# Lab 1 Using tools to sniff and spoof packets using Scapy

| Name: Vinay V | SRN: PES1UG24CS840 |
|---|---|
| Sub: CNS | Class: CSE “J sec” |

## Task 1.1: Sniffing packets

- **Task 1.1A: Sniff IP packets using Scapy**

  - On the Attackers terminal run ifconfig and copy the network interface and paste it on the Task1.1A.py file where ever its required
  - Open the wireshark and select the network interface same as attackers
  - And run python3 Task1.1A.py command.

```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.1A.py
SNIFFING PACKETS...
###[ Ethernet ]###
  dst       = 02:42:ff:c5:1d:51
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 53
     id        = 42650
```

  - Now go to Host A terminal and ping 8.8.8.8

```
hostA:PES1UG24CS840:Vinay:/
$>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=254 time=19.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=16.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=16.1 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=13.7 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=17.5 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=254 time=12.7 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=254 time=18.2 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=254 time=15.2 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=254 time=14.0 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=254 time=12.4 ms
64 bytes from 8.8.8.8: icmp_seq=11 ttl=254 time=12.8 ms
64 bytes from 8.8.8.8: icmp_seq=12 ttl=254 time=12.4 ms
64 bytes from 8.8.8.8: icmp_seq=13 ttl=254 time=33.2 ms
64 bytes from 8.8.8.8: icmp_seq=14 ttl=254 time=12.5 ms
64 bytes from 8.8.8.8: icmp_seq=15 ttl=254 time=12.4 ms
64 bytes from 8.8.8.8: icmp_seq=16 ttl=254 time=12.8 ms
64 bytes from 8.8.8.8: icmp_seq=17 ttl=254 time=21.3 ms
64 bytes from 8.8.8.8: icmp_seq=18 ttl=254 time=40.6 ms
^C
--- 8.8.8.8 ping statistics ---
18 packets transmitted, 18 received, 0% packet loss, tim
rtt min/avg/max/mdev = 12.353/17.437/40.614/7.466 ms
hostA:PES1UG24CS840:Vinay:/
$>
```

**Objective:** To capture and display network packets in real-time using Python's Scapy library from the attacker container in the SEED Lab setup, and analyze the captured packets using Wireshark.

**Packet Sniffing** is the process of intercepting and logging traffic that passes over a network.

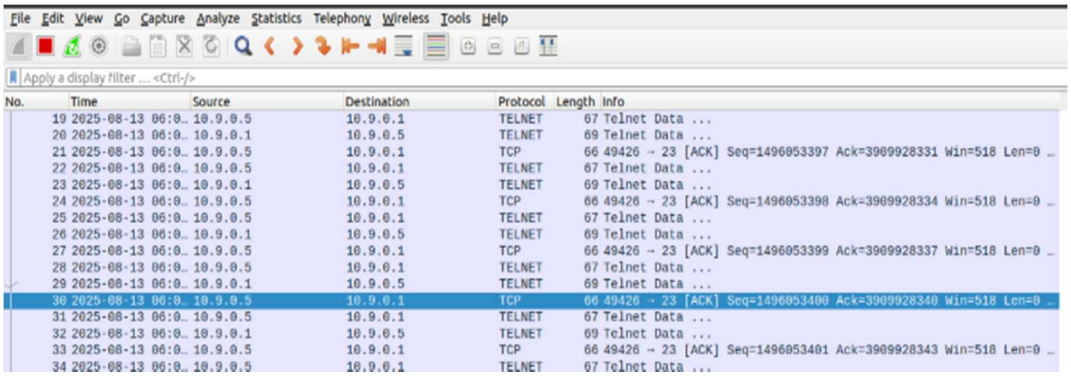
- In this task, we use **Scapy** (a Python packet manipulation tool) to sniff packets.
- The attacker container is connected to a **Docker bridge network** (interface name like br-xxxx).
- Wireshark is used to visualize and verify packet capture

**Key Observation:**

- **Sniffing Interface**: Must match the attacker's network interface.
- **Promiscuous Mode**: Captures all packets on the network segment.
- **Scapy sniff()** function:

  sniff(iface="interface_name", prn=callback_function)

- **iface:** network interface to listen on.
- **prn:** function to call for each captured packet.
- The Bridge interface connected to 10.0.9.1
- Even though **HostA** was pinging 8.8.8.8 (Google's DNS server), The **attacker container** was still able to capture it because: The attacker container, HostA, and HostB are connected to a Docker bridge network the bridge acts like a virtual switch all packets from any container pass through it.
- When hostA sends a packet 8.8.8.8 the packet still travels through the docker bridge and is forwarded out through the VM's interface to the internet since the attacker is listening on that bridge it can see the packet before it leaves for the internet



The Wireshark capture confirms that the attacker was able to observe Telnet TCP packets exchanged between HostA and Attacker over the `10.9.0.x` network. Since the attacker was sniffing on the bridge interface in promiscuous mode, any traffic on the same virtual network segment could be captured.

Code:

```
#!/usr/bin/python3
from scapy.all import *
print("SNIFFING PACKETS...");
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface = "br-****",prn=print_pkt)
```

pkt.show() prints a **detailed breakdown** of the packet, layer by layer (Ethernet, IP, TCP, etc.).

sniff() is a Scapy function used to capture packets.

**Parameters:**

- iface = "br-8794d4d6174d" → tells Scapy **which network interface** to listen on.
  In this case, it's the Docker bridge interface for your attacker container.
- prn = print_pkt → for every captured packet, call print_pkt(pkt) to display its details.

**What the code does overall:**

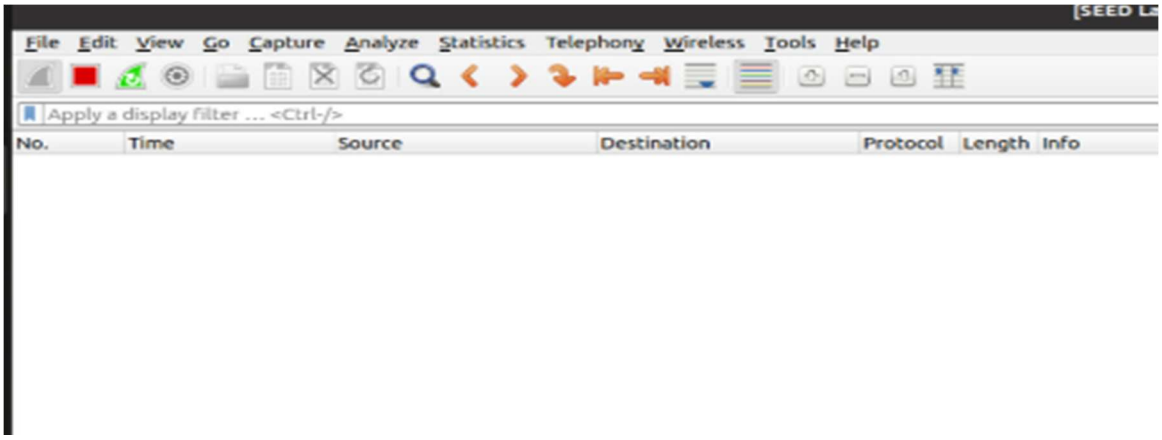
1. Starts listening on the attacker container's bridge interface.
2. Whenever a packet comes in or goes out on that interface, it calls print_pkt(pkt).
3. print_pkt prints a full protocol breakdown using pkt.show().

- **Running python3 Task1.1A.py without root privilege**

```
^Cseed-attacker:PES1UG24CS840:Vinay:/volumes
$>su seed
seed@VM:/volumes$ python3 Task1.1A.py
SNIFFING PACKETS...
Traceback (most recent call last):
  File "Task1.1A.py", line 6, in <module>
    pkt = sniff(iface = "br-8794d4d6174d",prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
seed@VM:/volumes$ su root
Password:
root@VM:/volumes#
```

PermissionError: [Errno 1] Operation not permitted
This happens because **packet sniffing** in Linux requires opening a **raw socket** (socket.SOCK_RAW), and raw sockets can only be created by the **root user** or a process with CAP_NET_RAW capability.

When running without root privileges, the program fails due to PermissionError because creating raw sockets for sniffing requires elevated privileges. This is a security measure to prevent unauthorized access to network traffic. Running as root resolves the issue.




**Wireshark Comparison**

- **If run as root** → packets will appear in the terminal output and in Wireshark (attacker's interface).
- **If run as seed (non-root)** → nothing is captured; Wireshark can still capture packets if run as non root, but Scapy will fail.

**Task 1.1 B: Applying Packet Filters to Capture certain types of packets for example: ICMP, TCP packet and Subnet**

In this task, the goal is to sniff only **ICMP packets** (such as ping requests/replies) on the network. ICMP packets are identified by the protocol field value icmp in the packet.
Instead of capturing *all* network packets, we apply a **BPF (Berkeley Packet Filter)** expression in Scapy to capture **only ICMP packets**.

Code:

```
#! / usr/bin/python3
from scapy.all import *
print ("SNIFFING PACKETS...");
def print_pkt(pkt):
    pkt.show()
pkt = sniff (iface = "br-8794d4d6174d",filter='icmp', prn=print_pkt)
```

- **iface** → specifies the attacker machine's network interface (bridge interface of the Docker network).
- **filter='icmp'** → applies the BPF filter so that only ICMP packets are captured (ignoring TCP, UDP, etc.).
- **prn=print_pkt** → calls the print_pkt function for each captured packet to display its details.

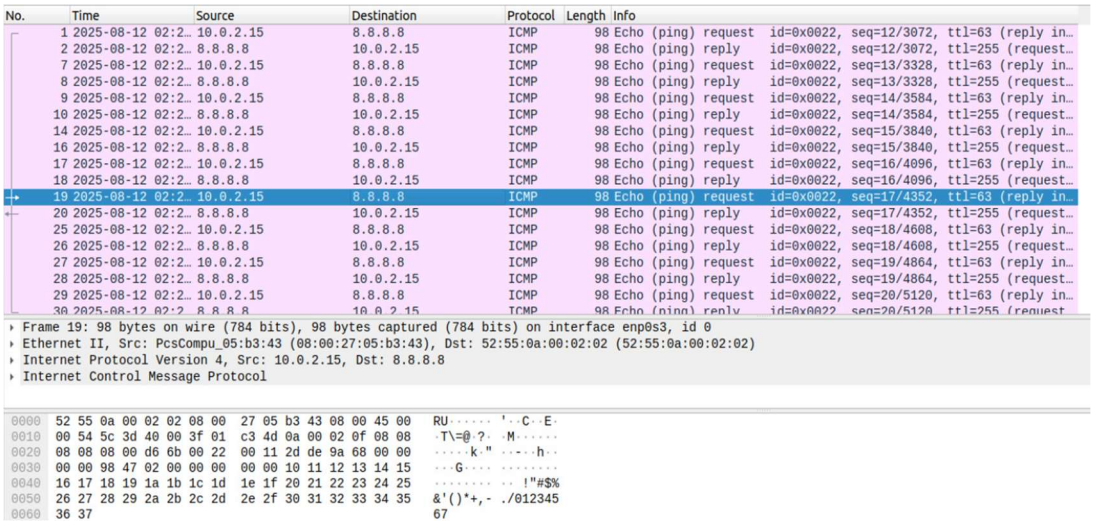
```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.1B-ICMP.py
SNIFFING PACKETS...
###[ Ethernet ]###
  dst         = 02:42:ff:c5:1d:51
  src         = 02:42:0a:09:00:05
  type        = IPv4
###[ IP ]###
     version    = 4
     ihl        = 5
     tos        = 0x0
     len        = 84
     id         = 26969
     flags      = DF
     frag       = 0
     ttl        = 64
     proto      = icmp
     chksum     = 0xb732
     src        = 10.9.0.5
     dst        = 8.8.8.8
     \options    \
###[ ICMP ]###
        type       = echo-request
        code       = 0
        chksum     = 0x3f1e
        id         = 0x23
        seq        = 0x1
###[ Raw ]###
```

- The attacker runs the Task1.1B-ICMP.py script with the filter='icmp'.

```
hostA:PES1UG24CS840:Vinay:/
$>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=254 time=55.7 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=46.5 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=56.7 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=61.8 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=61.5 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=254 time=59.0 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=254 time=62.4 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=254 time=61.0 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=254 time=54.5 ms
64 bytes from 8.8.8.8: icmp_seq=11 ttl=254 time=79.8 ms
64 bytes from 8.8.8.8: icmp_seq=12 ttl=254 time=54.0 ms
64 bytes from 8.8.8.8: icmp_seq=13 ttl=254 time=55.0 ms
64 bytes from 8.8.8.8: icmp_seq=15 ttl=254 time=57.9 ms
64 bytes from 8.8.8.8: icmp_seq=16 ttl=254 time=61.5 ms
64 bytes from 8.8.8.8: icmp_seq=17 ttl=254 time=57.6 ms
64 bytes from 8.8.8.8: icmp_seq=18 ttl=254 time=68.6 ms
64 bytes from 8.8.8.8: icmp_seq=19 ttl=254 time=83.1 ms
64 bytes from 8.8.8.8: icmp_seq=20 ttl=254 time=63.3 ms
64 bytes from 8.8.8.8: icmp_seq=21 ttl=254 time=61.4 ms
```

- On another machine in the same network (Host A), a **ping** command is run to 8.8.8.8.
- The attacker's script captures these ICMP packets because they pass the filter.

**Wireshark:**




- In the attacker's terminal output, each captured packet contains:

- **Ethernet header** (src MAC, dst MAC, type)
- **IP header** (src IP, dst IP, TTL, protocol)
- **ICMP header** (type, code, checksum)

- The **source IP** (10.9.0.5) and **destination IP** (8.8.8.8) match the ping traffic from Host A.
- Only ICMP packets are displayed; no TCP or UDP packets appear.

**Step 2: Capture any TCP packet that comes from a particular IP and with a destination port number 23**

In this task, the attacker machine uses Scapy to sniff only TCP packets that match two conditions:

1. **Source IP address** – A specific machine in the network (e.g., 10.9.0.5).
2. **Destination port number** – 23, which is used for the Telnet service.

**Code:**

```
#!/usr/bin/python3
from scapy.all import *
print ("SNIFFING PACKETS...")
def print_pkt(pkt):
    pkt.show()
pkt = sniff (iface = "br-8794d4d6174d",filter='tcp and src host 10.9.0.5 and dst port 23', prn=print_pkt)
pkt = sniff (iface = "br-8794d4d6174d",filter='
```

- Source IP is 10.9.0.5 (Host A),
- Destination IP is 10.9.0.1 (Telnet server),
- Destination Port is 23 (Telnet),
- Protocol is **TCP**.

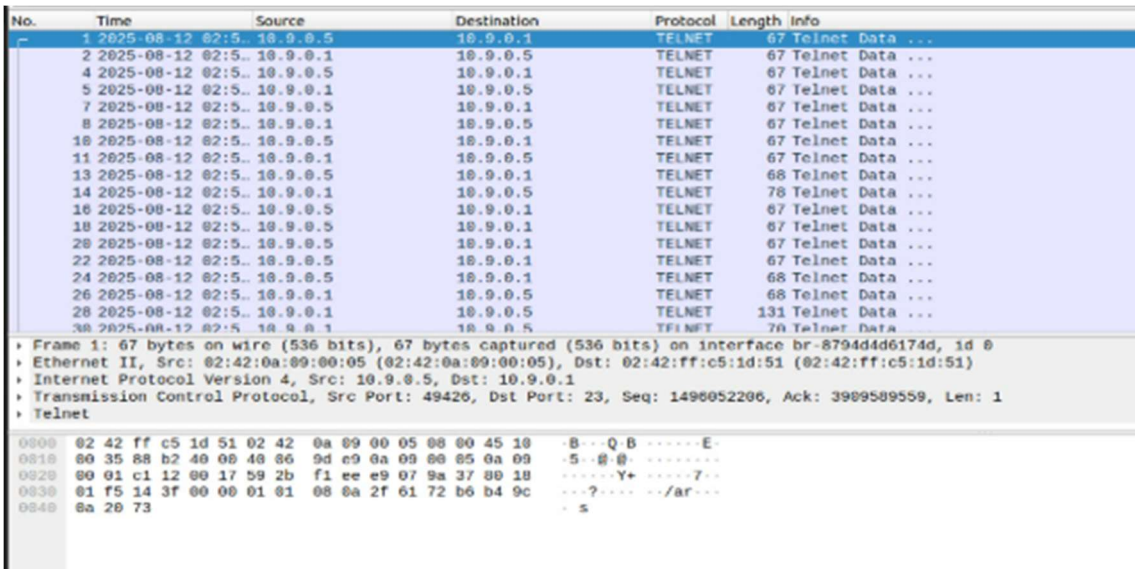
```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.1B-TCP.py
SNIFFING PACKETS...
###[ Ethernet ]###
  dst       = 02:42:ff:c5:1d:51
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 34820
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x9e90
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ TCP ]###
        sport     = 49426
```

- The attacker's Python script specifies the network interface (iface) and filter conditions (tcp and src host 10.9.0.5 and dst port 23).

```
[08/12/25]seed@VM:~$ export PS1="hostA:PES1UG24CS840:Vinay:\w\n\$>"
hostA:PES1UG24CS840:Vinay:~
$>telnet 10.9.0.q
telnet: could not resolve 10.9.0.q/telnet: Name or service not known
hostA:PES1UG24CS840:Vinay:~
$>telnet 10.9.0.1
Trying 10.9.0.1...
Connected to 10.9.0.1.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

0 updates can be installed immediately.
0 of these updates are security updates.


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Tue Aug 12 02:48:52 EDT 2025 from www.SeedLabSQLInjection.com on pts/6
```

- From Host A, a Telnet connection is initiated to `10.9.0.1` on port 23.

## Wireshark:




- Wireshark, running on the same network interface, also captures the same Telnet packets.
- The highlighted packets in Wireshark confirm that TCP communication occurred on port 23 between the specified hosts.

## Step 3: Capture packets that come from or go to a particular subnet

The objective of this task is to **capture and analyze network packets that come from or go to a specific subnet** using Python's **Scapy** library and **Wireshark**.

**Code:**

```
#!/usr/bin/python3
from scapy.all import *
print("SNIFFING PACKETS...")
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface = "br-8794d4d6174d",filter='src net 192.168.254.0/24', prn=print_pkt)
```

- The script uses scapy.sniff() to capture packets on a specified network interface.
- iface specifies the attacker's interface (br-8794d4d6174d).
- The filter 'src net 192.168.254.0/24' ensures only packets related to the chosen subnet are captured.
- The print_pkt() function prints packet details in a readable format

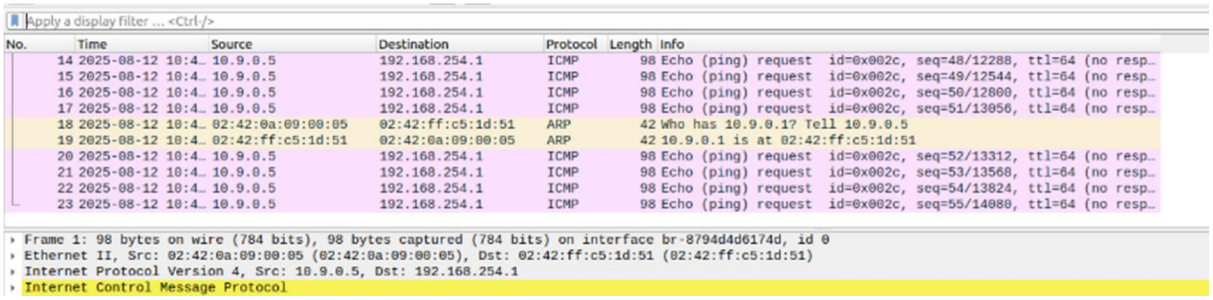
```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.1B-Subnet.py
SNIFFING PACKETS...
```

- When the script runs, it displays "SNIFFING PACKETS..." and starts listening for matching traffic.
- Once Host A sends the ping, the packets matching the subnet filter are displayed by the script.
- • Shows the ping request being sent to `192.168.254.1`.

```
root@060bacb69685:/# export PS1="hostA:PES1UG24CS840:Vina
hostA:PES1UG24CS840:Vinay:/
$>ping 192.168.254.1
PING 192.168.254.1 (192.168.254.1) 56(84) bytes of data.
```

- Shows the ping request being sent to 192.168.254.1.
- This generates ICMP packets visible in both the Python output and Wireshark.

## Wireshark:




- ICMP Echo Requests are visible from Host A (10.9.0.5) to 192.168.254.1.
- ARP requests are also seen:
  "Who has 10.9.0.1? Tell 10.9.0.5" (Address resolution request)

## Task 1.2: Packet Spoofing

The objective of this task is to **spoof ICMP echo request packets** with an arbitrary source IP address and send them to another machine on the same network (or to any reachable host).
This is done to demonstrate:

- How attackers can send forged packets pretending to be from another IP.
- How the receiver responds to spoofed packets.
- How such spoofed packets can be detected using Wireshark.

➢ Code:
➢ Task1.2A.py

```
#!/usr/bin/python3
from scapy.all import *
print ("SENDING SPOOFED ICMP PACKET...")
IPLayer = IP()
IPLayer.src="10.9.0.1"
IPLayer.dst="10.9.0.5"
ICMPpkt = ICMP()
pkt = IPLayer/ICMPpkt
pkt. show ()
send(pkt,verbose=0)
```

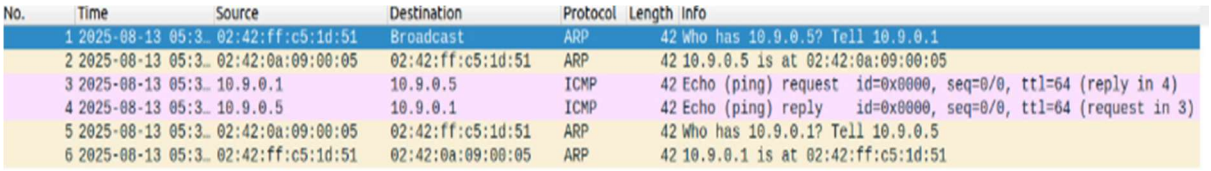
- **Create IP Layer** → IP() sets protocol details.

➢ src="10.9.0.1" → Fake source IP (spoofed).
➢ dst="10.9.0.5" → Target machine's IP.

- **Create ICMP Layer** → ICMP() defaults to type=8 (echo request).
- **Combine Layers** → pkt = IPLayer/ICMPpkt stacks IP + ICMP.
- **Display Packet** → pkt.show() prints all header fields.

- **Send Packet** → send(pkt, verbose=0) transmits the forged packet to the network.

```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.2A.py
SENDING SPOOFED ICMP PACKET...
###[ IP ]###
   version    = 4
   ihl        = None
   tos        = 0x0
   len        = None
   id         = 1
   flags      =
   frag       = 0
   ttl        = 64
   proto      = icmp
   chksum     = None
   src        = 10.9.0.1
   dst        = 10.9.0.5
   \options   \
###[ ICMP ]###
      type       = echo-request
      code       = 0
      chksum     = None
      id         = 0x0
      seq        = 0x0
```



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 2025-08-13 05:3… | 02:42:ff:c5:1d:51 | Broadcast | ARP | 42 | Who has 10.9.0.5? Tell 10.9.0.1 |
| 2 | 2025-08-13 05:3… | 02:42:0a:09:00:05 | 02:42:ff:c5:1d:51 | ARP | 42 | 10.9.0.5 is at 02:42:0a:09:00:05 |
| 3 | 2025-08-13 05:3… | 10.9.0.1 | 10.9.0.5 | ICMP | 42 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 4) |
| 4 | 2025-08-13 05:3… | 10.9.0.5 | 10.9.0.1 | ICMP | 42 | Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 3) |
| 5 | 2025-08-13 05:3… | 02:42:0a:09:00:05 | 02:42:ff:c5:1d:51 | ARP | 42 | Who has 10.9.0.1? Tell 10.9.0.5 |
| 6 | 2025-08-13 05:3… | 02:42:ff:c5:1d:51 | 02:42:0a:09:00:05 | ARP | 42 | 10.9.0.1 is at 02:42:ff:c5:1d:51 |

- **Attacker Terminal:** Shows the spoofed ICMP packet details — IP version, TTL, source/destination IPs, and ICMP type = echo-request.

- **Wireshark:**

- Shows ARP requests resolving IP to MAC before sending the ICMP.
- Shows an **ICMP Echo Request** from 10.9.0.1 to 10.9.0.5.
- Shows an **ICMP Echo Reply** from 10.9.0.5 to 10.9.0.1.

**Task 1.2B**

```
#!/usr/bin/python3
from scapy.all import *
print ("SENDING SPOOFED ICMP PACKET...")
IPLayer = IP()
IPLayer.src="10.9.0.11"
IPLayer.dst="10.9.0.99"
ICMPpkt = ICMP()
pkt = IPLayer/ICMPpkt
pkt. show()
send(pkt,verbose=0)
```

```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.2B.py
SENDING SPOOFED ICMP PACKET...
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 10.9.0.11
  dst       = 10.9.0.99
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
```

- **Attacker Terminal:** Shows the spoofed packet with the new source and destination IPs.

- **Wireshark:**

- ARP request: "Who has 10.9.0.99? Tell 10.9.0.1" → indicates network trying to resolve target MAC.
- ICMP Echo Request from 10.9.0.11 to 10.9.0.99.
- No ICMP reply observed — likely because:
  - The target host (10.9.0.99) is not active, or
  - The packet was blocked by a firewall.

## 1.3 Traceroute:

The aim of this task is to implement a **basic traceroute tool** using Python's **Scapy** library to estimate the number of hops (routers) between the attacker VM and a target destination.

**Code:**

```
#!/usr/bin/python3
from scapy.all import *
'''Usage:  ./traceroute.py " hostname or ip address"'''
host=sys.argv[1]
print ("Traceroute "+ host)
ttl=1
while 1:
    IPLayer=IP ()
    IPLayer.dst=host
    IPLayer.ttl=ttl
    ICMPpkt=ICMP()
    pkt=IPLayer/ICMPpkt
    replypkt = sr1(pkt,verbose=0)
    if replypkt is None:
        break
    elif replypkt [ICMP].type==0:
        print(f"{ttl} hops away: ", replypkt [IP].src)
        print( "Done", replypkt [IP].src)
        break
    else:
        print (f"{ttl} hops away: ", replypkt [IP].src)
        ttl+=1
```

This code is a simple **traceroute tool** using Scapy.

- It takes a target IP/hostname as input.
- Starts with **TTL = 1** and sends an ICMP Echo Request.
- If the reply is from the **final destination** (ICMP type 0), it prints hop count and stops.
- If the reply is from an **intermediate router**, it prints that hop's IP and increases TTL to check the next hop.
- Continues until the destination is reached or no reply is received

```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.3.py 10.9.0.6
Traceroute 10.9.0.6
1 hops away:  10.9.0.6
Done 10.9.0.6
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>
```

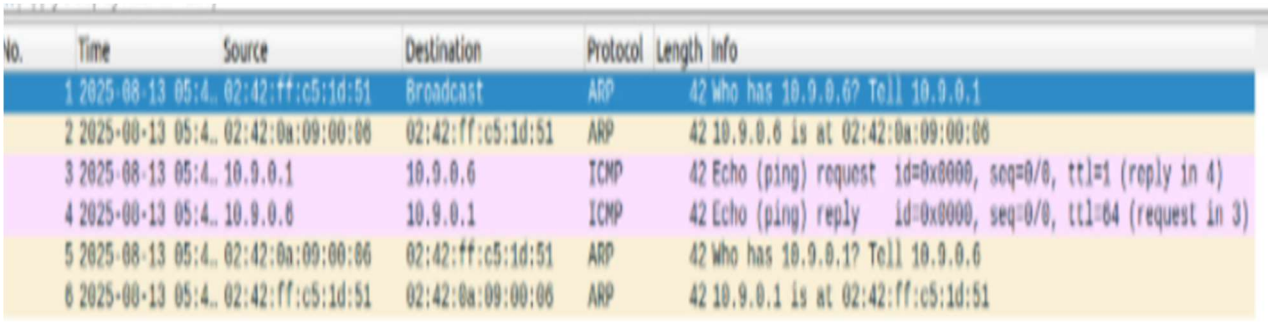
**Meaning:** The target (`10.9.0.6`) is only **1 hop away** — directly reachable without intermediate routers.

**Target Machine (10.9.0.6) ifconfig Output**:

- Confirms the machine's IP is indeed 10.9.0.6.

```
hostB:PES1UG24CS840:Vinay:/
$>ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 126  bytes 17781 (17.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- Ip add of hostB



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 2025-08-13 05:4… | 02:42:ff:c5:1d:51 | Broadcast | ARP | 42 | Who has 10.9.0.6? Tell 10.9.0.1 |
| 2 | 2025-08-13 05:4… | 02:42:0a:09:00:06 | 02:42:ff:c5:1d:51 | ARP | 42 | 10.9.0.6 is at 02:42:0a:09:00:06 |
| 3 | 2025-08-13 05:4… | 10.9.0.1 | 10.9.0.6 | ICMP | 42 | Echo (ping) request id=0x0000, seq=0/0, ttl=1 (reply in 4) |
| 4 | 2025-08-13 05:4… | 10.9.0.6 | 10.9.0.1 | ICMP | 42 | Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 3) |
| 5 | 2025-08-13 05:4… | 02:42:0a:09:00:06 | 02:42:ff:c5:1d:51 | ARP | 42 | Who has 10.9.0.1? Tell 10.9.0.6 |
| 6 | 2025-08-13 05:4… | 02:42:ff:c5:1d:51 | 02:42:0a:09:00:06 | ARP | 42 | 10.9.0.1 is at 02:42:ff:c5:1d:51 |

**Wireshark**

- Shows ARP requests for MAC address resolution before sending ICMP packets.
- Displays ICMP Echo Request from attacker to 10.9.0.6.
- Shows ICMP Echo Reply from 10.9.0.6 to the attacker.
- No intermediate “Time Exceeded” messages because it’s a single-hop connection.

# Task 1.4 Sniffing and then spoofing

The objective of this task is to demonstrate **sniffing and then spoofing ICMP packets**.
In this experiment:

- The victim (Host A) pings a **non-existent IP address** (`1.2.3.4`).
- The attacker sniffs these ICMP echo requests, then sends **fake ICMP echo replies** to make the victim believe the target is alive.

# Code:

```
#!/usr/bin/python3
from scapy.all import *
def spoof_pkt(pkt):
    newseq=0
    if ICMP in pkt:
        print("original packet.........")
        print("source IP :", pkt [IP].src)
        print("Destination IP :", pkt [IP]. dst)
        srcip = pkt [IP]. dst
        dstip = pkt[IP].src
        newihl = pkt [IP]. ihl
        newtype = 0
        newid = pkt [ICMP].id
        newseq = pkt [ICMP]. seq
        data = pkt [Raw]. load                    (function) ihl: Any
        IPLayer = IP (src=srcip, dst=dstip, ihl=newihl)
        ICMPpkt = ICMP (type=newtype, id=newid, seq=newseq)
        newpkt = IPLayer/ICMPpkt/data
        print ("spoofed packet.........")
        print ("Source IP:", newpkt [IP].src)
        print ("Destination IP:", newpkt [IP]. dst)
        send (newpkt, verbose=0)
pkt = sniff (iface="br-8794d4d6174d",filter='icmp and src host 10.9.0.5', prn=spoof_pkt)
```

This code **sniffs ICMP echo requests** from a victim and sends **spoofed ICMP echo replies** back.

- **Sniffing:** Listens for ICMP packets from 10.9.0.5.
- **Extracting:** Gets source/destination IPs, ICMP ID, sequence number, and payload from the sniffed packet.
- **Spoofing:** Swaps IPs, sets ICMP type to 0 (reply), keeps the same ID, sequence, and data.
- **Sending:** Transmits the forged packet so the victim thinks the ping target is alive, even if it's not.

```
seed-attacker:PES1UG24CS840:Vinay:/volumes
$>python3 Task1.4.py
original packet.........
source IP : 10.9.0.5
Destination IP : 1.2.3.4
spoofed packet........
Source IP: 1.2.3.4
Destination IP: 10.9.0.5
original packet.........
source IP : 10.9.0.5
Destination IP : 1.2.3.4
spoofed packet........
Source IP: 1.2.3.4
Destination IP: 10.9.0.5
original packet.........
source IP : 10.9.0.5
Destination IP : 1.2.3.4
spoofed packet........
Source IP: 1.2.3.4
Destination IP: 10.9.0.5
original packet.........
```

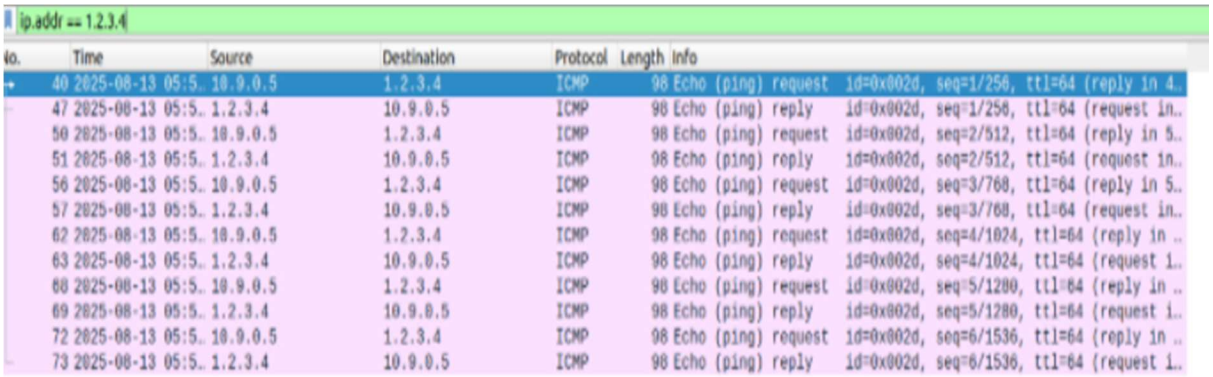
Shows logs for each sniffed and spoofed packet:

- **Original packet** → ICMP echo request from victim (10.9.0.5) to 1.2.3.4.
- **Spoofed packet** → ICMP echo reply from 1.2.3.4 to victim (10.9.0.5).

```
hostA:PES1UG24CS840:Vinay:/
$>ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=20.4 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=16.2 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=4.82 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=9.60 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=14.2 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=64 time=8.45 ms
^C
--- 1.2.3.4 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5009ms
rtt min/avg/max/mdev = 4.816/12.281/20.424/5.209 ms
hostA:PES1UG24CS840:Vinay:/
$>
```

- Command: ping 1.2.3.4
- Output: Receives ICMP replies with **0% packet loss**, indicating the IP is "alive".
- This is entirely due to the attacker's spoofed replies.

**Wireshark:**




- ICMP echo request packets from victim to 1.2.3.4.
- ICMP echo reply packets from 1.2.3.4 to victim, even though 1.2.3.4 doesn't exist.
- Confirms that replies are forged and injected by the attacker.