dataset.drop('Info', 1, inplace=True) # Removing column info
```

Final columns

In the end, we obtain a structure of 5 features. We will now try to build a first model.

	Time	Source	Destination	Protocol	Length	Class
0	0.000000	10.0.2.15	52.167.254.228	TCP	54	1
1	0.002624	52.167.254.228	10.0.2.15	TCP	60	1
2	0.204789	52.167.254.228	10.0.2.15	TCP	60	1
3	0.204888	10.0.2.15	52.167.254.228	TCP	54	1
4	22.343940	PcsCompu_43:73:bc	Broadcast	ARP	60	1
...	...	...	...	...	...	...
343532	860.774820	10.42.0.2	10.42.0.1	TCP	74	0
343533	860.775115	10.42.0.1	10.42.0.2	TCP	66	0
343534	860.775241	10.42.0.1	10.42.0.2	HTTP	256	0
343535	860.775314	10.42.0.2	10.42.0.1	TCP	66	0
343536	860.776418	10.42.0.2	10.42.0.1	TCP	7306	0

343537 rows × 6 columns

One-hot encoding

As the data is kind of raw for some columns (such as string columns), we need to convert it to a format easily readable by our model. To do such a thing, we will use one-hot encoding. One-hot encoding consists in converting categorical data into one-hot k bits arrays (with k the number of columns).

```
Example :

        0
0     cat
1     dog
2     cat
3     cat
4   horse

will be encoded as :

[[1 0 0]
 [0 1 0]
 [1 0 0]
 [1 0 0]
 [0 0 1]]
```

The conversion function

Let's create an algorithm to do this one-hot encoding quickly. First, we will create a function that extracts a mapping of all the categories of the data.

```python
def mapping(data):
    data = pd.Series(data)

    categories = data.unique()

    dict_cat = {}

    for i in range(len(categories)):
        dict_cat[categories[i]] = i
    return dict_cat
```

Then, we can create a function that one-hot encodes all the data using this mapping.

```python
def one_hot_encode(data, dict_cat):
    data = pd.Series(data)

    one_hot = []
    for el in data:
        one_hot.append(dict_cat[el])

    return to_categorical(one_hot, num_classes=len(dict_cat.keys()))
```

We can then have an output like the following :

```
The one-hot encoding of ['cat', 'dog', 'dog'] is :

[[1. 0.]
 [0. 1.]
 [0. 1.]]

with the following mapping {'cat': 0, 'dog': 1}
```

Processing of the protocol feature

For the moment, the column "Protocol" is composed of strings.

```python
dataset['Protocol'].unique()
```

```
array(['TCP', 'ARP', 'TLSv1.2', 'DHCPv6', 'BROWSER', 'HTTP', 'DNS', 'SSH',
       'SSDP', 'MDNS', 'ICMP', 'NTP', 'SSHv2'], dtype=object)
```

As we can see above, it is categorical data with 9 categories. We can thus use one-hot encoding to serialize the values.

```python
dict_protocol = mapping(dataset['Protocol'])

one_hot_protocol = one_hot_encode(dataset['Protocol'], dict_protocol)
```

The mapping is the following :

```python
print(dict_protocol)
```

```
{'TCP': 0, 'ARP': 1, 'TLSv1.2': 2, 'DHCPv6': 3, 'BROWSER': 4, 'HTTP': 5, 'DNS': 6, 'SSH': 7, 'SSDP': 8, 'MDNS': 9, 'I
CMP': 10, 'NTP': 11, 'SSHv2': 12}
```

We join the one-hot encoded array of the protocol column. We can then delete the former Protocol column.

```
dataset = dataset.join(pd.DataFrame(one_hot_protocol, columns=dict_protocol.keys()))
```

```
dataset.drop("Protocol", 1, inplace=True)
```

We obtain the following DataFrame :

```
dataset
```

	Time	Source	Destination	Length	Class	TCP	ARP	TLSv1.2	DHCPv6	BROWSER	HTTP	DNS	SSH	SSDP	MDNS	ICMP	NTP
0	0.000000	10.0.2.15	52.167.254.228	54	1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.002624	52.167.254.228	10.0.2.15	60	1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.204789	52.167.254.228	10.0.2.15	60	1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.204888	10.0.2.15	52.167.254.228	54	1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	22.343940	PcsCompu_43:73:bc	Broadcast	60	1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
343532	860.774820	10.42.0.2	10.42.0.1	74	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
343533	860.775115	10.42.0.1	10.42.0.2	66	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
343534	860.775241	10.42.0.1	10.42.0.2	256	0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
343535	860.775314	10.42.0.2	10.42.0.1	66	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
343536	860.776418	10.42.0.2	10.42.0.1	7306	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

343537 rows × 18 columns

Validation set

To check any overfitting pattern in our models, we need to import our own sets. Doing this, we will validate our models on brand new data. However, we need to apply the same processing as above to this dataset. We first import two datasets. *mysecondattack.csv* is a dataset picturing a DDoS attack using a Kali. *normal.csv* is a dataset of traffic while we simply navigate on the Internet. We apply the right classification for both.

```
att = pd.read_csv('mysecondattack.csv')
normal = pd.read_csv('normal.csv')
att['Class'] = 1
normal["Class"] = 0
```

We get the following dataset :

```
val_dataset
```

	No.	Time	Source	Destination	Protocol	Length	Info	Class
0	1	0.000000	fe80::2584:baa2:5d4d:c86d	ff02::1:2	DHCPv6	152	Solicit XID: 0xd0f8ae CID: 0001000121e8031c080...	1
1	2	32.019548	fe80::2584:baa2:5d4d:c86d	ff02::1:2	DHCPv6	152	Solicit XID: 0xd0f8ae CID: 0001000121e8031c080...	1
2	3	227.921904	PcsCompu_43:73:bc	Broadcast	ARP	60	Who has 10.0.2.15? Tell 10.0.2.5	1
3	4	227.922218	PcsCompu_fe:ec:1e	PcsCompu_43:73:bc	ARP	42	10.0.2.15 is at 08:00:27:fe:ec:1e	1
4	5	227.998806	142.7.230.241	10.0.2.15	TCP	60	27573 > 80 [SYN] Seq=0 Win=2493 Len=0	1
...	...	...	...	...	...	...	...	...
66717	17017	266.981299	20.42.65.85	10.0.2.15	TCP	60	[TCP Keep-Alive ACK] 443 > 49315 [ACK] Seq=6...	0
66718	17018	268.253266	54.92.160.104	10.0.2.15	TCP	60	11103 > 49298 [FIN, ACK] Seq=6518 Ack=2853 W...	0
66719	17019	268.253370	10.0.2.15	54.92.160.104	TCP	54	49298 > 11103 [ACK] Seq=2853 Ack=6519 Win=63...	0
66720	17020	269.384607	10.0.2.15	18.195.152.201	TCP	55	[TCP Keep-Alive] 49288 > 443 [ACK] Seq=2064 ...	0
66721	17021	269.385185	18.195.152.201	10.0.2.15	TCP	60	[TCP Keep-Alive ACK] 443 > 49288 [ACK] Seq=6...	0

66722 rows × 8 columns

We apply the same transformation as the train dataset. First, we remove the unused columns.

```
val_dataset.drop('No.', 1, inplace=True) # Removing column number
val_dataset.drop('Info', 1, inplace=True) # Removing column info
```

Then, we one-hot encode the Protocol feature.

```
val_dataset = val_dataset[val_dataset.Protocol.isin(dict_protocol.keys())] # Removal of outliers

one_hot_protocol = one_hot_encode(val_dataset['Protocol'], dict_protocol)
```

```
val_dataset = val_dataset.join(pd.DataFrame(one_hot_protocol, columns=dict_protocol.keys()))
```

```
val_dataset.dropna(inplace=True)
val_dataset = val_dataset.drop("Protocol", 1)
```

We obtain the following dataset :

```
val_dataset
```

	Time	Source	Destination	Protocol	Length	Class	TCP	ARP	TLSv1.2	DHCPv6	BROWSER	HTTP	DNS	SSH	SSDP
0	0.000000	fe80::2584:baa2:5d4d:c86d	ff02::1:2	DHCPv6	152	1	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	32.019548	fe80::2584:baa2:5d4d:c86d	ff02::1:2	DHCPv6	152	1	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
2	227.921904	PcsCompu_43:73:bc	Broadcast	ARP	60	1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	227.922218	PcsCompu_fe:ec:1e	PcsCompu_43:73:bc	ARP	42	1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	227.998806	142.7.230.241	10.0.2.15	TCP	60	1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
66717	266.981299	20.42.65.85	10.0.2.15	TCP	60	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
66718	268.253266	54.92.160.104	10.0.2.15	TCP	60	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
66719	268.253370	10.0.2.15	54.92.160.104	TCP	54	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
66720	269.384607	10.0.2.15	18.195.152.201	TCP	55	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
66721	269.385185	18.195.152.201	10.0.2.15	TCP	60	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

63887 rows × 19 columns

First Models

The columns Source and Destination are for the moment hard to use. Indeed, if we just one-hot encode them, there will be a huge overfitting as the DDoS dataset has an unique source IP and same for destination IP.

```
final = dataset.drop(['Source', 'Destination'], 1).copy()
```

```
col_order = list(final.columns.values)
```

```
final = final[col_order]
```

```
val_final = val_dataset.drop(['Source', 'Destination'], 1).copy()
```

```
val_final = val_final[col_order]
```

For this first model, we will not use the IP. We will introduce them when we will use a sliding-window based model.

To start our model, we create four variables : X_train for training data, y_train for training labels, X_test for testing data and y_test for testing labels.

```
X_train = final.drop('Class', 1).to_numpy()

dict_y = mapping(final['Class'])

y_train = one_hot_encode(final['Class'], dict_y)
```

```
X_test = val_final.drop('Class', 1).to_numpy()

y_test = one_hot_encode(val_final['Class'], dict_y)
```

Model 1

To create our model, we firstly need to import the corresponding libraries.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
```

The first model that we have chosen is a simple one. It is a two-layers network. We use sigmoid activation on the output layer and the binary cross entropy as the loss function because they are characteristic of the binary classification.

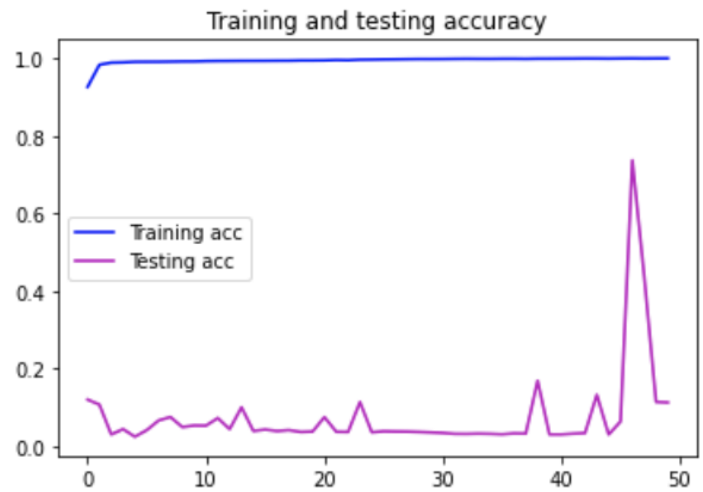
```
model = Sequential()

model.add(layers.Flatten())
model.add(layers.Dense(16, activation="relu"))
model.add(layers.Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', metrics=['acc'], loss='binary_crossentropy')
```

We then fit the model to our data :

```
: history = model.fit(X_train, y_train,  validation_data=(X_test, y_test), batch_size=200, epochs= 50, verbose=0)
```

And we obtain the following metrics :

```
Train accuracy : 0.9993799924850464
Test accuracy  : 0.11305763572454453
```



As you can see, the model undergoes a very very strong overfitting. This is characterized by a train accuracy much higher than the test accuracy and a test loss that is really high, with random spikes.

Using evaluate, we can see that the validation/testing accuracy is really low and the loss really high.

```
8]: model.evaluate(X_test, y_test, batch_size=200)

  308/308 [==============================] - 0s 1ms/step - loss: 8.1080 - acc: 0.1131

: [8.107953071594238, 0.11305763572454453]
```

We then use a confusion matrix to get more details on this model.

```
y_pred = model.predict(X_test)
y_pred=np.argmax(y_pred, axis=1)
y_test=np.argmax(y_test, axis=1)
```

```
from sklearn.metrics import classification_report, confusion_matrix

100 * confusion_matrix(y_test, y_pred) / len(y_pred)
```

```
.rray([[ 8.42559427, 72.43894497],
       [16.25529144,  2.88016933]])
```

As we can see, we have a lot of false positives. This is due to the fact that the DDoS data is omnipresent in our dataset.

Using a balanced accuracy score will help with this imbalanced dataset. We get, however, almost the same accuracy.

```
: balanced_accuracy_score(y_test, y_pred)

0.1273543468863646
```

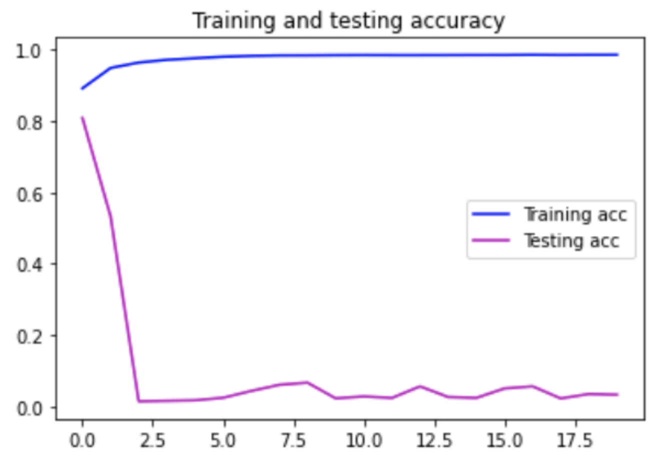
Model 2

To fight overfitting, we will enhance our first model using Dropout layers.

```
model = Sequential()

model.add(layers.Flatten())
model.add(layers.Dense(16, activation="relu"))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation="relu"))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation="relu"))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', metrics=['acc'], loss='binary_crossentropy')
```

```
history = model.fit(X_train, y_train,  validation_data=(X_test, y_test), batch_size=200, epochs= 20, verbose=0)
```

```
Train accuracy : 0.9854833483695984
Test accuracy  : 0.03309996798634529
```



As we can see, the model is even worse and is overfitting a lot. The loss is clearly increasing this time.

The model evaluation gives a horrible score, worse than the first model.

```
.]: model.evaluate(X_test, y_test, batch_size=200)
```

```
308/308 [==============================] - 1s 2ms/step - loss: 54.2673 - acc: 0.0331

[54.26731872558594, 0.03309996798634529]
```

The confusion matrix shows that there are even more false positives and the balanced score is terrible.

```
3]: from sklearn.metrics import balanced_accuracy_score, confusion_matrix

    100 * confusion_matrix(y_true, y_pred) / len(y_pred)
```

```
array([[3.25626832e-03, 8.08612830e+01],
       [1.58287203e+01, 3.30674048e+00]])
```

```
4]: balanced_accuracy_score(y_true, y_pred)
```

```
0.08642360554715660
```

As we can see, we have a clear overfitting, no matter which model we use. We need to modify our dataset to make it less "overfittable".

A model using a Sliding Window

As we have seen before, predicting line by line is almost impossible. We have to use the notion of time implied by the column "Time". For this, we will use what is called Sliding Windows. It consists in grouping the rows of our dataset in groups of length k. We will also deal with IPs in this section.

Sliding window function

We will use a window length of 5. Here is a basic function to create sliding windows from our data.

```
def windows_process(data, window_len = 5):

    data = np.array(data)
    windows = []
    classes = []
    for i in range(len(data) - window_len):
        window = []
        for j in range(window_len):
            window.append(data[i+j])
        class_name = window[0][4]
        classes.append(class_name)
        windows.append(window)
    windows = np.array(windows)
    classes = np.array(classes)
    return windows, classes
```

```
11]: test = dataset[:30]
     print("From a shape of", test.to_numpy().shape)
     print("To a shape of", windows_process(test)[0].shape)
```

```
From a shape of (30, 18)
To a shape of (25, 5, 18)
```

This function is working but we need more in our case.

Time normalization

In our case, we need to normalize a lot of things. First of all, we need to normalize time values. To do such a thing, we will make sure that every window starts at 0.

```
def normalize_time(window):
    start_time = window[0][0]
    for i in range(len(window)):
        window[i][0] = window[i][0] - start_time

    return window
```

Introducing IPs

We have to use IPs in a special way. Indeed, if we do a simple one hot encoding on the whole dataset, there will be overfitting as a DDoS comes from a certain IP. What we are going to do is to do a one hot encoding of the IPs inside the windows to detect IP repetitions.

```
def IP_process(window):
    dict_ip = mapping(window[:,1:3].flatten())

    for i in range(len(window)):
        window[i][1] = dict_ip[ window[i][1]]
        window[i][2] = dict_ip[ window[i][2]]

    return window
```

Processing our dataset

We then apply all these functions to our datasets.

```
time_dataset, classes = windows_process(dataset)
```

```
time_dataset_test, classes_test = windows_process(val_dataset)
```

```
for i in range(len(time_dataset)):
    time_dataset[i] = normalize_time(time_dataset[i])
    time_dataset[i] = IP_process(time_dataset[i])
```

```
for i in range(len(time_dataset_test)):
    time_dataset_test[i] = normalize_time(time_dataset_test[i])
    time_dataset_test[i] = IP_process(time_dataset_test[i])
```

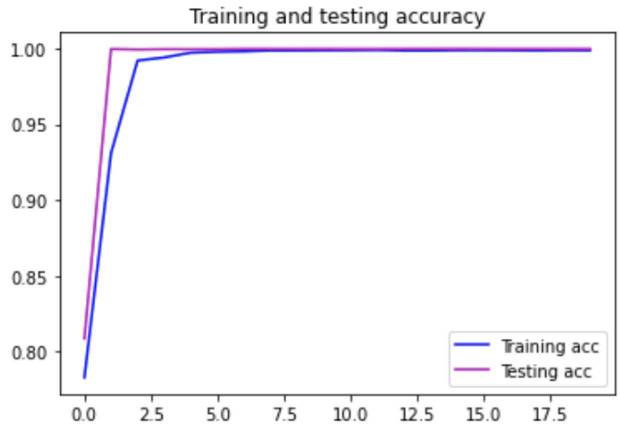
A new model

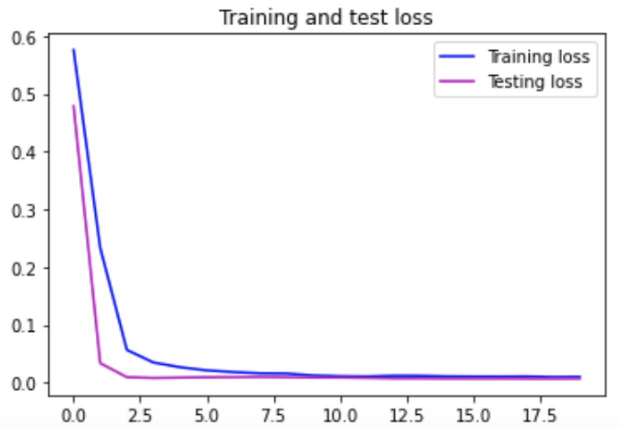
We have totally changed the structure of our model. First of all, we use an LSTM (Long Short Term Memory) layer which is a recursive layer that allows us to take into account time windows. We also added a lot of regularization, normalization and dropout in order to reduce overfitting.

```
from tensorflow.keras import regularizers
model = Sequential()

model.add(layers.Normalization())

model.add(layers.LSTM(2, return_sequences=True))
model.add(layers.Flatten())

model.add(layers.Dense(4,activation='relu', kernel_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4), bias_regularizer=r

model.add(layers.Dropout(.2))
model.add(layers.Dense(8,activation='relu', kernel_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4), bias_regularizer=r


model.add(layers.Dropout(.2))
model.add(layers.Dense(2,activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])


history = model.fit(time_dataset, y_train, validation_data=(time_dataset_test, y_test), epochs = 20, batch_size=500, v
```

As we can see, we have eliminated the overfitting and get a fairly efficient model. We recall that our train and test data come from two different datasets.

```
Train accuracy : 0.998715341091156
Test accuracy  : 0.9995929598808289
```





According to the confusion matrix below, the true negatives and true positives are predominant; this shows that the model is performing well. The balanced accuracy score is also pretty high, almost 100%.

```
]: y_pred = model.predict(time_dataset_test)
   y_pred=np.argmax(y_pred, axis=1)
   y_true=np.argmax(y_test, axis=1)
```

```
]: from sklearn.metrics import balanced_accuracy_score, confusion_matrix

   100 * confusion_matrix(y_true, y_pred) / len(y_pred)
```

```
array([[8.08304160e+01, 4.07066678e-02],
       [0.00000000e+00, 1.91288773e+01]])
```

```
]: balanced_accuracy_score(y_true, y_pred)
```

```
0.9997483238367528
```

This model is really good and we will keep this one. We save the model in our folder.

```
model.save("model.h5")
```

This detection method using deep learning is clearly different from the previous one. The latter only looks for ACKs that have no SYN while the model we just created looks at the traffic more globally. In particular, it uses the temporal factor to detect attacks.