

Lab 2 : CSE 508 Network Security

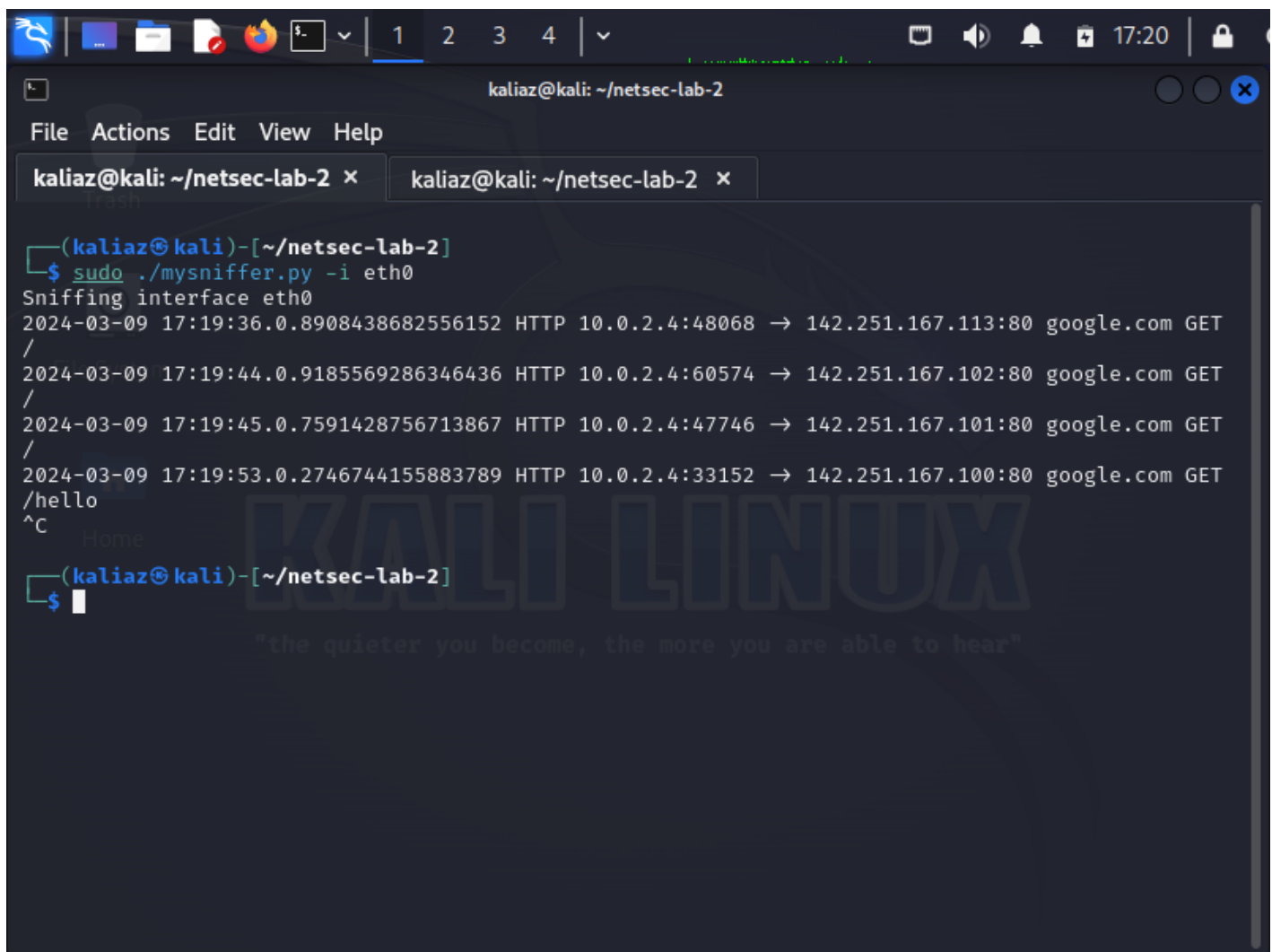
Packet Sniffer

This is a basic packet sniffer made in accordance with lab 2 of CSE 508 Network Security Course.

Sample Usage and Sample Output

[mysniffer.py] With Specified Network Interface

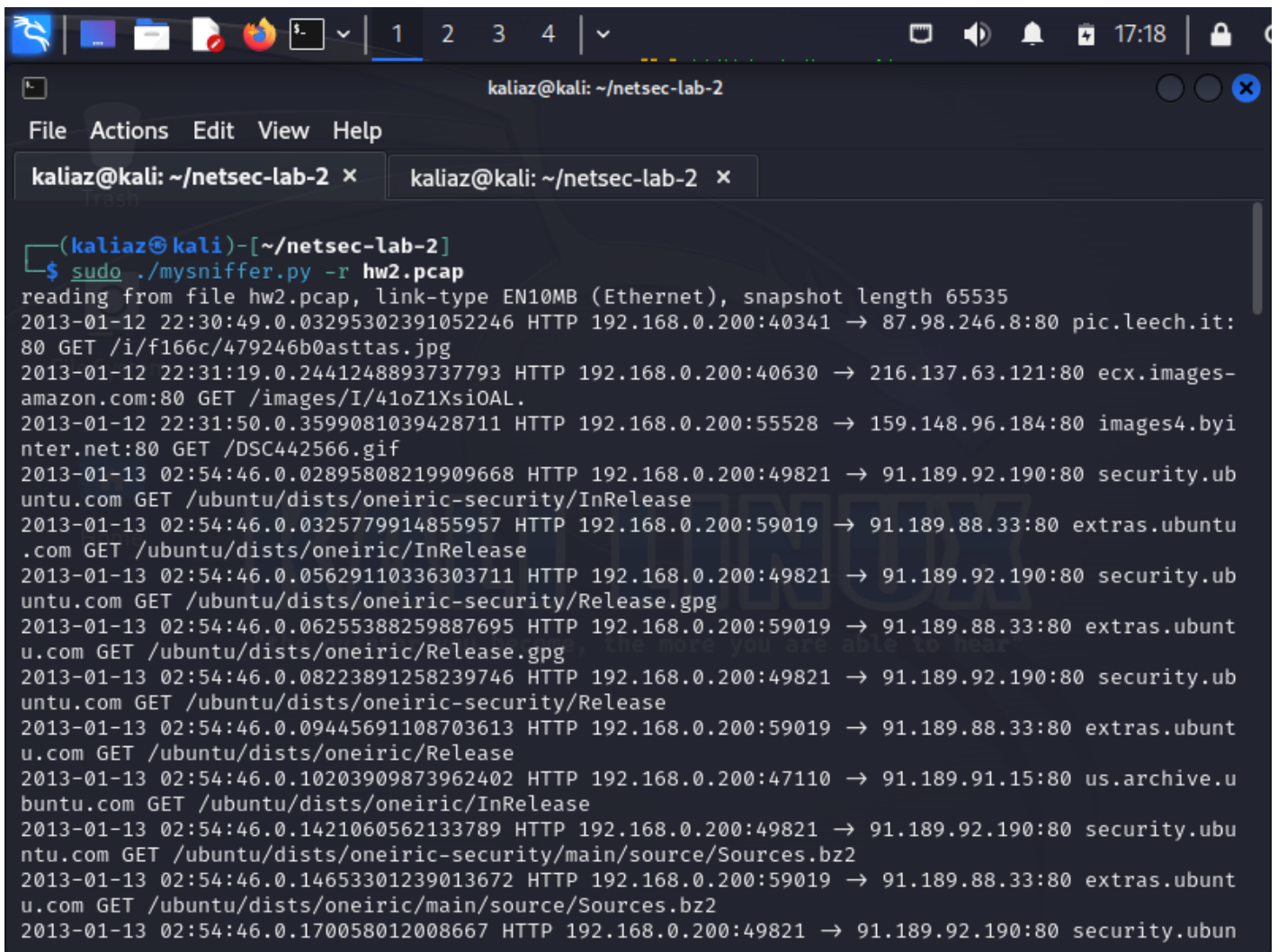
```
sudo ./mysniffer.py -i eth0
```



```
(kali@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
(kali@kali)~[~/netsec-lab-2]
$
```

[mysniffer.py] With Specified PCAP file

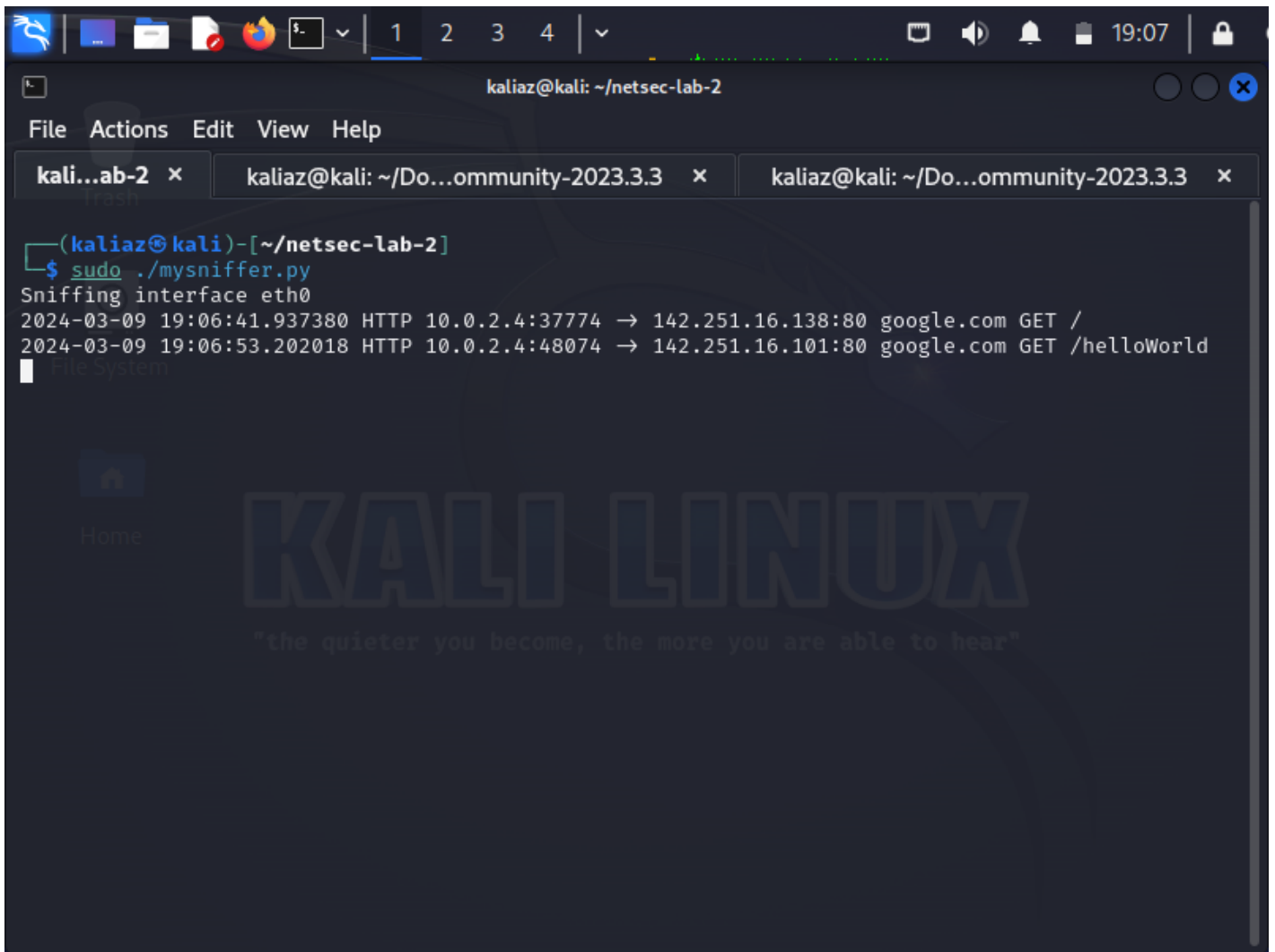
```
sudo ./mysniffer.py -r hw2.pcap
```



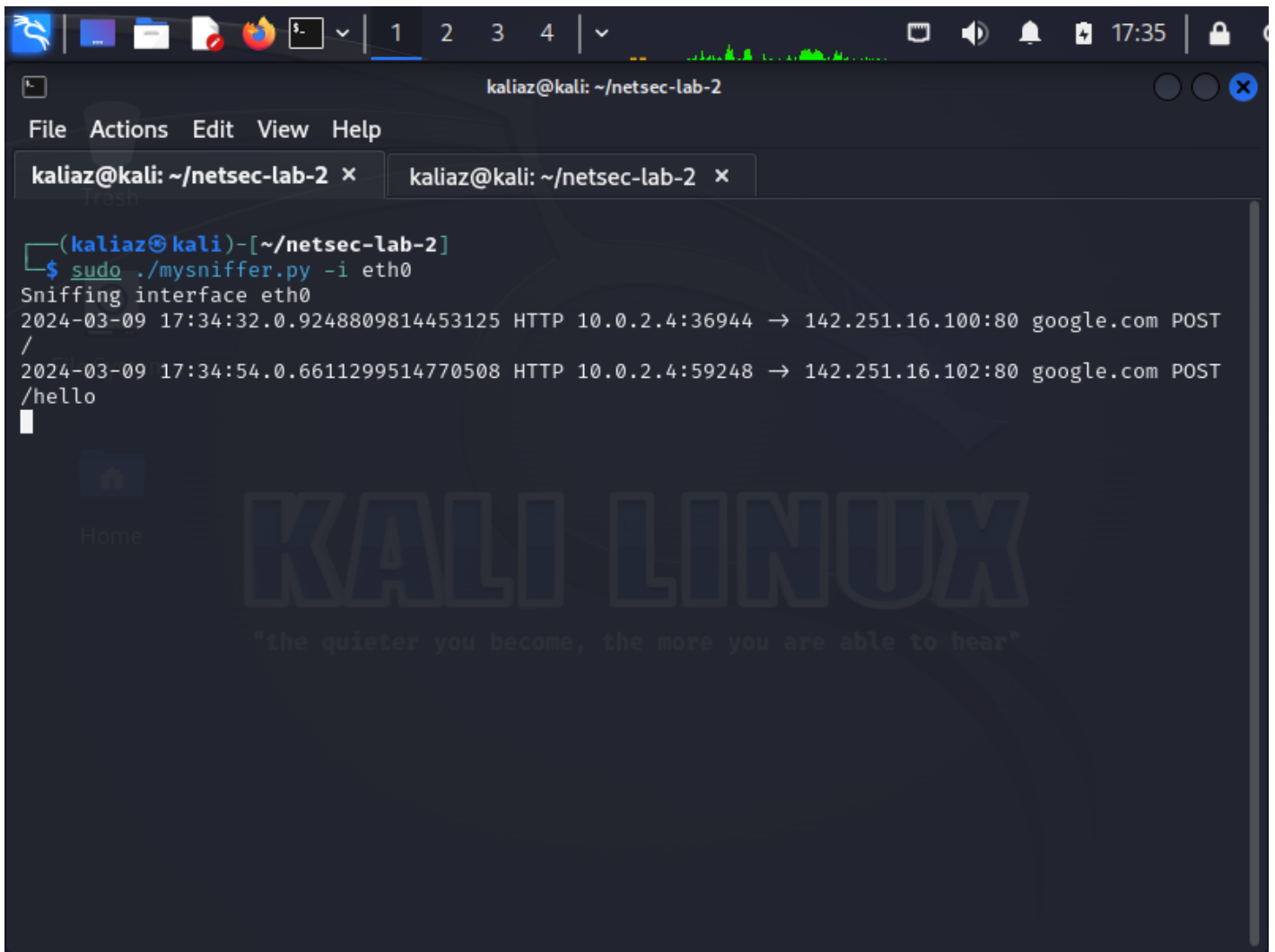
A terminal window titled 'kaliaz@kali: ~/netsec-lab-2' displays the output of the command `sudo ./mysniffer.py -r hw2.pcap`. The output shows a list of HTTP requests captured from a file named 'hw2.pcap', which is identified as an EN10MB (Ethernet) snapshot with a length of 65535. The requests are timestamped and include details such as the source IP, destination IP, and the requested URL. The requests are as follows:

Timestamp	Source IP	Destination IP	Request
2013-01-12 22:30:49.0.03295302391052246	192.168.0.200	87.98.246.8:80	pic.leech.it:80 GET /i/f166c/479246b0asttas.jpg
2013-01-12 22:31:19.0.2441248893737793	192.168.0.200	216.137.63.121:80	ecx.images-amazon.com:80 GET /images/I/4loZ1XsioAL.
2013-01-12 22:31:50.0.3599081039428711	192.168.0.200	159.148.96.184:80	images4.byinter.net:80 GET /DSC442566.gif
2013-01-13 02:54:46.0.02895808219909668	192.168.0.200	91.189.92.190:80	security.ubuntu.com GET /ubuntu/dists/oneiric-security/InRelease
2013-01-13 02:54:46.0.0325779914855957	192.168.0.200	91.189.88.33:80	extras.ubuntu.com GET /ubuntu/dists/oneiric/InRelease
2013-01-13 02:54:46.0.05629110336303711	192.168.0.200	91.189.92.190:80	security.ubuntu.com GET /ubuntu/dists/oneiric-security/Release.gpg
2013-01-13 02:54:46.0.06255388259887695	192.168.0.200	91.189.88.33:80	extras.ubuntu.com GET /ubuntu/dists/oneiric/Release.gpg
2013-01-13 02:54:46.0.08223891258239746	192.168.0.200	91.189.92.190:80	security.ubuntu.com GET /ubuntu/dists/oneiric-security/Release
2013-01-13 02:54:46.0.09445691108703613	192.168.0.200	91.189.88.33:80	extras.ubuntu.com GET /ubuntu/dists/oneiric/Release
2013-01-13 02:54:46.0.10203909873962402	192.168.0.200	91.189.91.15:80	us.archive.ubuntu.com GET /ubuntu/dists/oneiric/InRelease
2013-01-13 02:54:46.0.1421060562133789	192.168.0.200	91.189.92.190:80	security.ubuntu.com GET /ubuntu/dists/oneiric-security/main/source/Sources.bz2
2013-01-13 02:54:46.0.14653301239013672	192.168.0.200	91.189.88.33:80	extras.ubuntu.com GET /ubuntu/dists/oneiric/main/source/Sources.bz2
2013-01-13 02:54:46.0.170058012008667	192.168.0.200	91.189.92.190:80	security.ubuntu.com

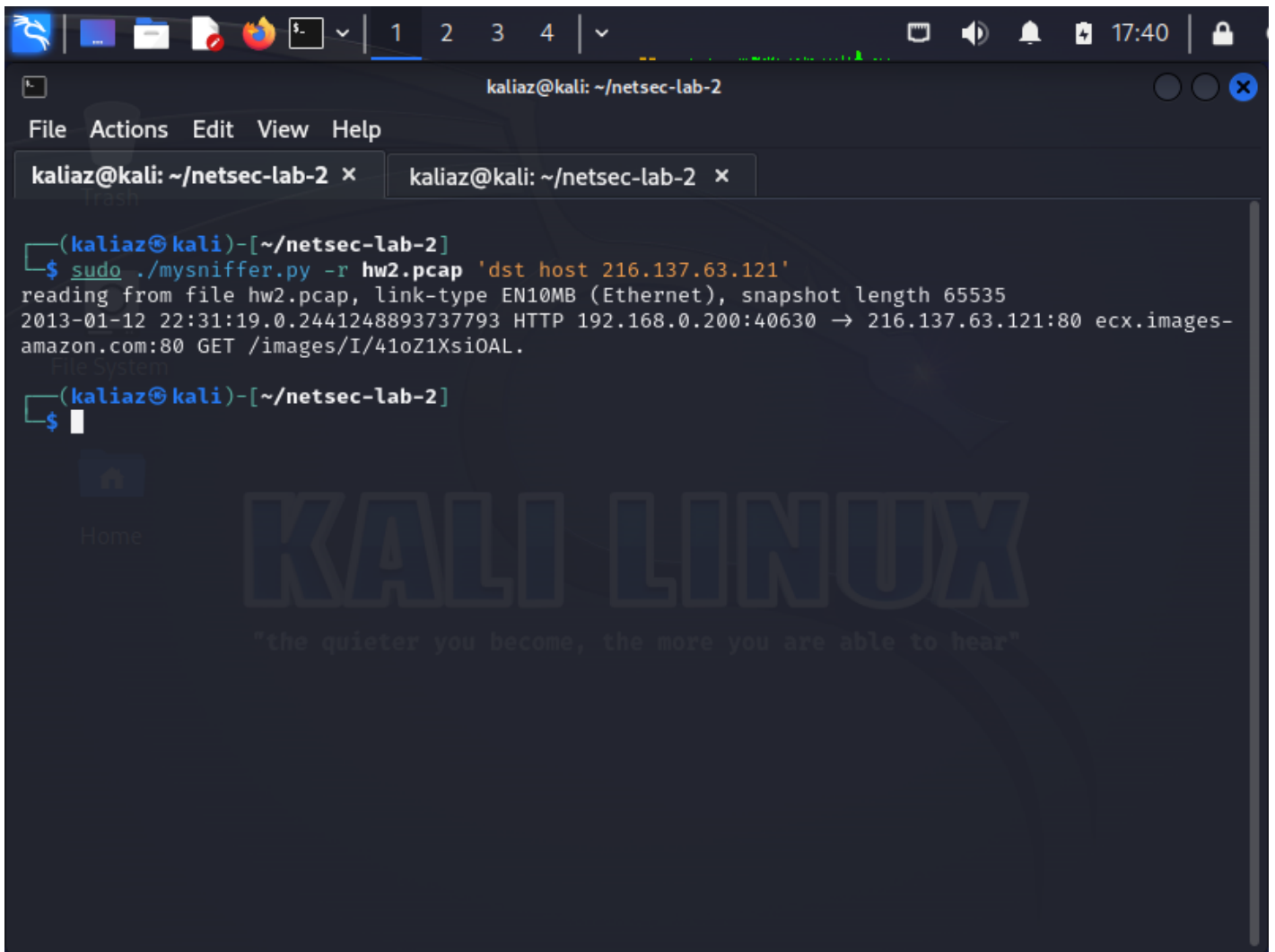
[mysniffer.py] With Default Interface (No interface and No pcap file provided)



[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
```

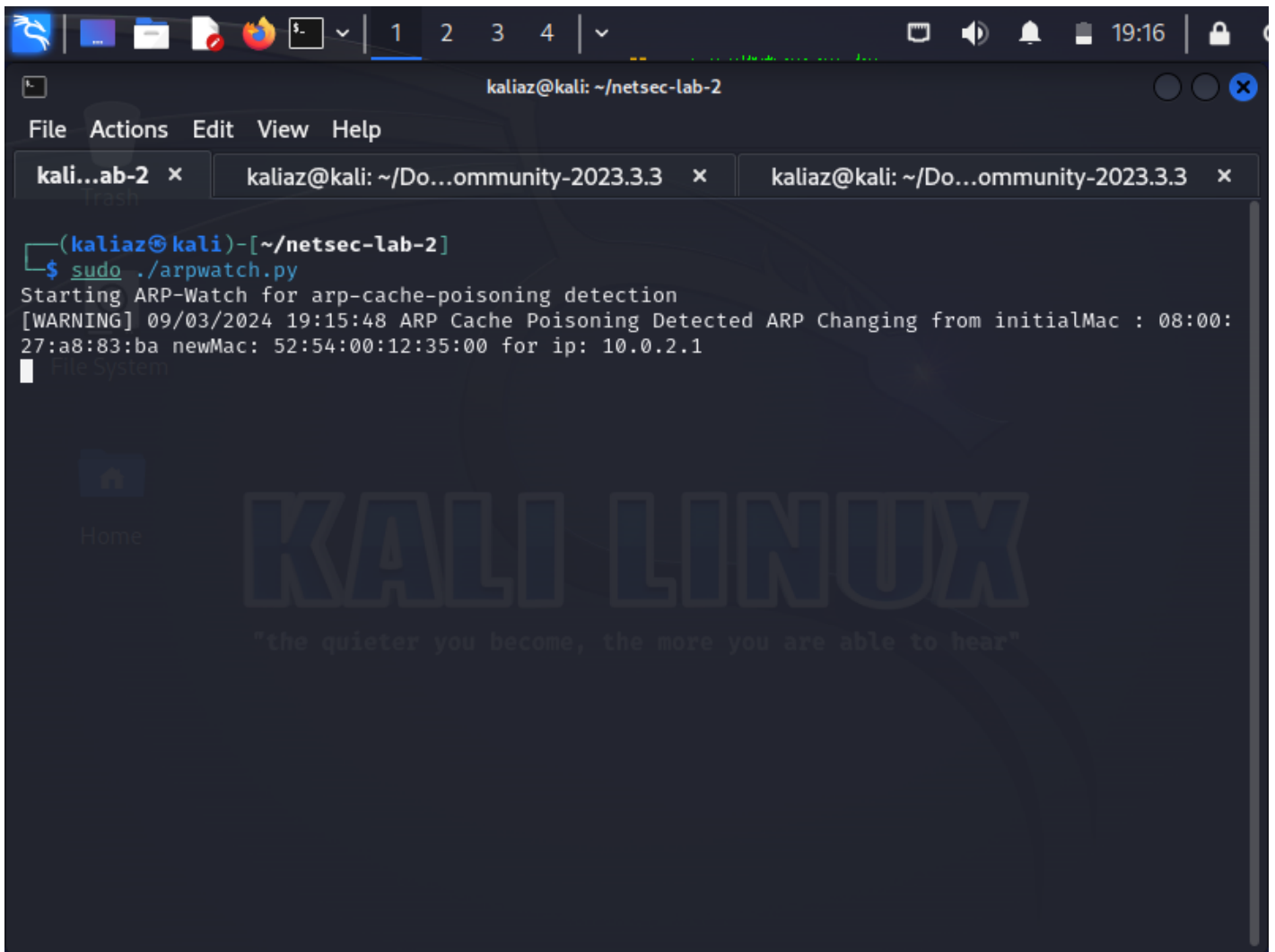
Arpwatch

This is a simple arp cache poisoning detector written to work on kali linux and in accordance with lab-2 for CSE 508 : Network Security

[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

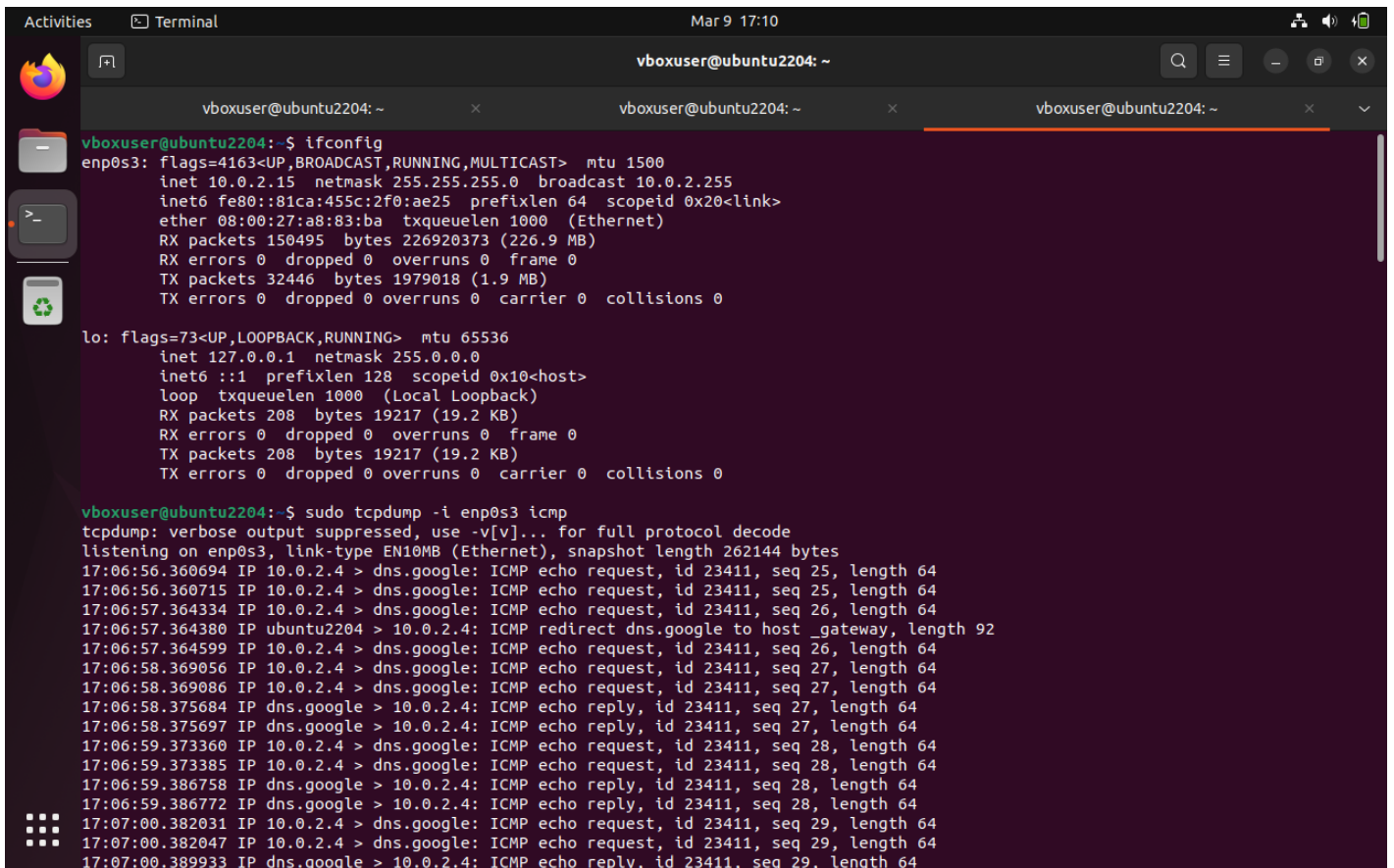
Victim IP Address : 10.0.2.4

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
```

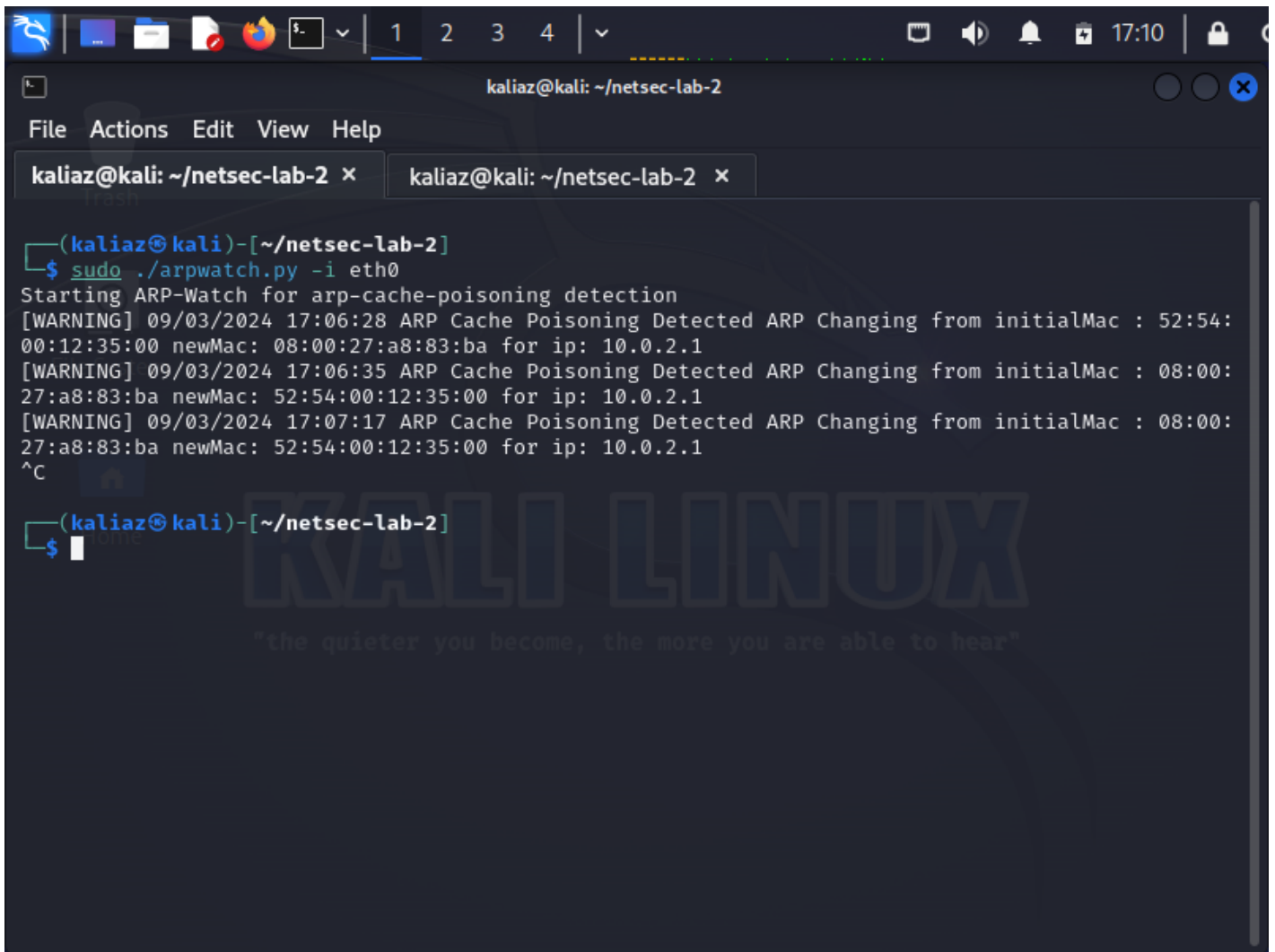
```
vboxuser@ubuntu2204: ~  
vboxuser@ubuntu2204:~$ ifconfig  
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255  
    inet6 fe80::81ca:455c:2f0:ae25 prefixlen 64 scopeid 0x20<link>  
    ether 08:00:27:a8:83:ba txqueuelen 1000 (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 arpwatc script. One for demo purposes

Victim Terminal 1

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

The below image indicates that the arpwatc.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`, which starts ARP-Watch for arp-cache-poisoning detection. It displays three warning messages indicating ARP cache poisoning detected and the ARP changing from an initial MAC to a new MAC for IP 10.0.2.1. The session ends with a Ctrl+C (^C) and returns to the shell prompt. A large 'KALI LINUX' watermark is visible in the background.

```
(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
```

```
kaliaz@kali: ~/netsec-lab-2
File Actions Edit View Help
kaliaz@kali: ~/netsec-lab-2 x kaliaz@kali: ~/netsec-lab-2 x

(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
```