

Lab 2 : CSE 508 Network Security

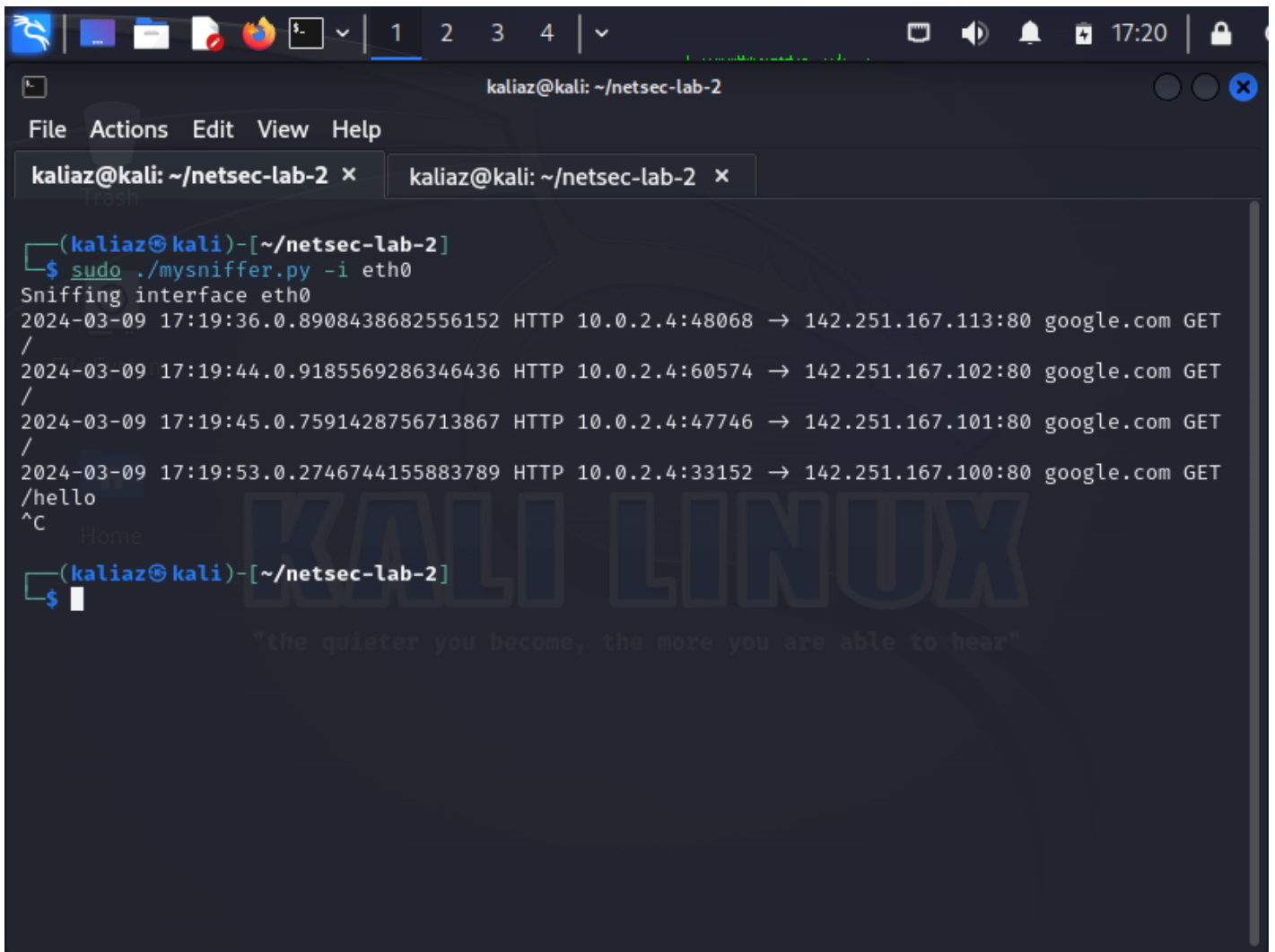
Packet Sniffer

This is a basic packet sniffer made in accordance with lab 2 of CSE 508 Network Security Course. This Python script implements a simple packet sniffer specifically designed for monitoring HTTP and TLS traffic. It utilizes the Scapy library to capture packets over a specified network interface or from a pcap file. The script is capable of identifying and displaying information about HTTP requests (including method, host, and path) as well as details about TLS sessions (such as the version and server name, if available). Users can specify the network interface to monitor or a pcap file to read from via command-line arguments. Additionally, the script allows for the application of a Berkeley Packet Filter (BPF) expression to limit the captured traffic according to the user's needs. The program is structured to be user-friendly, offering help messages and usage instructions through its command-line interface.

Sample Usage and Sample Output

[mysniffer.py] With Specified Network Interface

```
sudo ./mysniffer.py -i eth0
```

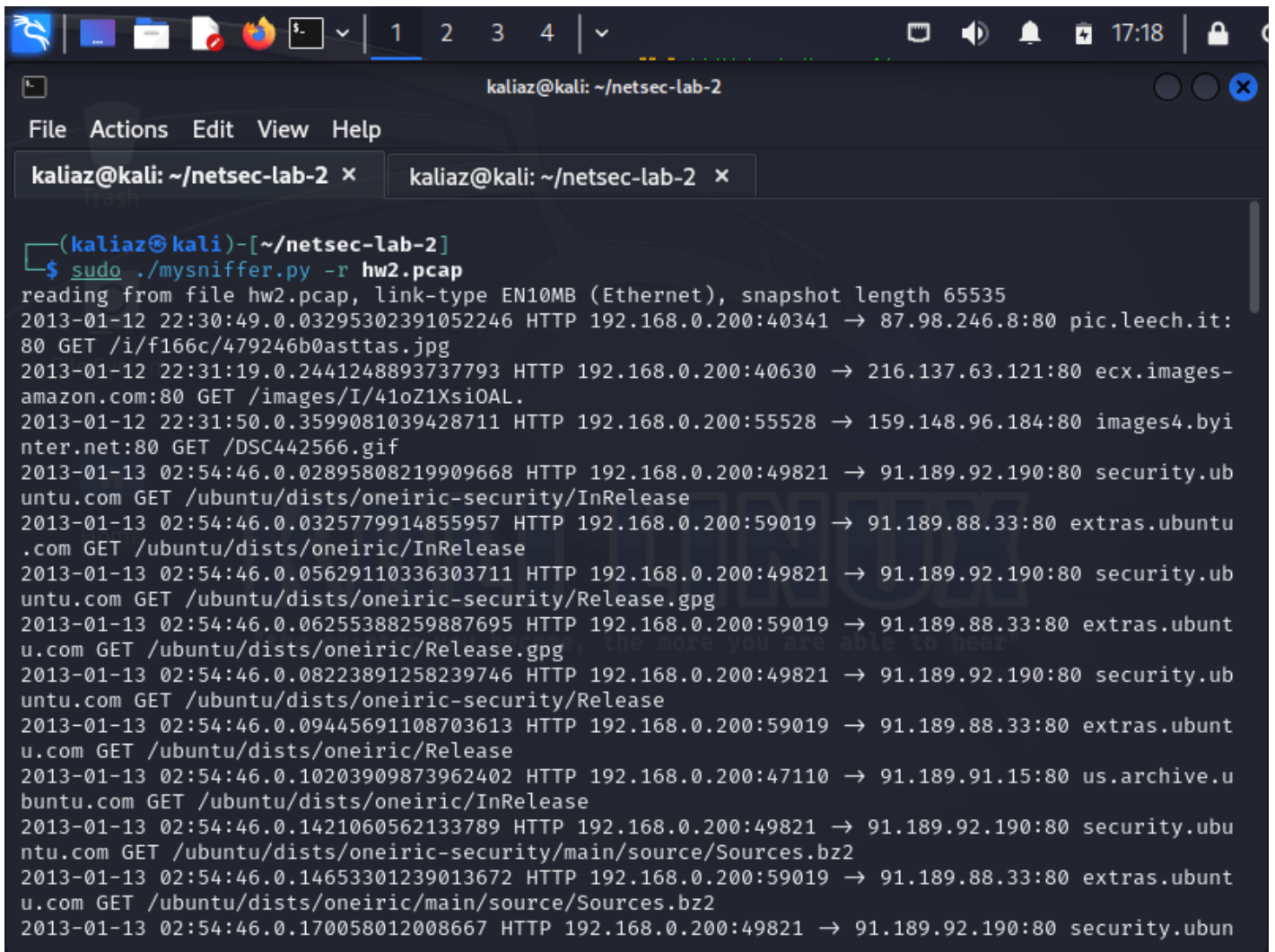


The screenshot shows a Kali Linux terminal window with the title bar 'kaliaz@kali: ~/netsec-lab-2'. The terminal has a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. Two tabs are open, both labeled 'kaliaz@kali: ~/netsec-lab-2'. The terminal content shows the user running the command `sudo ./mysniffer.py -i eth0`. The output indicates that the script is sniffing interface `eth0` and displays four captured HTTP requests to `google.com` with their respective timestamps, source IP addresses, and destination ports. The user then presses `^C` to stop the script, and the prompt returns to `(kaliaz@kali)-[~/netsec-lab-2]` with a cursor on the next line.

```
(kaliaz@kali)-[~/netsec-lab-2]
$ sudo ./mysniffer.py -i eth0
Sniffing interface eth0
2024-03-09 17:19:36.0.8908438682556152 HTTP 10.0.2.4:48068 → 142.251.167.113:80 google.com GET
/
2024-03-09 17:19:44.0.9185569286346436 HTTP 10.0.2.4:60574 → 142.251.167.102:80 google.com GET
/
2024-03-09 17:19:45.0.7591428756713867 HTTP 10.0.2.4:47746 → 142.251.167.101:80 google.com GET
/
2024-03-09 17:19:53.0.2746744155883789 HTTP 10.0.2.4:33152 → 142.251.167.100:80 google.com GET
/hello
^C
(kaliaz@kali)-[~/netsec-lab-2]
$
```

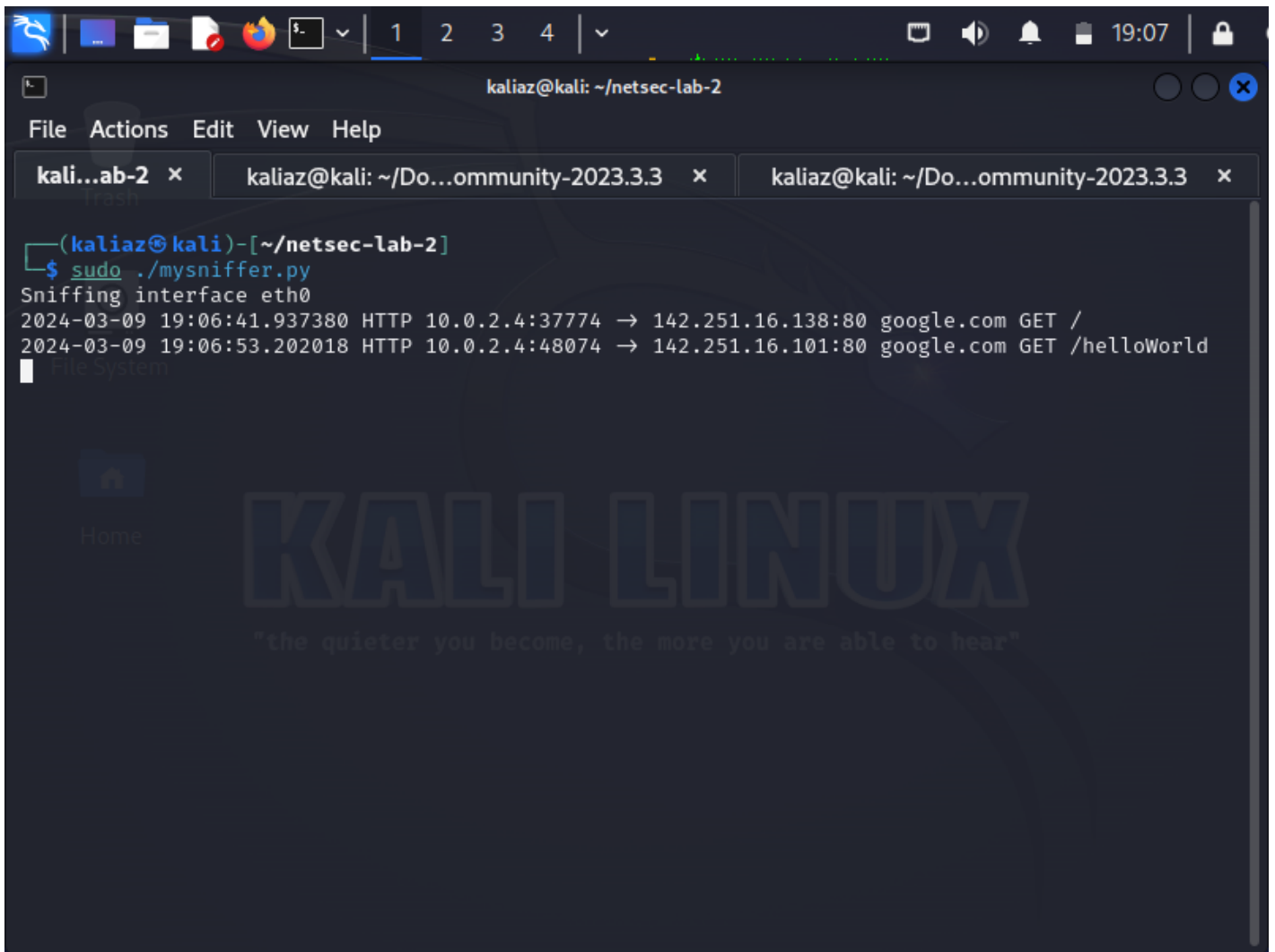
[mysniffer.py] With Specified PCAP file

```
sudo ./mysniffer.py -r hw2.pcap
```

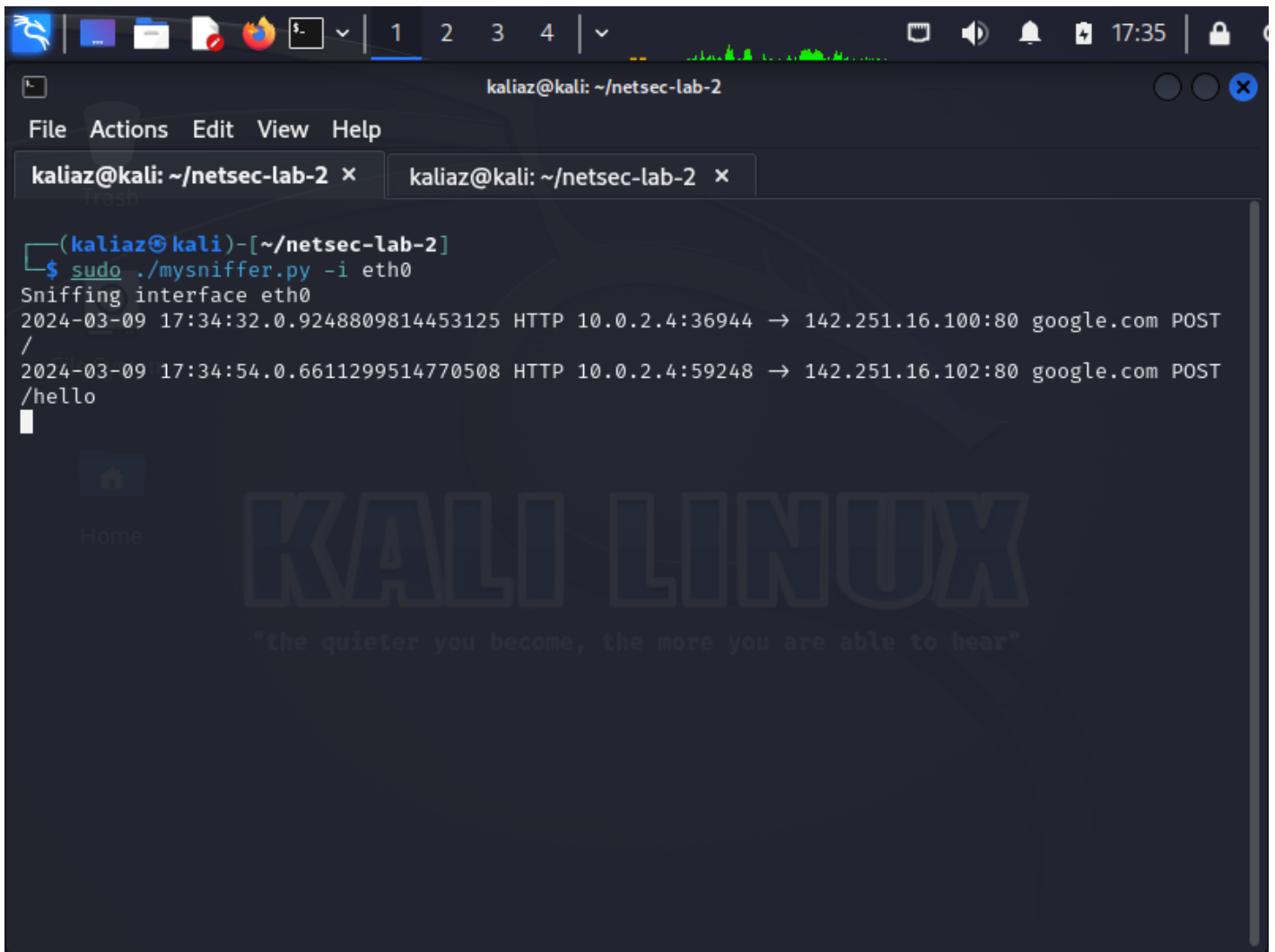


```
(kaliaz@kali)-[~/netsec-lab-2]
$ sudo ./mysniffer.py -r hw2.pcap
reading from file hw2.pcap, link-type EN10MB (Ethernet), snapshot length 65535
2013-01-12 22:30:49.0.03295302391052246 HTTP 192.168.0.200:40341 → 87.98.246.8:80 pic.leech.it:
80 GET /i/f166c/479246b0asttas.jpg
2013-01-12 22:31:19.0.2441248893737793 HTTP 192.168.0.200:40630 → 216.137.63.121:80 ecx.images-
amazon.com:80 GET /images/I/41oZ1XsioAL.
2013-01-12 22:31:50.0.3599081039428711 HTTP 192.168.0.200:55528 → 159.148.96.184:80 images4.byi
nter.net:80 GET /DSC442566.gif
2013-01-13 02:54:46.0.02895808219909668 HTTP 192.168.0.200:49821 → 91.189.92.190:80 security.ub
untu.com GET /ubuntu/dists/oneiric-security/InRelease
2013-01-13 02:54:46.0.0325779914855957 HTTP 192.168.0.200:59019 → 91.189.88.33:80 extras.ubuntu
.com GET /ubuntu/dists/oneiric/InRelease
2013-01-13 02:54:46.0.05629110336303711 HTTP 192.168.0.200:49821 → 91.189.92.190:80 security.ub
untu.com GET /ubuntu/dists/oneiric-security/Release.gpg
2013-01-13 02:54:46.0.06255388259887695 HTTP 192.168.0.200:59019 → 91.189.88.33:80 extras.ubunt
u.com GET /ubuntu/dists/oneiric/Release.gpg
2013-01-13 02:54:46.0.08223891258239746 HTTP 192.168.0.200:49821 → 91.189.92.190:80 security.ub
untu.com GET /ubuntu/dists/oneiric-security/Release
2013-01-13 02:54:46.0.09445691108703613 HTTP 192.168.0.200:59019 → 91.189.88.33:80 extras.ubunt
u.com GET /ubuntu/dists/oneiric/Release
2013-01-13 02:54:46.0.10203909873962402 HTTP 192.168.0.200:47110 → 91.189.91.15:80 us.archive.u
buntu.com GET /ubuntu/dists/oneiric/InRelease
2013-01-13 02:54:46.0.1421060562133789 HTTP 192.168.0.200:49821 → 91.189.92.190:80 security.ubu
ntu.com GET /ubuntu/dists/oneiric-security/main/source/Sources.bz2
2013-01-13 02:54:46.0.14653301239013672 HTTP 192.168.0.200:59019 → 91.189.88.33:80 extras.ubunt
u.com GET /ubuntu/dists/oneiric/main/source/Sources.bz2
2013-01-13 02:54:46.0.170058012008667 HTTP 192.168.0.200:49821 → 91.189.92.190:80 security.ubun
```

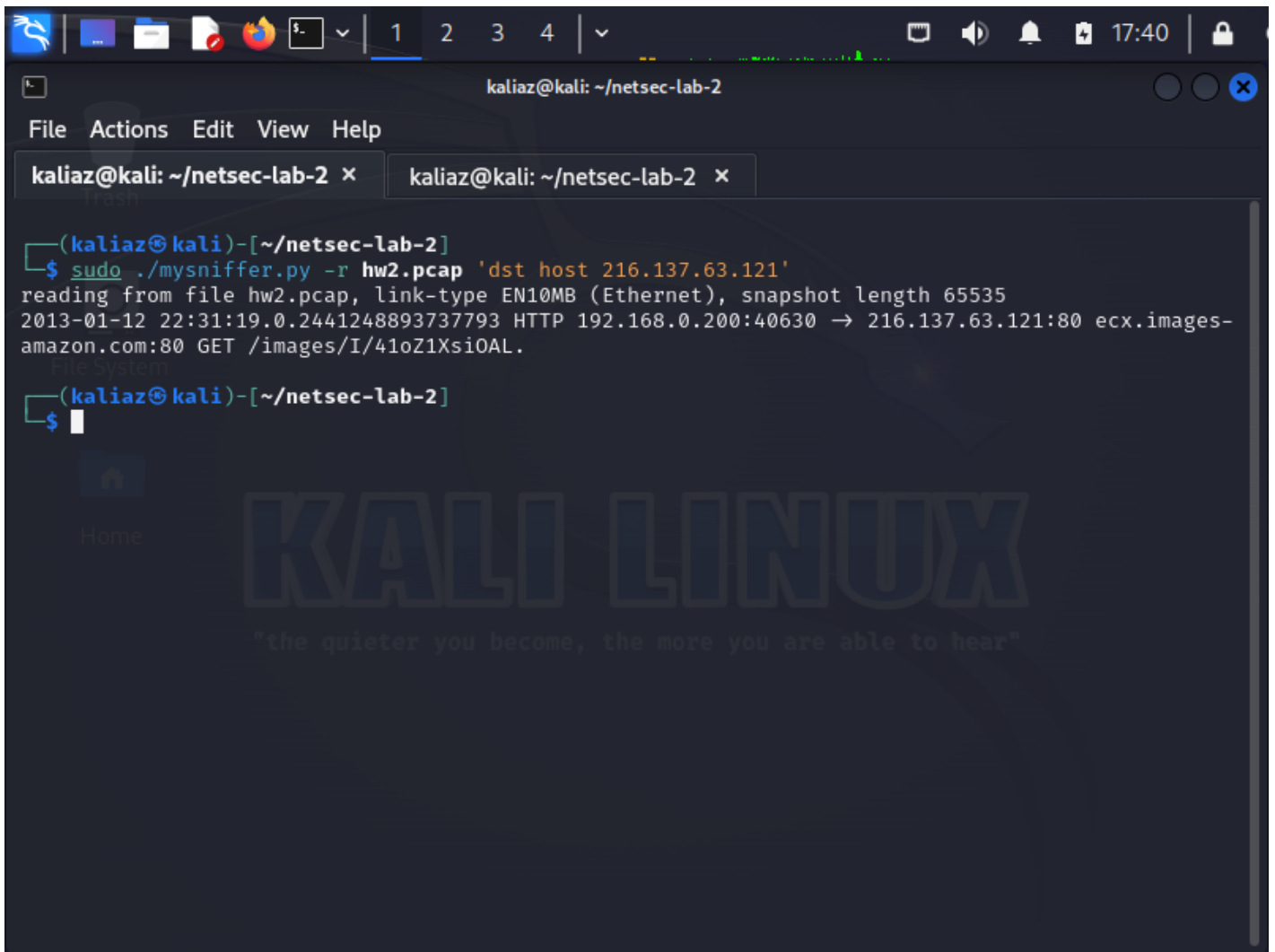
[mysniffer.py] With Default Interface (No interface and No pcap file provided)



[mysniffer.py] POST Requests



[mysniffer.py] BPF Filter



Testing

[mysniffer.py] Different Versions of TLS on standard port

For testing on standard port on different versions of TLS, use the below curl commands

```
# TLSv1.3 on Standard Port
curl -X G--tlsv1.3 --tls-max 1.3 --ciphers DEFAULT@SECLEVEL=0 -vI https://www.goo

# TLSv1.2 on Standard Port
curl -X G--tlsv1.2 --tls-max 1.2 --ciphers DEFAULT@SECLEVEL=0 -vI https://www.goo

# TLSv1.1 on Standard Port
curl -X G--tlsv1.1 --tls-max 1.1 --ciphers DEFAULT@SECLEVEL=0 -vI https://www.goo

# TLSv1.0 on Standard Port
curl -X G--tlsv1.0 --tls-max 1.0 --ciphers DEFAULT@SECLEVEL=0 -vI https://www.goo
```

[mysniffer.py] Different Versions of TLS on non-standard port

For testing on non-standard port of TLS, we used the <https://portquiz.takao-tech.com> website. This website is running on https and supports multiple non-standard ports like 9001, 8080, 80, 8, 666, etc. I chose the non-standard port 9001 and different TLS versions for testing. The following curl requests were made for testing tls requests on non-standard ports

```
# TLSv1.3 on Non-Standard Port
curl -X G--tlsv1.3 --tls-max 1.3 --ciphers DEFAULT@SECLEVEL=0 -vI https://portqui

# TLSv1.2 on Non-Standard Port
curl -X G--tlsv1.2 --tls-max 1.2 --ciphers DEFAULT@SECLEVEL=0 -vI https://portqui

# TLSv1.1 on Non-Standard Port
curl -X G--tlsv1.1 --tls-max 1.1 --ciphers DEFAULT@SECLEVEL=0 -vI https://portqui

# TLSv1.0 on Non-Standard Port
curl -X G--tlsv1.0 --tls-max 1.0 --ciphers DEFAULT@SECLEVEL=0 -vI https://portqui
```

[mysniffer.py] Testing on Standard and Non-Standard Port for HTTP

Testing on custom port can be done via website <http://portquiz.net:8080/>. This website is running on http and supports multiple ports like 8080, 80, 8, 666, etc. The following curl requests were made for testing http requests

```
# For port 80, standard port
curl -X GET http://portquiz.net

# For port 8080, non-standard port testing
curl -X GET http://portquiz.net:8080

# For port 8, non-standard port testing
curl -X GET http://portquiz.net:8

# For port 666, non-standard port testing
curl -X GET http://portquiz.net:666
```

Testing of TLS with custom HTTPS Server

This testing method is presented as an alternative for the above testing method for tls and is present in this documentation only for information purposes This testing method must be used only if the above tls testing methods are absolutely not working. The file `tls_server.py` contains a quickly

whipped up `tls_server` for testing purposes. You need to set up this server on a computer and then from a different computer on the same network, you can call this server. Preferably both the computers should be linux. For testing, Macintosh operating system was used. Below are the relevant steps

Step 1: Generate a Self-Signed SSL Certificate

First, use OpenSSL to generate a private key and a self-signed certificate. Open a terminal and run:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

Step 2: Run the python server

```
python3 tls_server.py
```

Step 3: Connect to the Server

For connecting to the server, go on a different computer on the same network, figure out the IP address of the computer where the server is running and then replace `localhost` in the below line with the IP address of the target computer

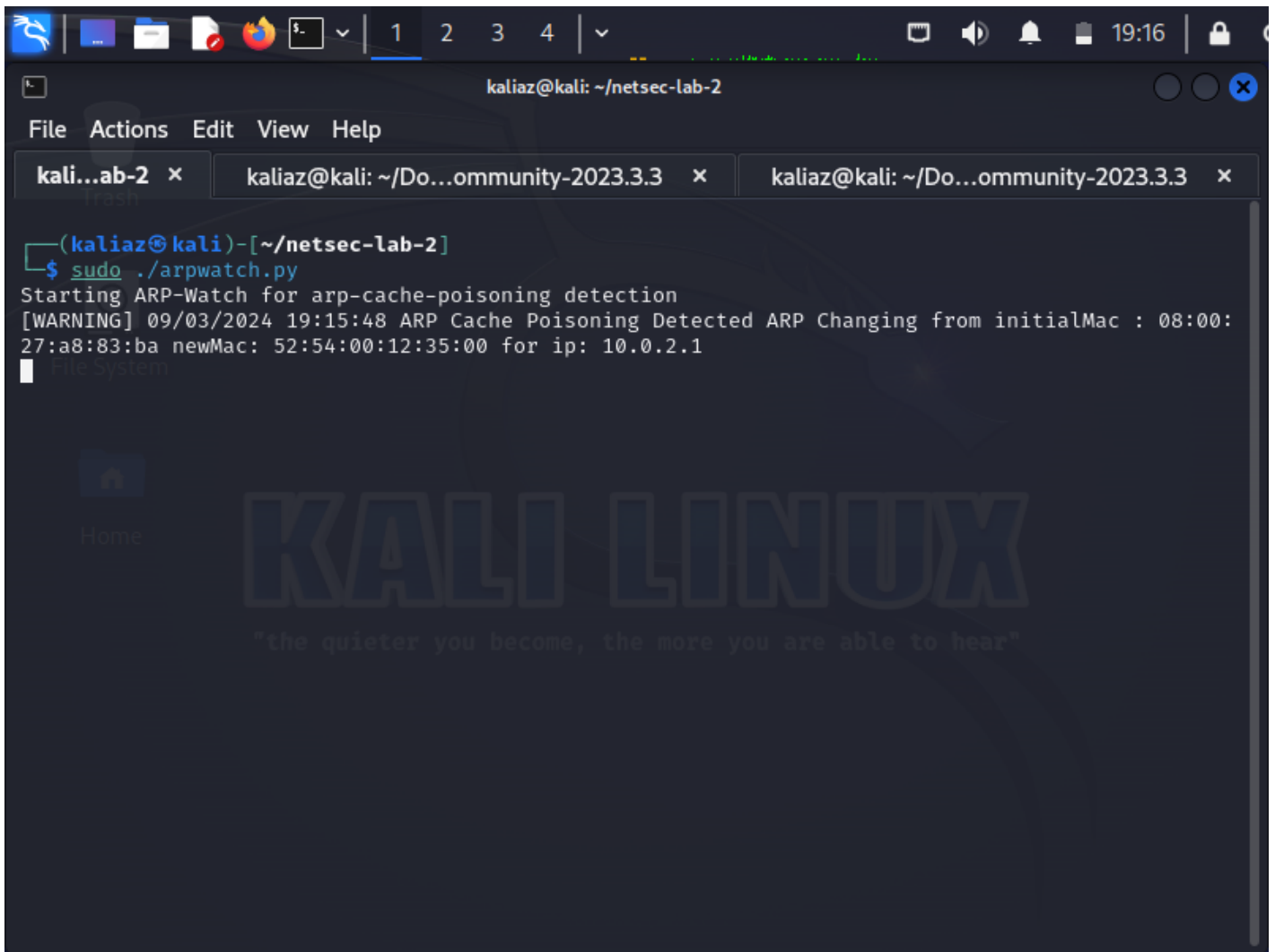
```
openssl s_client -connect localhost:8443
```

Arpwatch

This script is an ARP Cache Poisoning Detector that uses the Scapy library to monitor ARP packets on a network interface, aiming to detect ARP spoofing attacks. It reads the ARP table from the system, listens for ARP responses, and alerts if it finds any discrepancies between the packet's source IP and MAC addresses compared to the initial ARP table entries. Users can specify a network interface for monitoring; otherwise, the script defaults to the system's primary interface. It's designed to be simple and user-friendly, providing basic command-line options for operation.

[arpwatch.py] With Specified Interface

```
sudo ./arpwatch.py -i eth0
```

Let me explain the attacker and victim situation. There is one attacker machine(ubuntu machine) and one victim machine (kali-linux machine). The arpwatch.py script will be run on the victim machine in accordance with the lab requirements. The arpwatch.py script will detect if there is an ongoing arp spoof attack on the victim machine The attacker will launch full man in the middle attack using arpspoof command installed on ubuntu

Attacker IP Address : 10.0.2.15

Victim IP Address : 10.0.2.4

Gateway IP Address : 10.0.2.1

Attacker will have 3 terminals

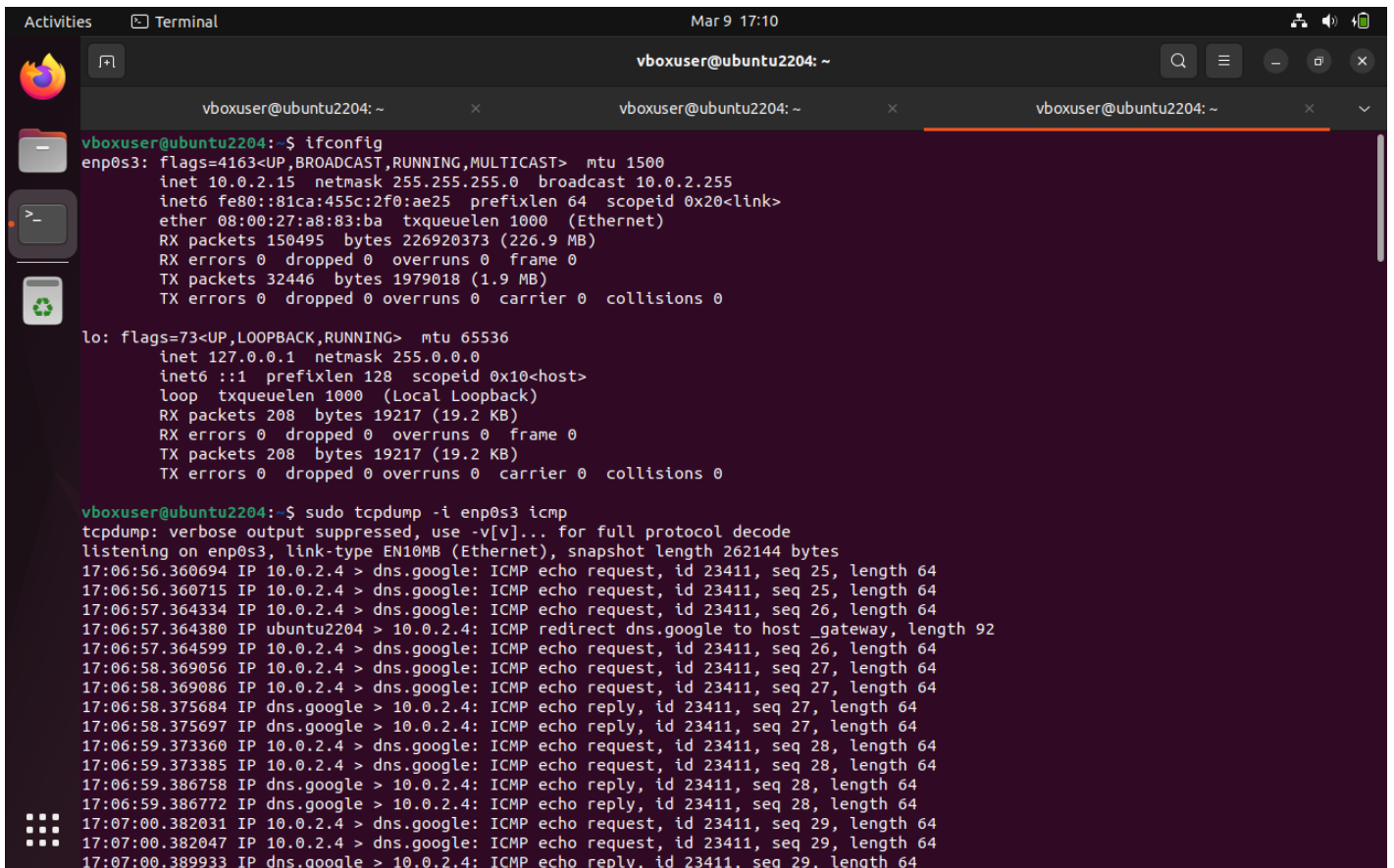
Attacker Terminal 1

```
sudo arpspoof -i enp0s3 -t 10.0.2.4 10.0.2.1
```

[illegible]

Attacker Terminal 2

```
sudo arpspoof -i enp0s3 -t 10.0.2.1 10.0.2.4
```

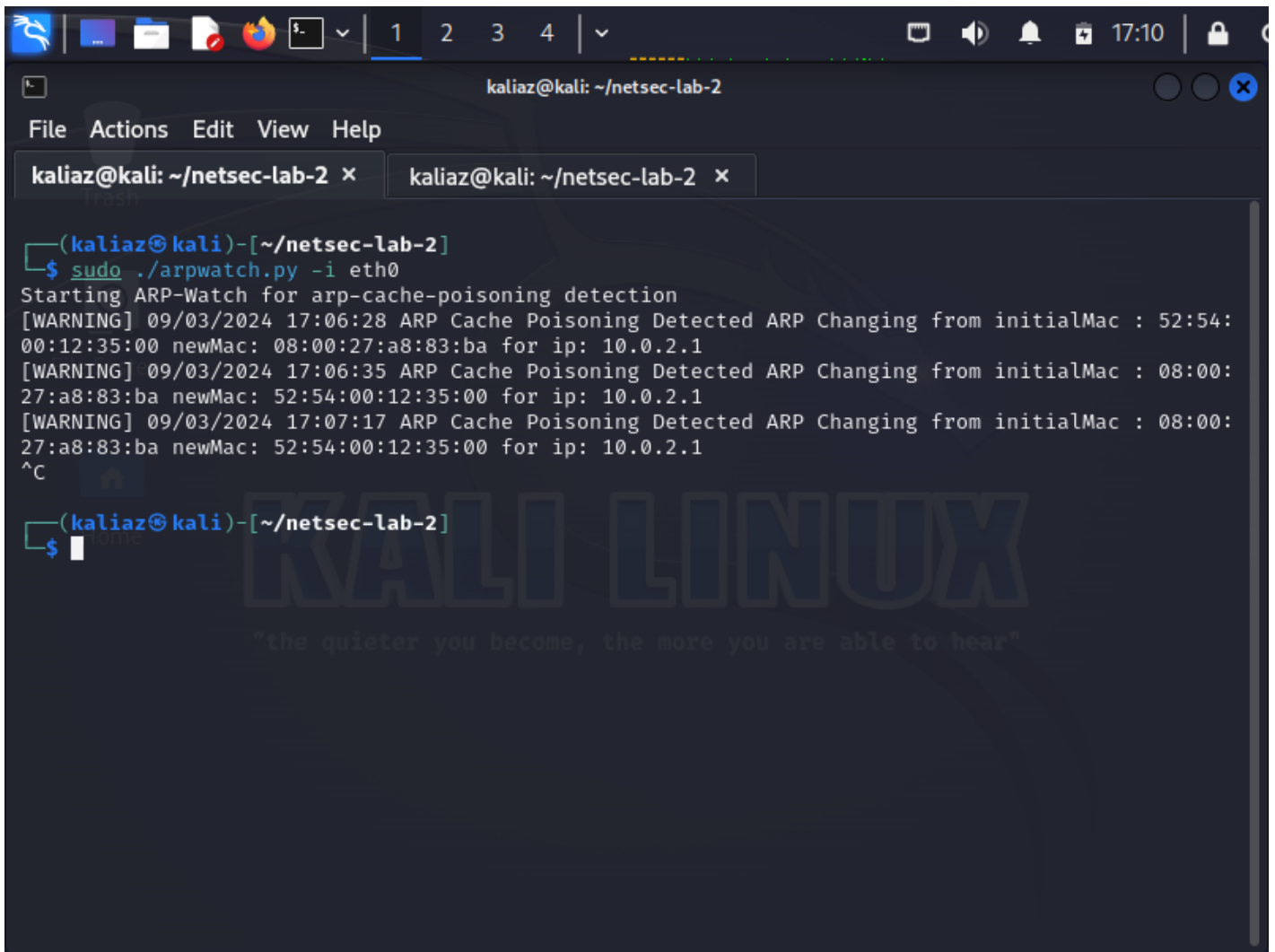
```
vboxuser@ubuntu2204: ~  
vboxuser@ubuntu2204:~$ ifconfig  
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255  
inet6 fe80::81ca:455c:2f0:ae25 prefixlen 64 scopeid 0x20<link>  
ether 08:00:27:a8:83:ba txqueuelen 1000 (Ethernet)  
RX packets 150495 bytes 226920373 (226.9 MB)  
RX errors 0 dropped 0 overruns 0 frame 0  
TX packets 32446 bytes 1979018 (1.9 MB)  
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  
  
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536  
inet 127.0.0.1 netmask 255.0.0.0  
inet6 ::1 prefixlen 128 scopeid 0x10<host>  
loop txqueuelen 1000 (Local Loopback)  
RX packets 208 bytes 19217 (19.2 KB)  
RX errors 0 dropped 0 overruns 0 frame 0  
TX packets 208 bytes 19217 (19.2 KB)  
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  
  
vboxuser@ubuntu2204:~$ sudo tcpdump -i enp0s3 icmp  
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode  
listening on enp0s3, link-type EN10MB (Ethernet), snapshot length 262144 bytes  
17:06:56.360694 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 25, length 64  
17:06:56.360715 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 25, length 64  
17:06:57.364334 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 26, length 64  
17:06:57.364380 IP ubuntu2204 > 10.0.2.4: ICMP redirect dns.google to host _gateway, length 92  
17:06:57.364599 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 26, length 64  
17:06:58.369056 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 27, length 64  
17:06:58.369086 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 27, length 64  
17:06:58.375684 IP dns.google > 10.0.2.4: ICMP echo reply, id 23411, seq 27, length 64  
17:06:58.375697 IP dns.google > 10.0.2.4: ICMP echo reply, id 23411, seq 27, length 64  
17:06:59.373360 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 28, length 64  
17:06:59.373385 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 28, length 64  
17:06:59.386758 IP dns.google > 10.0.2.4: ICMP echo reply, id 23411, seq 28, length 64  
17:06:59.386772 IP dns.google > 10.0.2.4: ICMP echo reply, id 23411, seq 28, length 64  
17:07:00.382031 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 29, length 64  
17:07:00.382047 IP 10.0.2.4 > dns.google: ICMP echo request, id 23411, seq 29, length 64  
17:07:00.389933 IP dns.google > 10.0.2.4: ICMP echo reply, id 23411, seq 29, length 64
```

Victim will have 2 terminal open. One for running the arpwatrch script. One for demo purposes

Victim Terminal 1

```
sudo ./arpwatch.py -i eth0
```

The below image indicates that the arpwatrch.py script can successfully detect the arp spoofing attacks.



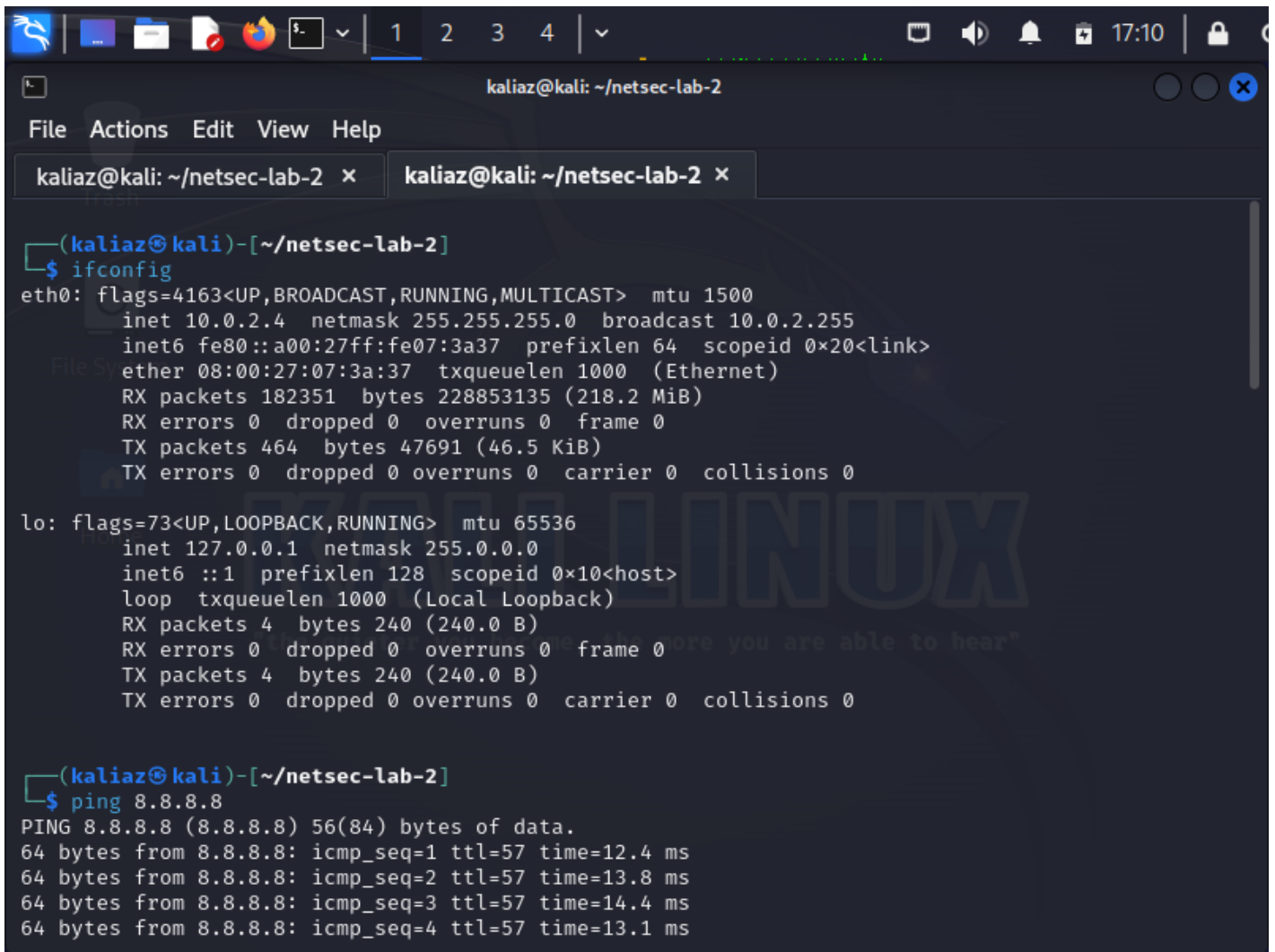
The screenshot shows a Kali Linux terminal window with the title bar 'kaliaz@kali: ~/netsec-lab-2'. The terminal has a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. Two tabs are open, both labeled 'kaliaz@kali: ~/netsec-lab-2'. The terminal content shows the execution of the command `sudo ./arpwatch.py -i eth0`. The output indicates that ARP-Watch is started for arp-cache-poisoning detection. Three warning messages are displayed, each reporting a detected ARP cache poisoning event where the ARP entry for IP 10.0.2.1 changes from an initial MAC address to a new one. The first warning occurs at 17:06:28, the second at 17:06:35, and the third at 17:07:17. The terminal ends with a carriage return (^C) and a new prompt line.

```
(kaliaz@kali)-[~/netsec-lab-2]
$ sudo ./arpwatch.py -i eth0
Starting ARP-Watch for arp-cache-poisoning detection
[WARNING] 09/03/2024 17:06:28 ARP Cache Poisoning Detected ARP Changing from initialMac : 52:54:00:12:35:00 newMac: 08:00:27:a8:83:ba for ip: 10.0.2.1
[WARNING] 09/03/2024 17:06:35 ARP Cache Poisoning Detected ARP Changing from initialMac : 08:00:27:a8:83:ba newMac: 52:54:00:12:35:00 for ip: 10.0.2.1
[WARNING] 09/03/2024 17:07:17 ARP Cache Poisoning Detected ARP Changing from initialMac : 08:00:27:a8:83:ba newMac: 52:54:00:12:35:00 for ip: 10.0.2.1
^C
(kaliaz@kali)-[~/netsec-lab-2]
$
```

Victim Terminal 2

This is used to generate icmp packets for demo purposes

```
ping 8.8.8.8
```

The screenshot shows a Kali Linux terminal window with the title bar 'kaliaz@kali: ~/netsec-lab-2'. The terminal has a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. Below the menu bar, there are two tabs, both labeled 'kaliaz@kali: ~/netsec-lab-2'. The terminal content shows the execution of the 'ifconfig' command, displaying details for the 'eth0' and 'lo' interfaces. The 'eth0' interface is configured with IP 10.0.2.4, netmask 255.255.255.0, and broadcast 10.0.2.255. The 'lo' interface is the loopback address 127.0.0.1. Following this, the 'ping' command is used to test connectivity to 8.8.8.8, showing four successful pings with varying response times.

```
(kaliaz@kali)-[~/netsec-lab-2]
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::a00:27ff:fe07:3a37 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:07:3a:37 txqueuelen 1000 (Ethernet)
    RX packets 182351 bytes 228853135 (218.2 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 464 bytes 47691 (46.5 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 4 bytes 240 (240.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 4 bytes 240 (240.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

(kaliaz@kali)-[~/netsec-lab-2]
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=57 time=12.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=57 time=13.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=57 time=14.4 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=57 time=13.1 ms
```

Interesting Facts

When reading the `/proc/net/arp` file, the call may block if the file is not updated. I would highly advise against reading the file continuously for each packet. This is because updates to this file don't happen after every packet and if the file is not updated then the call to read the file will block your code.