NETWORK SECURITY BASICS

CS44500 Computer Security

Outline

- IP Address and Network Interface
- TCP/IP Protocols
- Packet Sniffing
- Packet Spoofing
- Programming using Scapy
- Lab environment and containers

IP ADDRESS

IP Address: Original Scheme (Classful Addressing)

```
                   Network ID
Class A                        |<--       Host ID      -->|
  0.  0.  0.  0 = 00000000.00000000.00000000.00000000
127.255.255.255 = 01111111.11111111.11111111.11111111

Class B                                  |<-- Host ID -->|
128.  0.  0.  0 = 10000000.00000000.00000000.00000000
191.255.255.255 = 10111111.11111111.11111111.11111111

Class C                                          |HostID|
192.  0.  0.  0 = 11000000.00000000.00000000.00000000
223.255.255.255 = 11011111.11111111.11111111.11111111

Class D                   |<--   Address Range       -->|
224.  0.  0.  0 = 11100000.00000000.00000000.00000000
239.255.255.255 = 11101111.11111111.11111111.11111111

Class E                   |<--   Address Range       -->|
240.  0.  0.  0 = 11110000.00000000.00000000.00000000
255.255.255.255 = 11111111.11111111.11111111.11111111
```

Class	Range (First Octet)	Subnet Mask	Usage
A	0 – 127	255.0.0.0 (/8)	Large networks
B	128 – 191	255.255.0.0 (/16)	Medium-sized networks
C	192 – 223	255.255.255.0 (/24)	Small networks
D	224 – 239	N/A	Multicasting
E	240 – 255	N/A	Experimental

Inefficient IP Usage: Classful addressing is often inefficient because it limits networks to predefined sizes, which can lead to wasted IP addresses

CIDR Scheme
(Classless Inter-Domain Routing)

192.168.60.5/**24**

Indicate the first 24 bits are network ID

CIDR allows for more flexible allocation by disregarding fixed boundaries and supporting custom subnet masks

Question: What is the address range of the network **192.168.192.0/19 ?**

$\underbrace{\text{192.168.(110}}_{\text{Network ID = 19 bits}}$ 00000).00000000 to 192.168.(110 11111).11111111, i.e., it is 192.168.192.0 to 192.168.223.255.

Special IP Addresses

- Private IP Addresses
  - 10.0.0.0/8
  - 172.16.0.0/12
  - 192.168.0.0/16

Reserved for use within private networks and are not routable on the public internet. They allow devices within a local network to communicate without using globally unique IP addresses.

- Loopback Address
  - 127.0.0.0/8
  - Commonly used: 127.0.0.1

used by a host to send network traffic back to itself, often for testing purposes

List IP Address on Network Interface

```
$ ip -br address
lo                UNKNOWN        127.0.0.1/8 ::1/128
enp0s3            UP             10.0.5.5/24 fe80::bed8:53e2:5192:f265/64
docker0           DOWN           172.17.0.1/16 fe80::42:13ff:fee7:90d6/64
```

Interface name | Status | ip address(es) | IPv4 and IPv6

- *ip*: The main command used for network configuration tasks (showing addresses, configuring routes, etc.).
- *-br* (or --brief): Requests a concise, tabular output that makes it easier to quickly scan.
- *address*: Specifies that you want to display IP addresses and related information for each interface.

Manually Assign IP Address

```
$ sudo ip addr add 192.168.60.6/24 dev enp0s3
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
 group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_f
ast state UP group default qlen 1000
    link/ether 08:00:27:84:5e:b9 brd ff:ff:ff:ff:ff:ff
    inet 192.168.60.6/24 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::3fc4:1dac:bbbb:948/64 scope link
       valid_lft forever preferred_lft forever
```

Automatically Assign IP Address

- DHCP: Dynamic Host Configuration Protocol

The **Dynamic Host Configuration Protocol (DHCP)** is a network management protocol used to **automatically assign IP addresses** and other network configuration settings to devices on a network.

DHCP simplifies network management by enabling devices to connect to the network without requiring manual IP configuration.

Get IP Addresses for Host Names: DNS

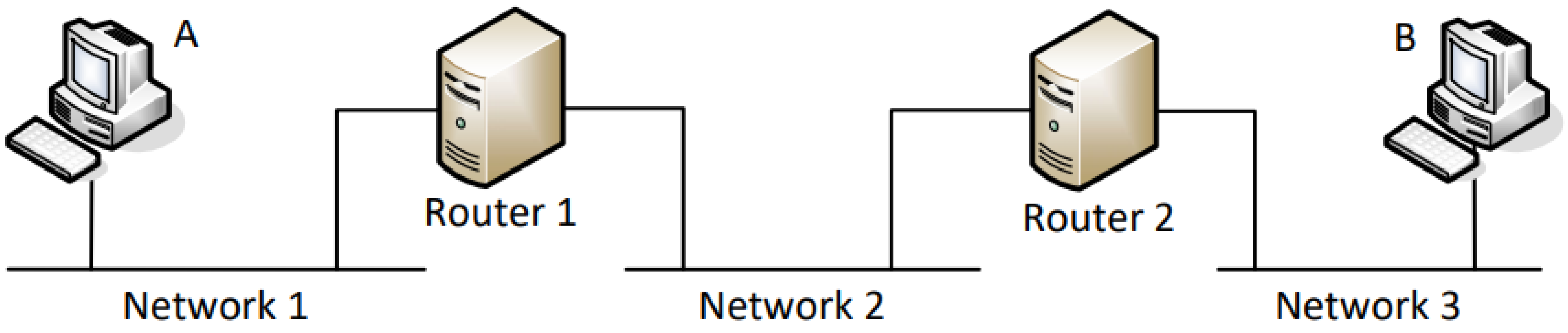
```
seed@VM:~$ dig www.example.com
```

(Domain Information Groper)

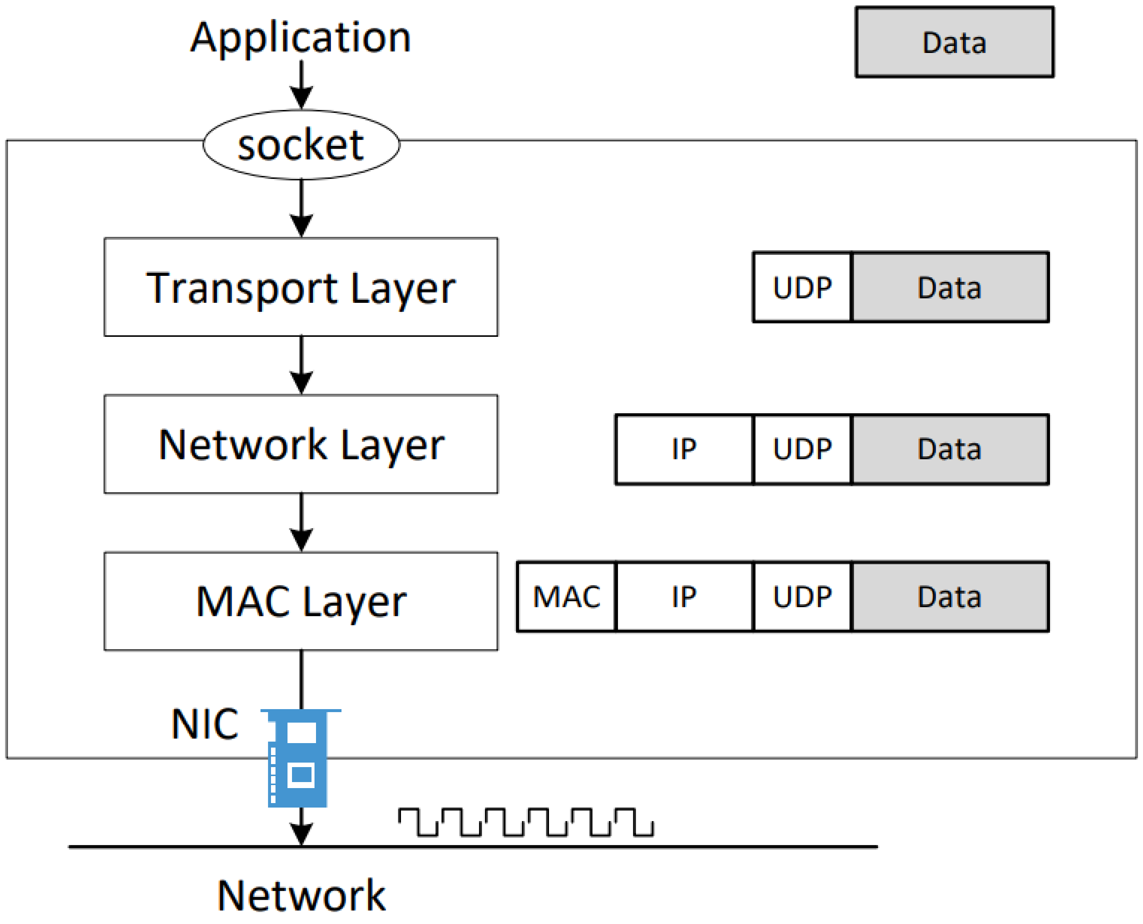
```
; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 18093
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

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

;; ANSWER SECTION:
www.example.com.        57405   IN      A       93.184.216.34
```

NETWORK STACK

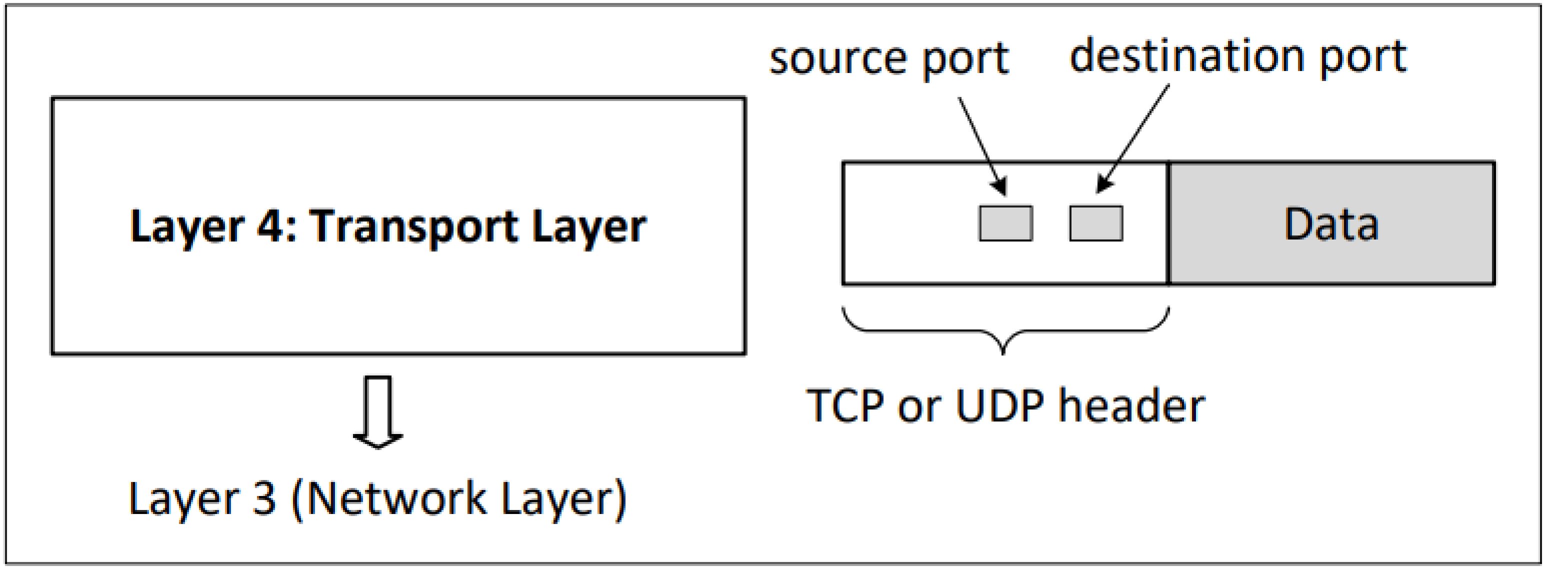
Packet Journey at High Level



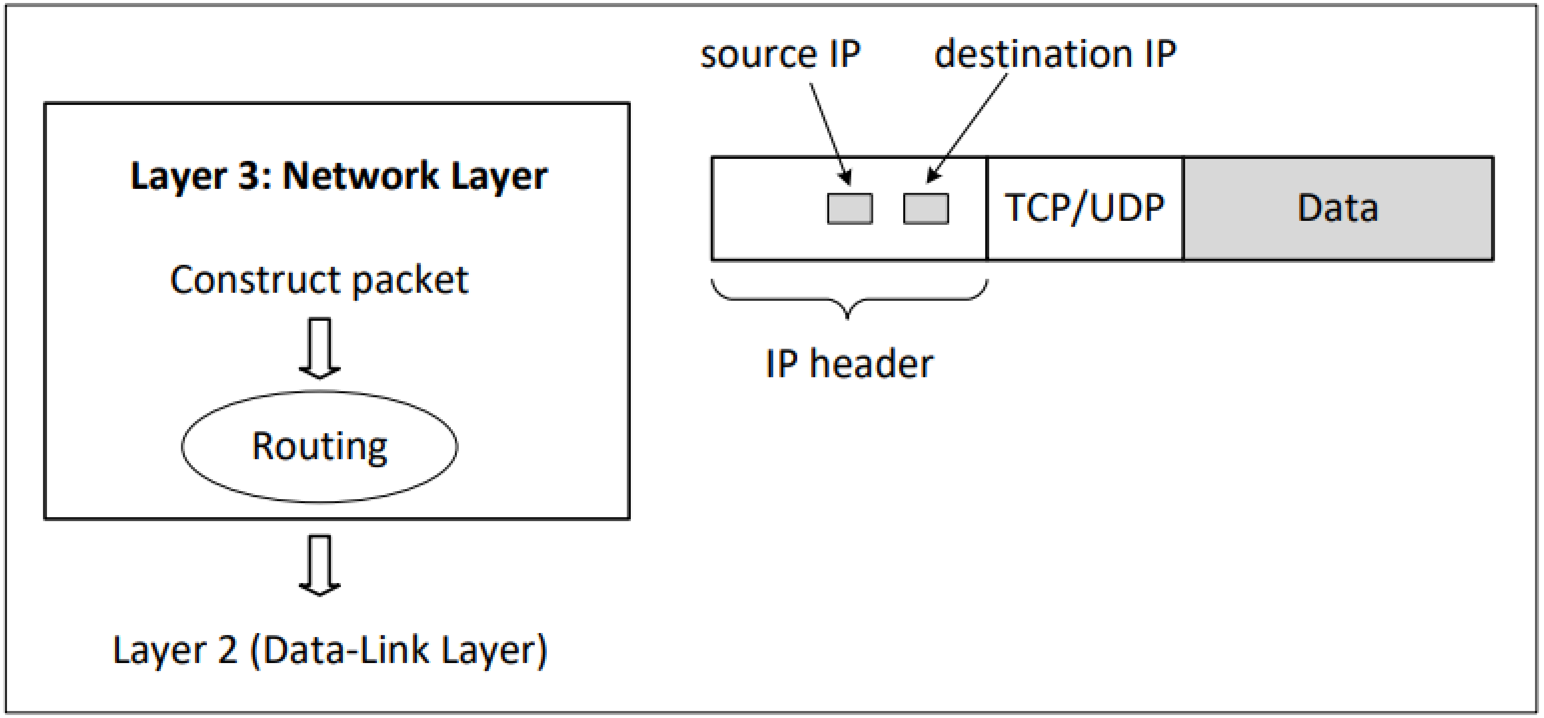
How Packets Are Constructed



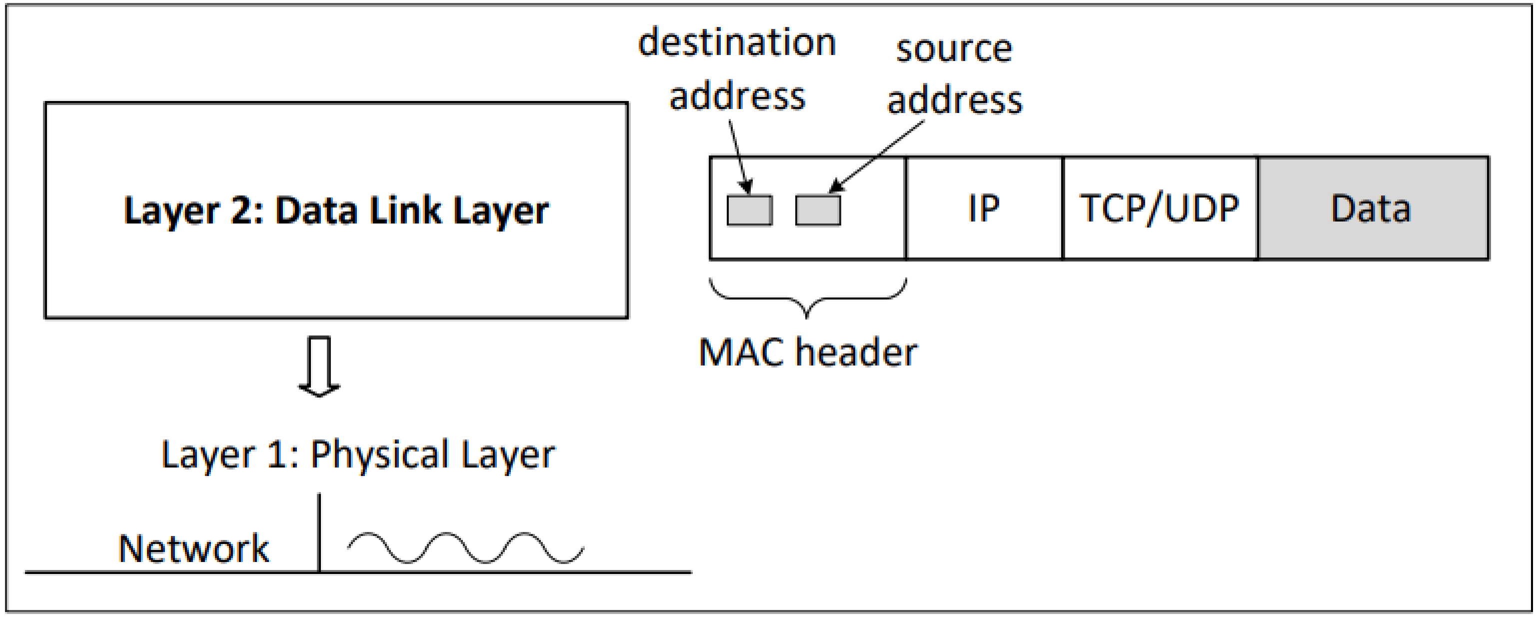
Layer 4: Transport Layer



Layer 3: Network Layer



Layer 2: Data Link Layer (MAC Layer)



Sending Packet in Python (1)

- UDP Client

```python
#!/usr/bin/python3

import socket

IP   = "127.0.0.1"
PORT = 9090
data = b'Hello, World!'

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(data, (IP, PORT))
```

Sending Packet in Python (1)

- Execution Results

```
$ nc -luv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Hello, World!
```

nc: Netcat: This is the command that invokes the Netcat program.

-l: Listen mode: The -l flag tells Netcat to run in listen mode. In this mode, Netcat will wait for incoming connections on the specified port

-u: UDP mode: The -u flag tells Netcat to use the UDP protocol instead of the default TCP.

-v: Verbose mode: The -v flag enables verbose mode, which provides additional information about the operation of Netcat.

Receiving Packets in Python

- UDP Server

indicating that the file should be executed using python3, otherwise you need to call run “python3 script.py”

IPv4

UPD

```
#!/usr/bin/python3

import socket

IP   = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP, PORT))# tells the OS that this socket will listen for incoming data

while True:
  data, (ip, port) = sock.recvfrom(1024)
  print("Sender: {} and Port: {}".format(ip, port))
  print("Received message: {}".format(data))
```

1024 is the maximum size (in bytes) of the data that can be received in one call.

UDP Server

```python
#!/usr/bin/python3

import socket

IP   = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP, PORT))

while True:
  data, (ip, port) = sock.recvfrom(1024)
  print("Sender: {} and Port: {}".format(ip, port))
  print("Received message: {}".format(data))
```

sends UDP packets

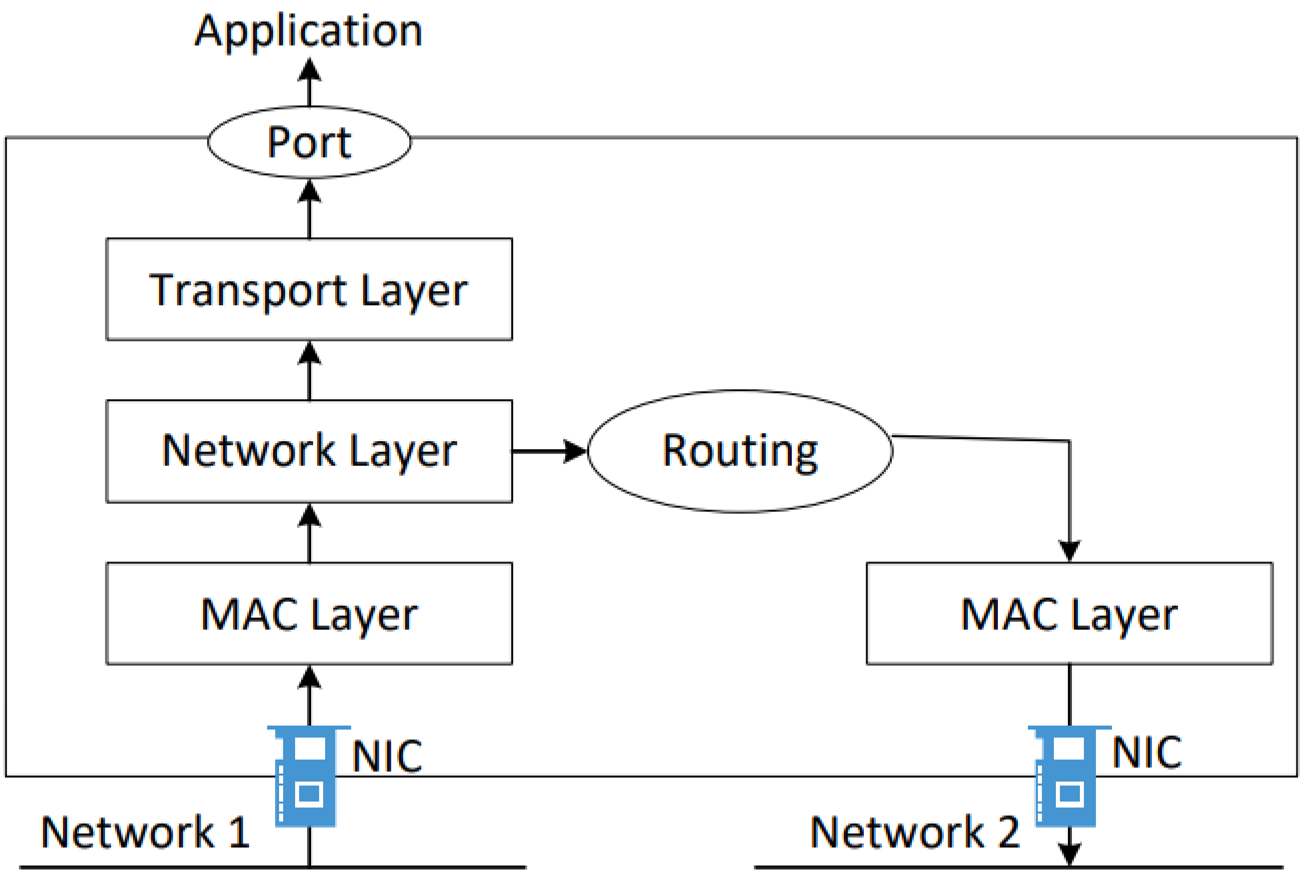
Terminal

```
seed@10.0.2.6:$ nc -u 10.0.2.7 9090
hello
hello again
```

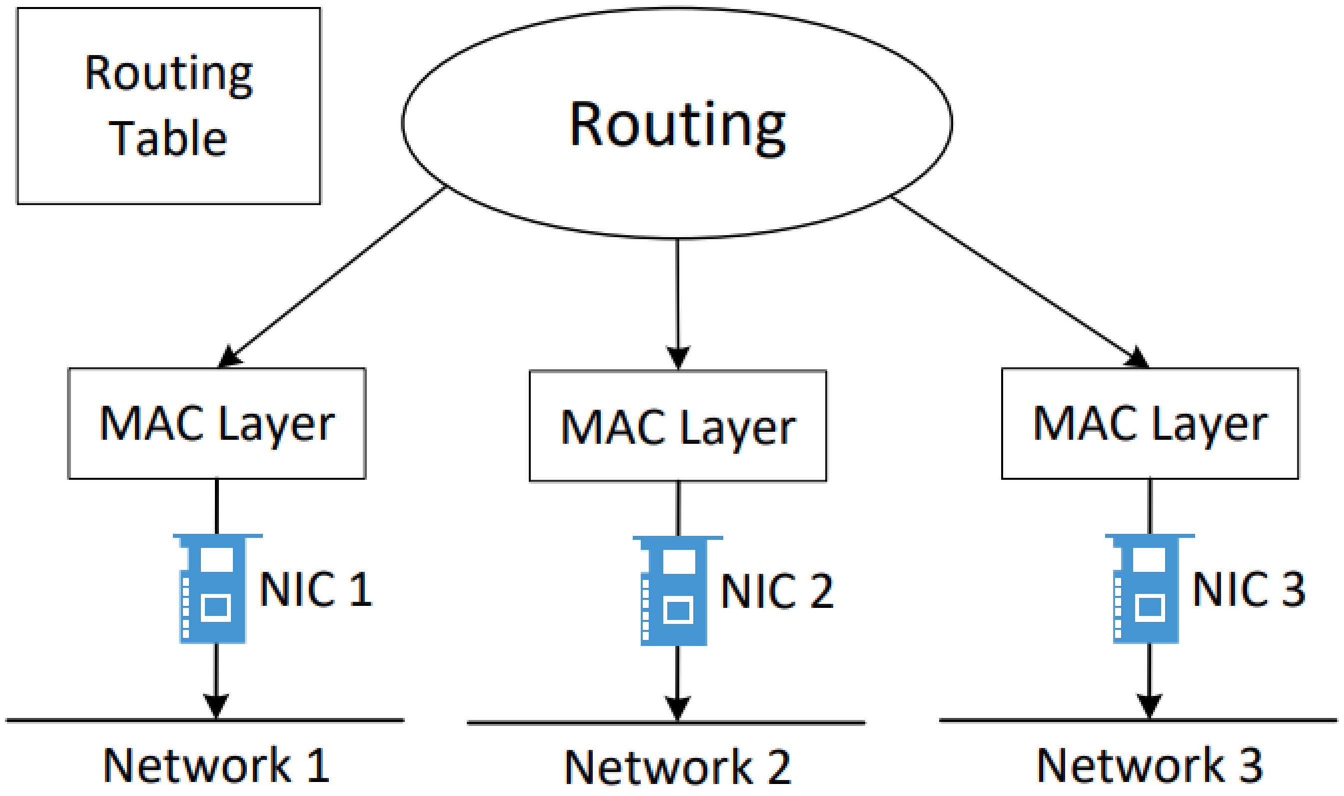
Terminal

```
Server(10.0.2.7):$ udp_server.py
Sender: 10.0.2.6 and Port: 49112
Received message: b'hello\n'
Sender: 10.0.2.6 and Port: 49112
Received message: b'hello again\n'
```

How Packets Are Received



Routing



Routing
Table
Routing
MAC Layer
MAC Layer
MAC Layer
NIC 1
NIC 2
NIC 3
Network 1
Network 2
Network 3

The “ip route” Command

View the Routing Table

```
# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.11
192.168.60.0/24 dev eth1 proto kernel scope link src 192.168.60.11

# ip route get 10.9.0.1
10.9.0.1 dev eth0 src 10.9.0.11 uid 0

# ip route get 192.168.60.5
192.168.60.5 dev eth1 src 192.168.60.11 uid 0

# ip route get 1.2.3.4
1.2.3.4 via 10.9.0.1 dev eth0 src 10.9.0.11 uid 0
```

Next hop IP

Interface name

default route, used for any destination not specifically listed in the routing table

Destination network

route was automatically added by the kernel

source address

destination IP

Packet Sending Tools

- Using netcat

```
$ nc <ip> <port>         ← send out TCP packet
$ nc -u <ip> <port>      ← send out UDP packet
```

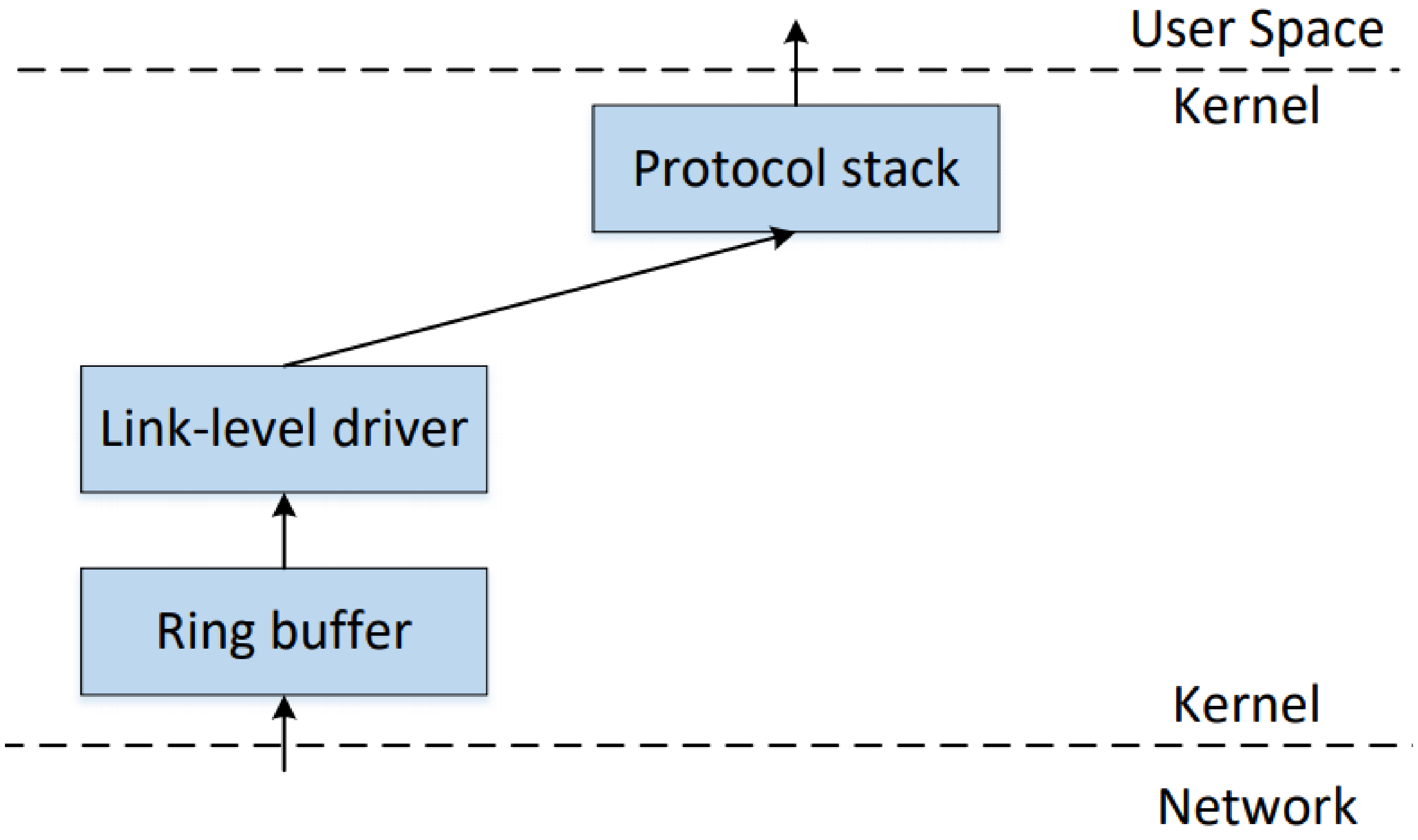
- Bash: /dev/tcp or /dev/udp pseudo device

```
$ echo "data" > /dev/udp/<ip>/<port>
$ echo "data" > /dev/tcp/<ip>/<port>
```

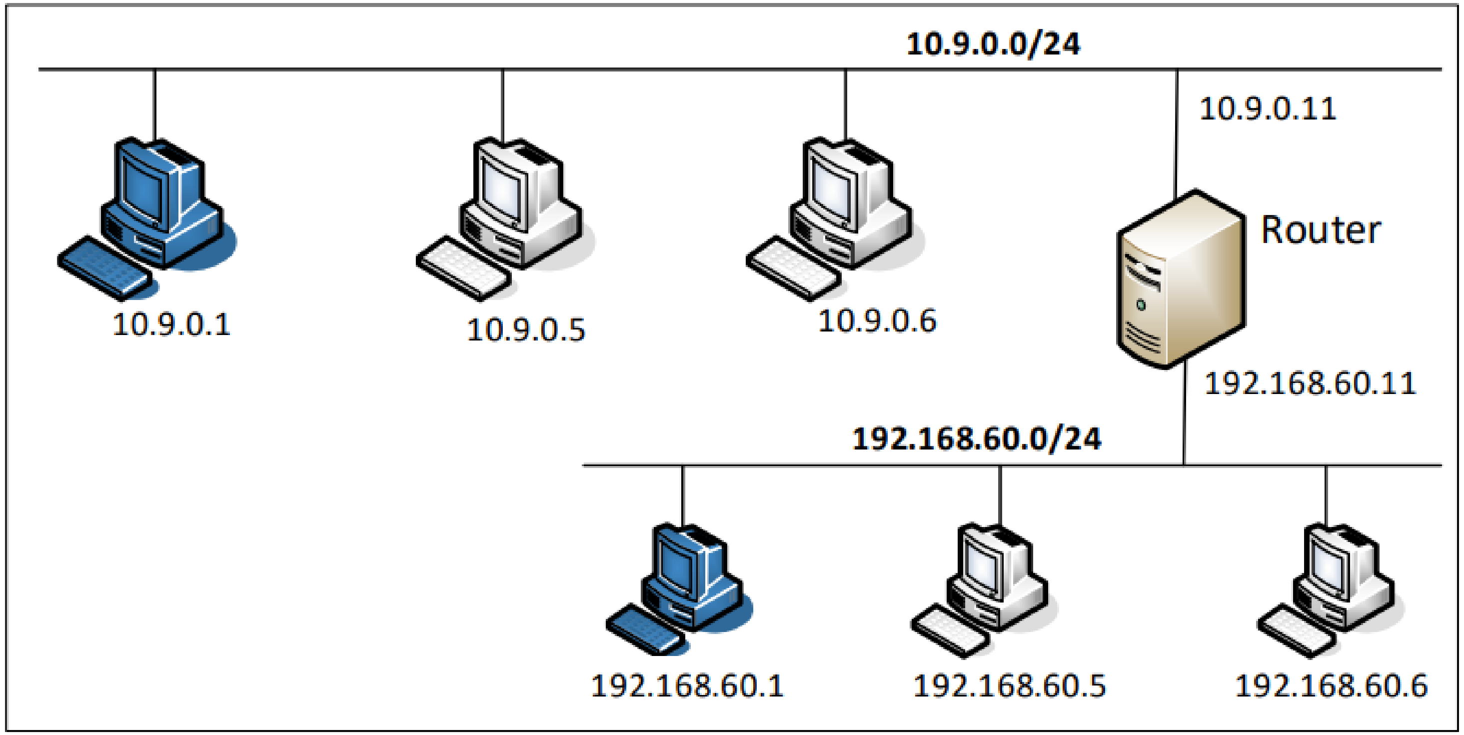
- Others: telnet, ping, etc.

PACKET SNIFFING

How Packets Are Received



Lab Setup




```
seed@VM:~$ dockps
9eb2c057887f  host-10.9.0.5
89a0dfac1c75  host-10.9.0.6
f452376e85a5  host-192.168.60.5
8856896b15ea  host-192.168.60.6
9aa28fadb047  router
```

Packet Sniffing Tools

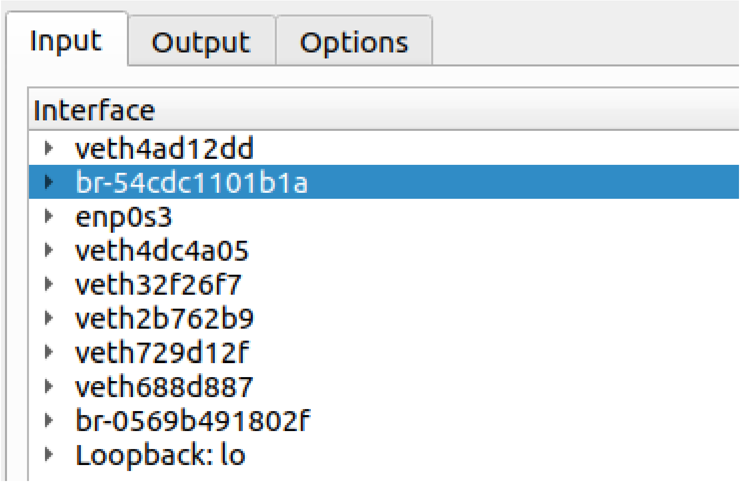
- Tcpdump
  - Command line
  - Good choice for containers (in the lab setup)
- Wireshark
  - GUI
  - Good choices for the environment supporting GUI (not containers)
- Scapy
  - Implement your own sniffing tools

Tcpdump Examples

- `tcpdump -n -i eth0`
  - **-n**: do not resolve the IP address to host name, shows ip addres instead of hostname
  - **-i**: specify the interface
- `tcpdump -n -i eth0 -vvv “tcp port 179”`
  - **-vvv**: asks the program to produce more verbose output.
- `tcpdump -i eth0 -w /tmp/packets.pcap`
  - saves the captured packets to a PCAP file
  - use Wireshark to display them

Wireshark and Containers

Find the correct interface




```
seed@VM:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d10f14b6b6f9        bridge              bridge              local
b3581338a28d        host                host                local
54cdc1101b1a        net-10.9.0.0        bridge              local
0569b491802f        net-192.168.60.0    bridge              local
77acecccbe26        none                null                local
seed@VM:~$ ip -br address
lo               UNKNOWN        127.0.0.1/8 ::1/128
enp0s3           UP             10.0.5.5/24 fe80::bed8:53e2:5192:f265/64
docker0          DOWN           172.17.0.1/16 fe80::42:13ff:fee7:90d6/64
br-54cdc1101b1a  UP             10.9.0.1/24 fe80::42:1cff:fe17:f3e6/64
br-0569b491802f  UP             192.168.60.1/24 fe80::42:b5ff:fe9b:6b49/64
```

Scapy Example 1

```
#!/usr/bin/python3

from scapy.all import *

pkt = sniff(iface='enp0s3',
            filter='icmp or udp',
            count=10)

pkt.summary()
```

```
seed@VM:~$ ip -br addr
lo               UNKNOWN        127.0.0.1/8 ::1/
enp0s3           UP             10.0.5.5/24 fe80
docker0          DOWN           172.17.0.1/16 fe
br-54cdc1101b1a  UP             10.9.0.1/24 fe80
br-0569b491802f  UP             192.168.60.1/24
```

```
root@9eb2c057887f:~# ip -br addr
lo               UNKNOWN        127.0.0.1/8
eth0@if1882      UP             10.9.0.5/24
```

Scapy Example 2

```
#!/usr/bin/python3

from scapy.all import *

def process_packet(pkt):
     #hexdump(pkt)
     pkt.show()
     print("-----------------------------------")

f = 'udp and dst portrange 50-55 or icmp'

sniff(iface='enp0s3', filter = f, prn=process_packet)
```

Filter Examples for Scapy

- Berkeley Packet Filter (BPF) syntax
- Same as tcpdump

```
dst host 10.0.2.5: only capture the packets going to 10.0.2.5.
src host 10.0.2.6: only capture the packets coming from 10.0.2.6.
host 10.0.2.6 and src port 9090: only capture the packets coming
         from or going to 10.0.2.6 with the source port being 9090.
tcp: only capture TCP packets.
```

Scapy: Display Packets

- Using hexdump()

```
>>> hexdump(pkt)
0000  52 54 00 12 35 00 08
0010  00 54 F2 29 40 00 40
0020  08 08 08 00 98 01 10
0030  0C 00 08 09 0A 0B 0C
0040  16 17 18 19 1A 1B 1C
0050  26 27 28 29 2A 2B 2C
0060  36 37
```

- Using pkt.show()

```
>>> pkt.show()
###[ Ethernet ]###
  dst        = 52:54:00:12:35:00
  src        = 08:00:27:77:2e:c3
  type       = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     ...
     proto     = icmp
     chksum    = 0x3c9a
     src       = 10.0.2.8
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
```

Scapy: Iterate Through Layers

```
>>> pkt = Ether()/IP()/UDP()/"hello"
>>> pkt
<Ether type=IPv4 |<IP frag=0 proto=udp |<UDP |<Raw load='hello' |>>>>
```

```
>>> pkt.payload                              ← an IP object
<IP  frag=0 proto=udp |<UDP  |<Raw  load='hello' |>>>

>>> pkt.payload.payload                      ← a UDP object
<UDP  |<Raw  load='hello' |>>

>>> pkt.payload.payload.payload              ← a Raw object
<Raw  load='hello' |>

>>> pkt.payload.payload.payload.load         ← the actual payload
b'hello'
```

Accessing Layers

Get inner layers

```
>>> pkt.getlayer(UDP)
<UDP  |<Raw  load='hello' |>>
>>> pkt[UDP]
<UDP  |<Raw  load='hello' |>>

>>> pkt.getlayer(Raw)
<Raw  load='hello' |>
>>> pkt[Raw]
<Raw  load='hello' |>
```

Check layer existence

```
>>> pkt.haslayer(UDP)
True
>>> pkt.haslayer(TCP)
0
>>> pkt.haslayer(Raw)
True
```

A Sniffer Example

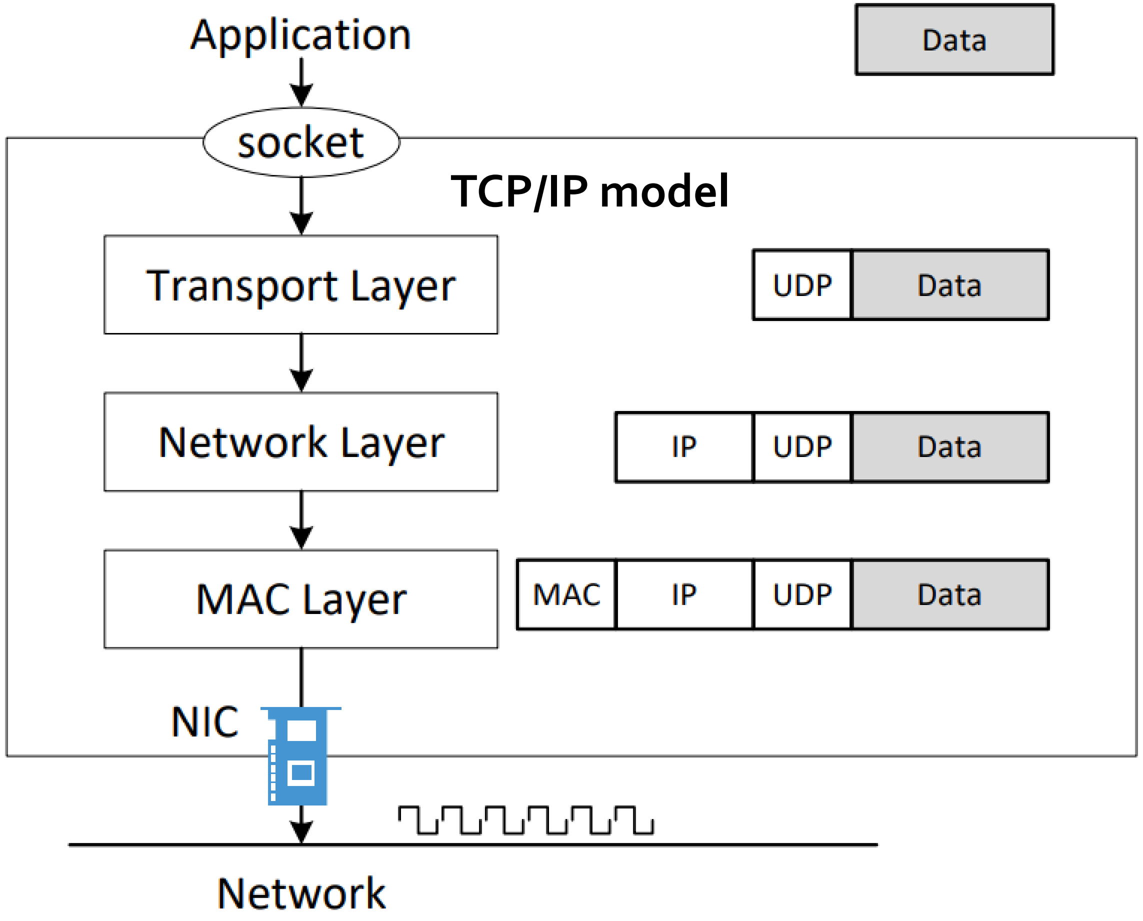
```
def process_packet(pkt):
    if pkt.haslayer(IP):
       ip =  pkt[IP]
       print("IP: {} --> {}".format(ip.src, ip.dst))

    if pkt.haslayer(TCP):
       tcp = pkt[TCP]
       print("    TCP  port: {} --> {}".format(tcp.sport, tcp.dport))

    elif pkt.haslayer(UDP):
       udp = pkt[UDP]
       print("    UDP  port: {} --> {}".format(udp.sport, udp.dport))

    elif pkt.haslayer(ICMP):
       icmp = pkt[ICMP]
       print("    ICMP type: {}".format(icmp.type))

    else:
       print("    Other protocol")


sniff(iface='enp0s3', filter='ip', prn=process_packet)
```

PACKET SPOOFING

Packet Spoofing

- In normal packet construction
  - Only some selected header fields can be set by users
  - OS set the other fields
- Packet spoofing
  - Set arbitrary header fields
  - Using tools like Scapy, a packet manipulation tool and library for Python

How To Spoof Packets



Spoofing ICMP Packets

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

print("SENDING SPOOFED ICMP PACKET.........")
ip = IP(src="1.2.3.4", dst="93.184.216.34")
icmp = ICMP()
pkt = ip/icmp
pkt.show()
send(pkt,verbose=0)
```

ICMP (Internet Control Message Protocol)

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 UDP!\n"                   # Payload
pkt = ip/udp/data
pkt.show()
send(pkt,verbose=0)
```

Sniff Request and Spoof Reply: Code

```
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, ttl = 99)
     icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
     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(iface = 'br-54cdc1101b1a',
            filter = 'icmp and src host 10.9.0.5',
            prn = spoof_pkt)
```

In ICMP, a "type" of 8 corresponds to an Echo Request message, which is commonly used for pinging a remote host.

Swap source and destination

Echo Reply (Type 0)

Other Uses of Scapy: Send and Receive

- `send()`: Send packets at Layer 3.
- `sendp()`: Send packets at Layer 2.
- `sr()`: Sends packets at Layer 3 and receiving answers.
- `srp()`: Sends packets at Layer 2 and receiving answers.
- `sr1()`: Sends packets at Layer 3 and waits for the first answer.

1

- `sr1p()`: Sends packets at Layer 2 and waits for the first answer.
- `srloop()`: Send a packet at Layer 3 in a loop and print the answer each time.

L

- `srploop()`: Send a packet at Layer 2 in a loop and print the answer each time.

Example: implement ping

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

ip = IP(dst="8.8.8.8")
icmp = ICMP()
pkt = ip/icmp
reply = sr1(pkt)
print("ICMP reply .........")
print("Source IP : ", reply[IP].src)
print("Destination IP :", reply[IP].dst)
```

Example: implement traceroute

Traceroute Code

```
b = ICMP()
a = IP()
a.dst = '93.184.216.34'

TTL = 3
a.ttl = TTL
h = sr1(a/b, timeout=2, verbose=0)
if h is None:
    print("Router: *** (hops = {})".format(TTL))
else:
    print("Router: {} (hops = {})".format(h.src, TTL))
```

Sniffing/Spoofing Using C

- C is much faster
  - My experiment: 40 times faster
- Speed is important for some attacks
  - SYN flooding
  - DNS remote attack
- Covered in another chapter