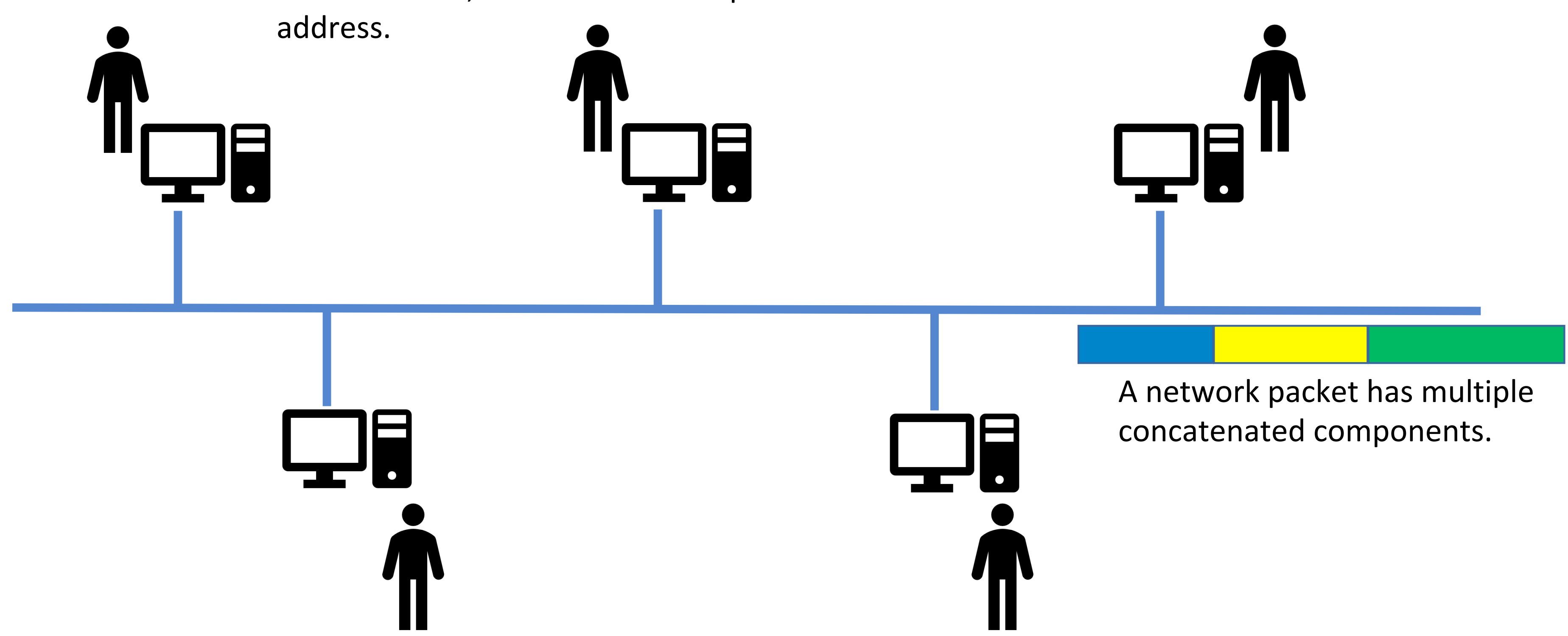
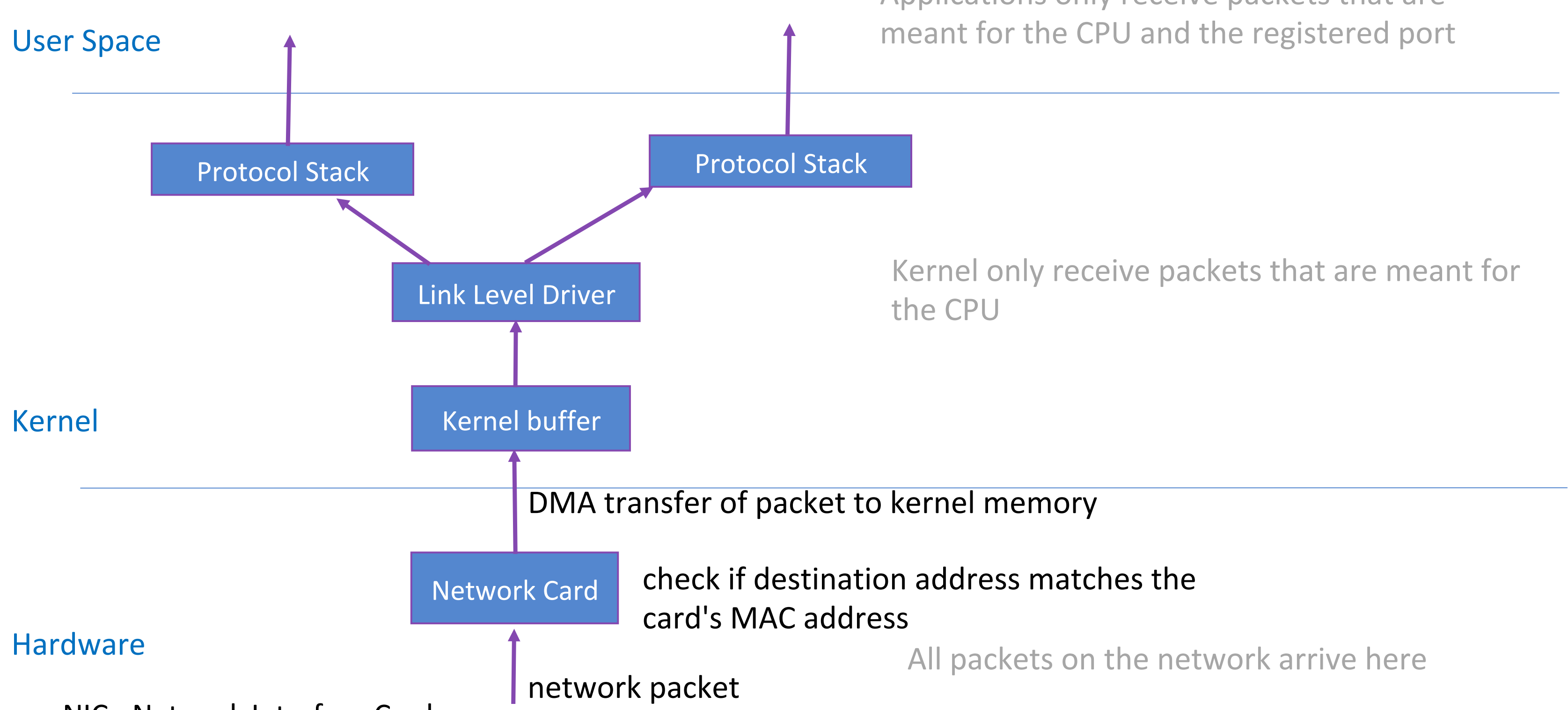
# Packet Sniffing and Spoofing

# Shared Networks



# How Packets Are Received



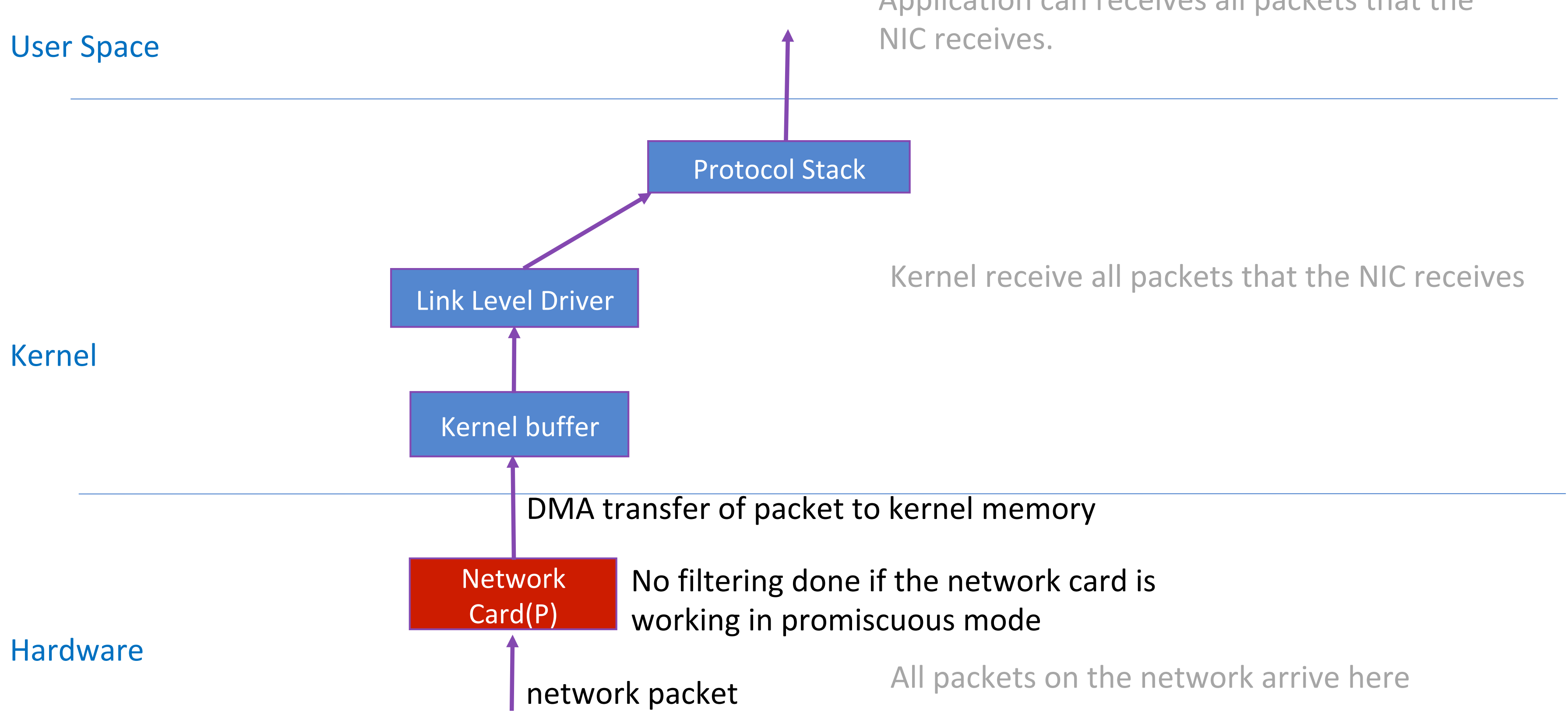

NIC - Network Interface Card

- a physical/logical link between a machine and a network
- each has a MAC address
- hear all the frames on the wire

# Promiscuous Mode

- The frames that are not destined to a given NIC are discarded
- When operating in promiscuous mode, NIC passes every frame received from the network to the kernel
- If a sniffer program is registered with the kernel, it will be able to see all the packets
- In Wi-Fi, it is called Monitor Mode

# Promiscuous Mode



# Packet Sniffing

- Packet sniffing describes the process of capturing live data as they flow across a network
- Applications that register with the kernel so as to capture all packets seen in the network.
- Typically requires superuser permissions
- Let us first see how computers receive packets.

# Receiving Packets Using Socket

Domain: IPV4. Other alternatives are AF_INET6, etc.

Type: datagram, connectionless, fixed length, unreliable

Create the socket

Provide information about server

Receive packets

```
// 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)
    error("ERROR on binding");

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

associate an address with the socket using bind()

# Receiving Packets Using Socket

Create the socket

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

Provide information about server

```
// 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)
    error("ERROR on binding");
```

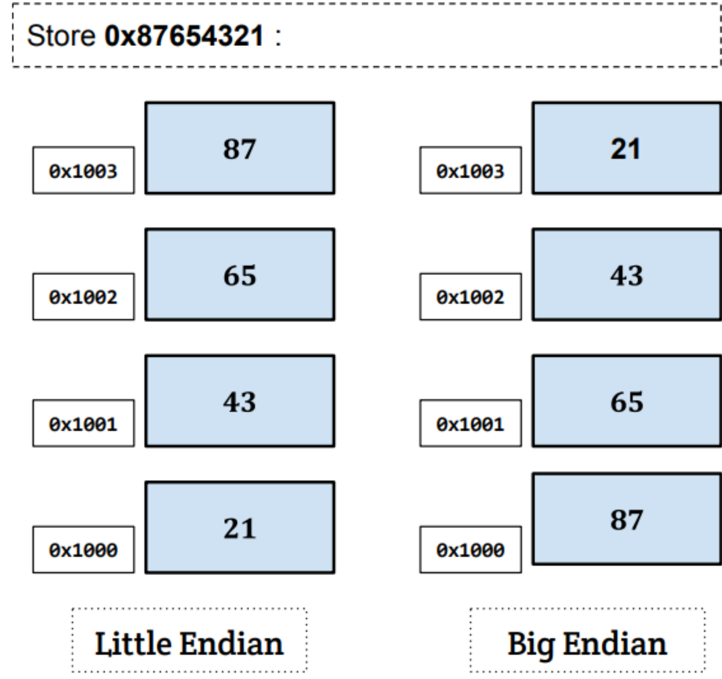
Receive packets

```
// Step ③
while (1) {
    bzero(buf,
    recvfrom(so
             (
    printf("%s\n", buf);
}
```

htons(): unsigned short from host order to network order
htonl(): unsigned long from host order to network order
ntohs() : unsigned short network to host order
ntohl() : unsigned long, network to host order

# Endianness

- Endianness: a term that refers to the order in which a given multi-byte data item is stored in memory.
  - **Little Endian**: store the most significant byte of data at the highest address
  - **Big Endian**: store the most significant byte of data at the lowest address

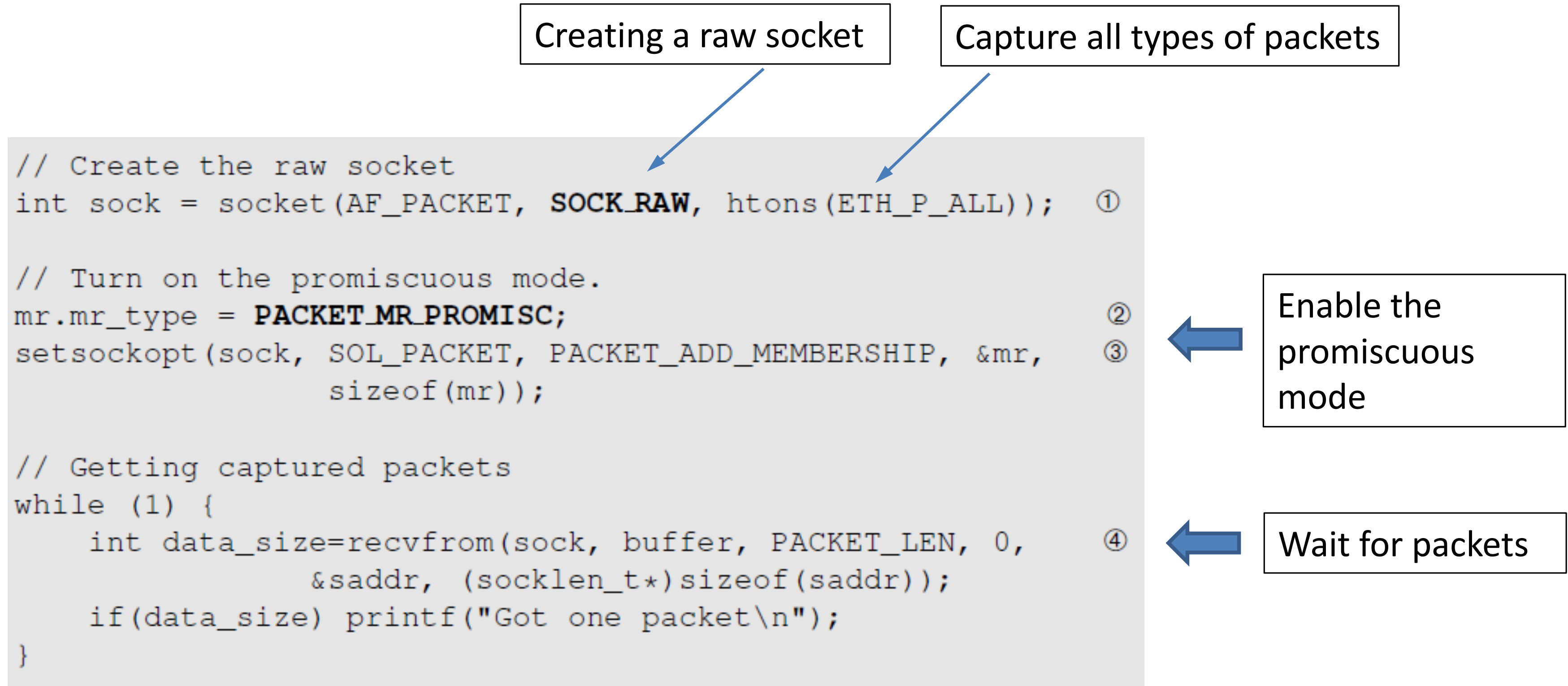


# Endianness In Network Communication

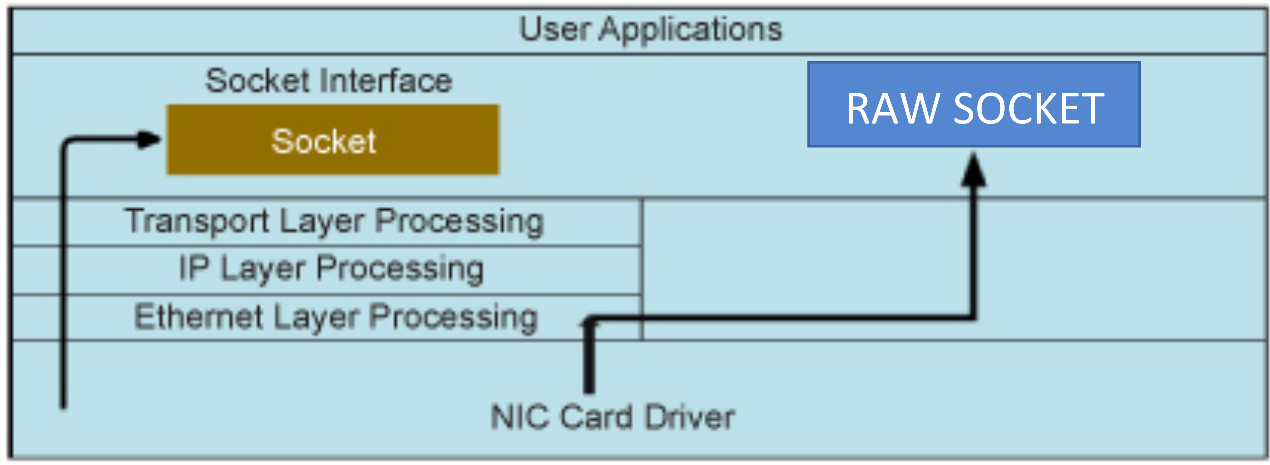
- Computers with different byte orders will "misunderstand" each other.
  - Solution: agree upon a common order for communication
  - This is called "network order", which is the same as **big endian** order
- All computers need to convert data between "host order" and "network order" .

| 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. |

# Receiving Packets Using Raw Socket



# Normal Socket vs Raw Socket




- An application creating a normal socket like a stream or datagram, will not receive the packet headers. Information like MAC address, source IP, etc. is not received. Instead only the payload present in each packet.
- In raw sockets, the headers are not clipped. Application obtains an unintercepted packet.

# Flooding of Packets in User Space

- Applications that register with the kernel so as to capture all packets seen in the network.
- Typically, sniffers are only interested in a small subset of packets, all the other packets are discarded.
  - Improves performance considerably (less processing time)
  - Would require much less expensive hardware
- Filtering must be as close to the NIC as possible (filter as early as possible)
- BSD packet filtering (BPF) provides a means by which sniffers can specify to the kernel, the packets they are interested in.

# BSD Packet Filter (BPF)

```
struct sock_filter code[] = {
  { 0x28,  0,  0, 0x0000000c }, { 0x15,  0,  8, 0x000086dd },
  { 0x30,  0,  0, 0x00000014 }, { 0x15,  2,  0, 0x00000084 },
  { 0x15,  1,  0, 0x00000006 }, { 0x15,  0, 17, 0x00000011 },
  { 0x28,  0,  0, 0x00000036 }, { 0x15, 14,  0, 0x00000016 },
  { 0x28,  0,  0, 0x00000038 }, { 0x15, 12, 13, 0x00000016 },
  { 0x15,  0, 12, 0x00000800 }, { 0x30,  0,  0, 0x00000017 },
  { 0x15,  2,  0, 0x00000084 }, { 0x15,  1,  0, 0x00000006 },
  { 0x15,  0,  8, 0x00000011 }, { 0x28,  0,  0, 0x00000014 },
  { 0x45,  6,  0, 0x00001fff }, { 0xb1,  0,  0, 0x0000000e },
  { 0x48,  0,  0, 0x0000000e }, { 0x15,  2,  0, 0x00000016 },
  { 0x48,  0,  0, 0x00000010 }, { 0x15,  0,  1, 0x00000016 },
  { 0x06,  0,  0, 0x0000ffff }, { 0x06,  0,  0, 0x00000000 },
};

struct sock_fprog bpf = {
   .len = ARRAY_SIZE(code),
   .filter = code,
};
```

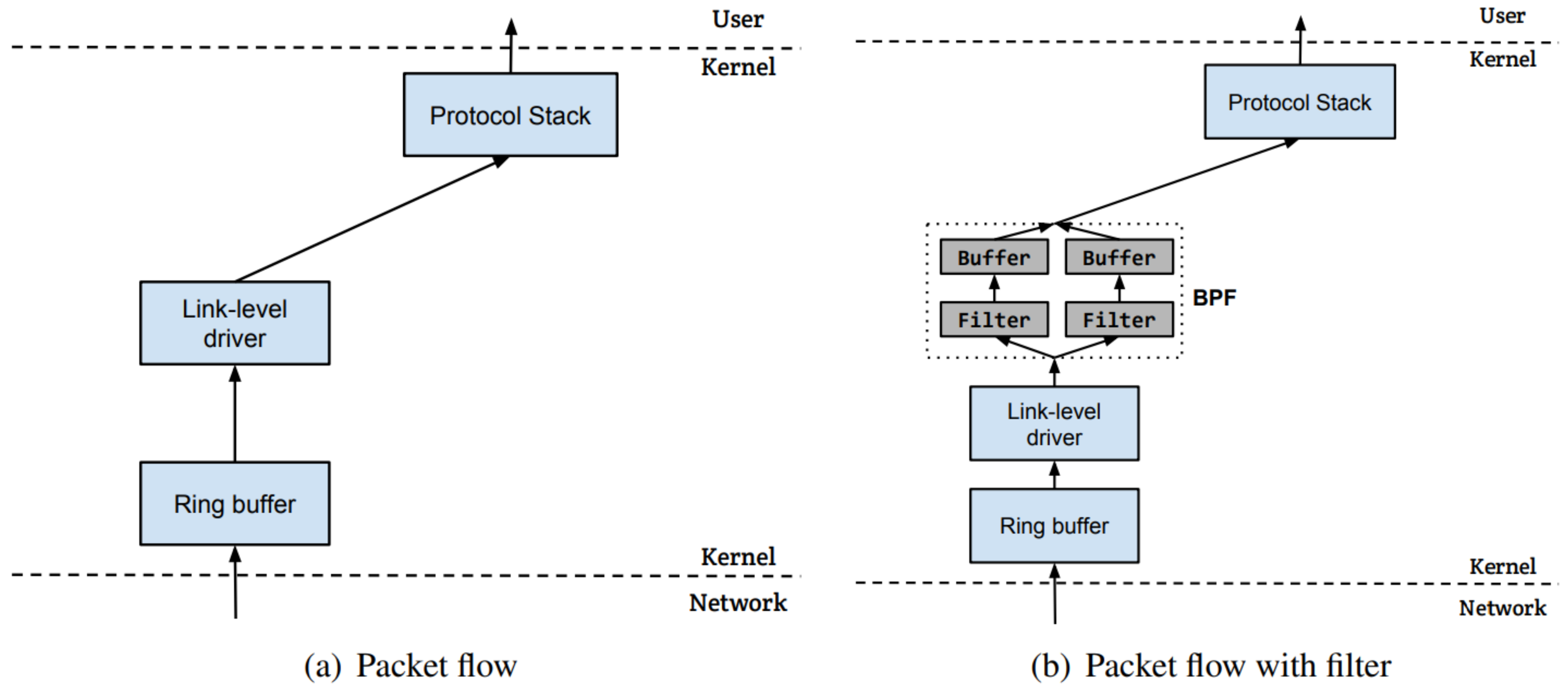
- BPF allows a user-program to attach a filter to the socket, which tells the kernel to discard unwanted packets.
- An example of the compiled BPF code is shown here.

# BSD Packet Filter (BPF)

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

- A compiled BPF pseudo-code can be attached to a socket through `setsockopt()`
- When a packet is received by kernel, BPF will be invoked
- An accepted packet is pushed up the protocol stack. See the diagram on the following slide.

# Packet Flow With/Without Filters




(a) Packet flow

(b) Packet flow with filter

# Limitations of the Approach

- The simple sniffer
  - Not portable across different operating systems
  - Not easy to set filters
  - Not explore any optimization to improve performance
- PCAP library
  - Still uses raw sockets internally, but its API is standard across all platforms. OS-specifics are hidden by PCAP's implementation.
  - Allows programmers to specify filtering rules using human readable Boolean expressions

# Packet Sniffing Using the PCap API

```
char filter_exp[] = "ip proto icmp";
```

Initialize a raw socket, set the network device into promiscuous mode.

Filter

```
// Step 1: Open live pcap session on NIC with name eth3
handle = pcap_open_live("eth3", 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
```

fills compiled BPF program in fp. Has the form **struct bpf_program *fp**

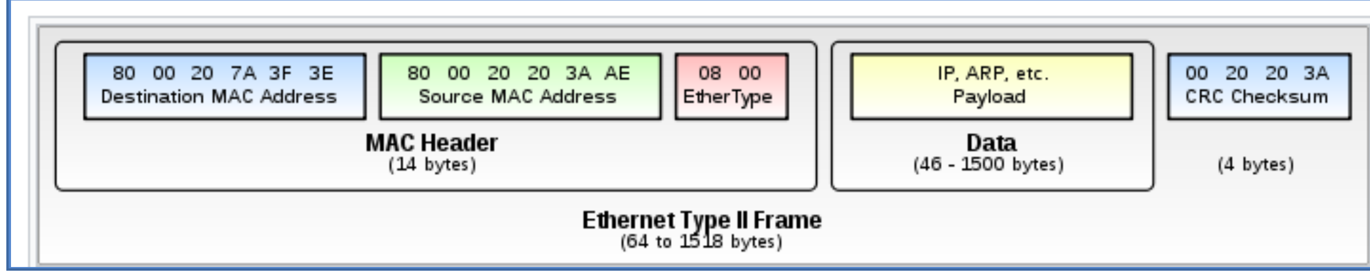
Invoke this function for every captured packet

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

filled with the packet received; contains the raw ICMP packet

# Processing Captured Packet: Ethernet Header

```
/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};
```



# Processing Captured Packet: Ethernet Header

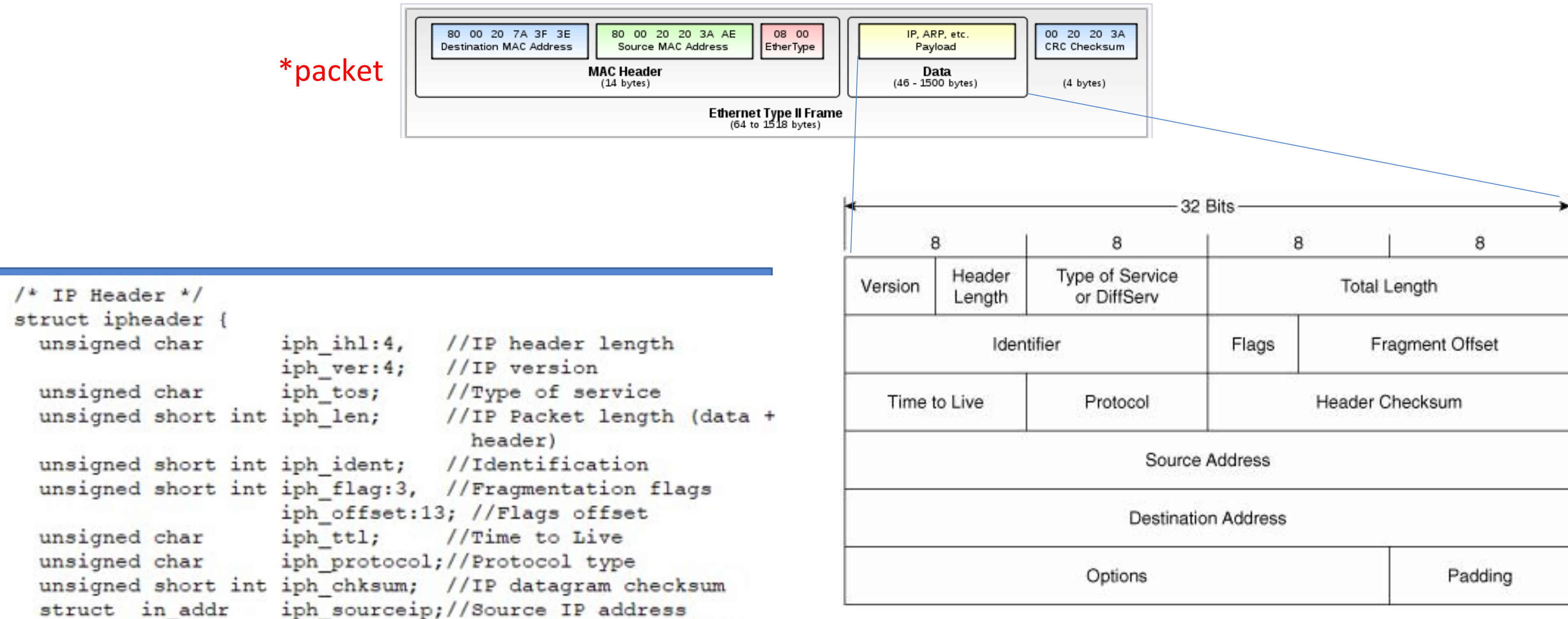
```
/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};

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

The **packet** argument contains a copy of the packet, including the Ethernet header. We typecast it to the Ethernet header structure.

Now we can access the field of the structure

# Processing Captured Packet: IP Header




```
/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4,    //IP header length
                     iph_ver:4;    //IP version
  unsigned char      iph_tos;      //Type of service
  unsigned short int iph_len;      //IP Packet length (data +
                                     header)
  unsigned short int iph_ident;    //Identification
  unsigned short int iph_flag:3,   //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl;      //Time to Live
  unsigned char      iph_protocol;//Protocol type
  unsigned short int iph_chksum;   //IP datagram checksum
  struct  in_addr    iph_sourceip;//Source IP address
  struct  in_addr    iph_destip;  //Destination IP address
};
```

# Processing Captured Packet: IP Header

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
{
  struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)
                           (packet + sizeof(struct ethheader)); ①

    printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));   ②
    printf("         To: %s\n", inet_ntoa(ip->iph_destip));     ③

    /* determine protocol */
    switch(ip->iph_protocol) {                                   ④
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
```

Find where the IP header starts and typecast it to the IP Header structure.

Now we can easily access the fields in the IP header.

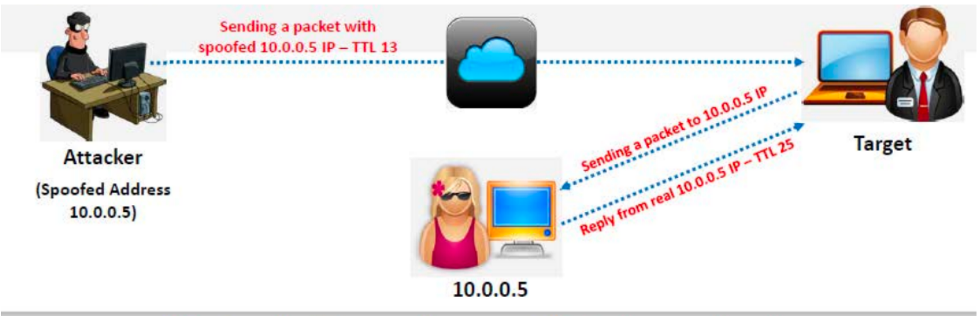
# Further Processing Captured Packet

- If we want to further process the packet, such as printing out the header of the TCP, UDP and ICMP, we can use the similar technique.
  - We move the pointer to the beginning of the next header and type-cast
  - We need to use the header length field in the IP header to calculate the actual size of the IP header
- In the following example, if we know the next header is ICMP, we can get a pointer to the ICMP part by doing the following:

```
int ip_header_len = ip->iph_ihl * 4;
u_char *icmp = (struct icmpheader *)
               (packet + sizeof(struct ethheader) + ip_header_len);
```

# Packet Spoofing

- When some critical information in the packet is forged, we refer to it as packet spoofing.
- Many network attacks rely on packet spoofing.
- Let us see how to send packets without spoofing.



# Sending Normal Packets Using Sockets

```
void main()
{
    struct sockaddr_in dest_info;
    char *data = "UDP message\n";

    // Step 1: Create a network socket
    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    // Step 2: Provide information about destination.
    memset((char *) &dest_info, 0, sizeof(dest_info));
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr.s_addr = inet_addr("10.0.2.5");
    dest_info.sin_port = htons(9090);

    // Step 3: Send out the packet.
    sendto(sock, data, strlen(data), 0,
                 (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
```

Testing

- `netcat` (`nc`) command to run a UDP server on `10.0.2.5`.
- run the program on the left from another machine
- message delivered to the server machine

```
seed@Server(10.0.2.5):$ nc -luv 9090
Connection from 10.0.2.6 port 9090 [udp/*] accepted
UDP message
```

# Manipulating Transmitted Packets

- Generally, transmitting packets has only control of few fields in the header.
  - e.g., destination IP address can be set, source IP address is not set
  - OS will automatically fill these fields before transmitting the packet to the hardware
- Spoofing
  - Permits manipulation of critical fields in the packet headers
  - Creates unrealistic / bogus packets
    - E.g., Transmit a TCP packet with SYN and FIN bits turned on
    - The response from the receiver is unpredictable (depends on the OS )
  - Is used in many network attacks
    - E.g., TCP SYN Flooding, TCP session hijacking, DNS cache poisoning attack
    - Supplied information depends on the type of attack being carried out

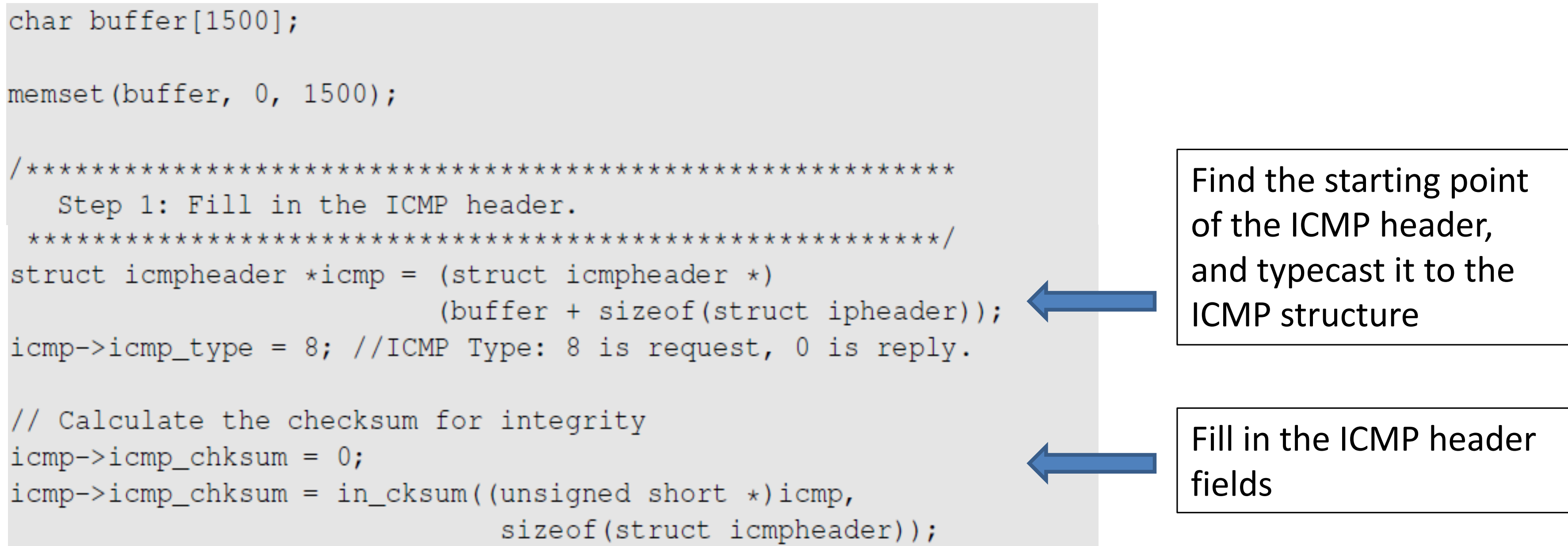
# Spoofing Packets Using Raw Sockets

Two major steps in packet spoofing

- Constructing the packet
- Sending the packet out

# Spoofing Packets: Step 1. Constructing the Packet

## Fill in the ICMP Header



```
char buffer[1500];

memset(buffer, 0, 1500);

/*********************************************************
   Step 1: Fill in the ICMP header.
 *********************************************************/
struct icmpheader *icmp = (struct icmpheader *)
                          (buffer + sizeof(struct ipheader));
icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

// Calculate the checksum for integrity
icmp->icmp_chksum = 0;
icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                              sizeof(struct icmpheader));
```

# Spoofing Packets: Step 1. Constructing the Packet

## Fill in the IP Header

```
/*************************************************************
   Step 2: Fill in the IP header.
 *************************************************************/
struct ipheader *ip = (struct ipheader *) buffer;
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("10.0.2.5");
ip->iph_protocol = IPPROTO_ICMP;
ip->iph_len = htons(sizeof(struct ipheader) +
                    sizeof(struct icmpheader));
```

Typecast the buffer to the IP structure

Fill in the IP header fields

**Finally, send out the packet**

```
send_raw_ip_packet (ip);
```

# Spoofing Packets: Step 2. Sending Packets Using Raw Sockets

```
/*************************************************************
  Given an IP packet, send it out using a raw socket.
*************************************************************/
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);
}
```

We use *setsockopt()* to enable *IP_HDRINCL* on the socket.

For raw socket programming, since the destination information is already included in the provided IP header, we do not need to fill all the fields

Since the socket type is raw socket, the system will send out the IP packet as is.

# Spoofing UDP Packets

```
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. */
```

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

# Spoofing UDP Packets (continued)

```
/*********************************************************
   Step 3: Fill in the IP header.
 *********************************************************/

...... /* Code omitted here; same as that in Listing 12.6 */
ip->iph_protocol = IPPROTO_UDP; // The value is 17.
ip->iph_len = htons(sizeof(struct ipheader) +
                    sizeof(struct udpheader) + data_len);
```

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.

```
seed@Server(10.0.2.5):$ nc -luv 9090
Connection from 1.2.3.4 port 9090 [udp/*] accepted
Hello Server!
```

# Sniffing and Then Spoofing

- In many situations, we need to capture packets first, and then spoof a response based on the captured packets.
- 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

# UDP Packet

```
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);
```

# UDP Packet (Continued)

```
    // 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);
}
```

# Packing Sniffing Using Scapy

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

print("SNIFFING PACKETS.........")

def print_pkt(pkt):                                   ①
   print("Source IP:", pkt[IP].src)
   print("Destination IP:", pkt[IP].dst)
   print("Protocol:", pkt[IP].proto)
   print("\n")

pkt = sniff(filter='icmp',prn=print_pkt)              ②
```

# Spoofing ICMP & UDP Using Scapy

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

print("SENDING SPOOFED ICMP PACKET.........")
ip = IP(src="1.2.3.4", dst="93.184.216.34")  ①
icmp = ICMP()                                 ②
pkt = ip/icmp                                 ③
pkt.show()
send(pkt,verbose=0)                           ④
```

```
#!/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      # Construct the complete packet
pkt.show()
send(pkt,verbose=0)
```

# Sniffing and Then Spoofing Using Scapy

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

def spoof_pkt(pkt):
  if ICMP in pkt and pkt[ICMP].type == 8:
     print("Original Packet.........")
     print("Source IP : ", pkt[IP].src)
     print("Destination IP :", pkt[IP].dst)

     ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
     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(filter='icmp and src host 10.0.2.69',prn=spoof_pkt)
```

# Packet Spoofing: Scapy v.s C

- Python + Scapy
  - Pros: constructing packets is very simple
  - Cons: much slower than C code
- C Program (using raw socket)
  - Pros: much faster
  - Cons: constructing packets is complicated
- Hybrid Approach
  - Using Scapy to construct packets
  - Using C to slightly modify packets and then send packets