
Packet Sniffing and Spoofing

How Packets Are Received

- **NIC (Network Interface Card) is a physical or logical link between a machine and a network**
- **Each NIC has a MAC address**
- **Every NIC on the network will hear all the frames on the wire**
- **NIC checks the destination address for every packet, if the address matches the cards MAC address, it is further copied into a buffer in the kernel**

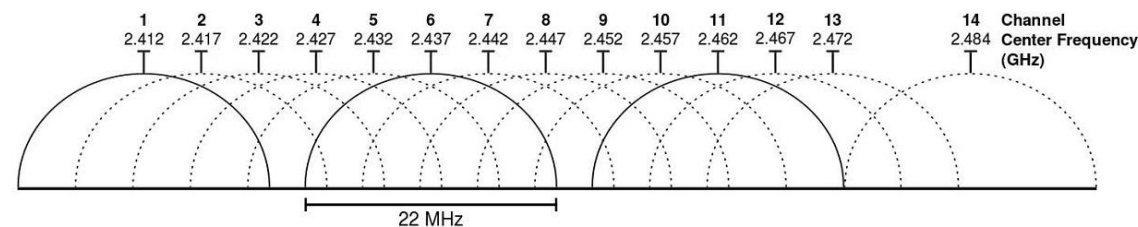
Promiscuous Mode

- **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**

2.4 Channels

Channel number	Bottom of channel	Center frequency	Top of channel
1	2401	2412	2423
2	2406	2417	2428
3	2411	2422	2433
4	2416	2427	2438
5	2421	2432	2443
6	2426	2437	2448
7	2431	2442	2453
8	2436	2447	2458
9	2441	2452	2463
10	2446	2457	2468
11	2451	2462	2473

CHANNELS 1, 6, 11
DON'T OVERLAP WITH
ONE ANOTHER



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,
};
```

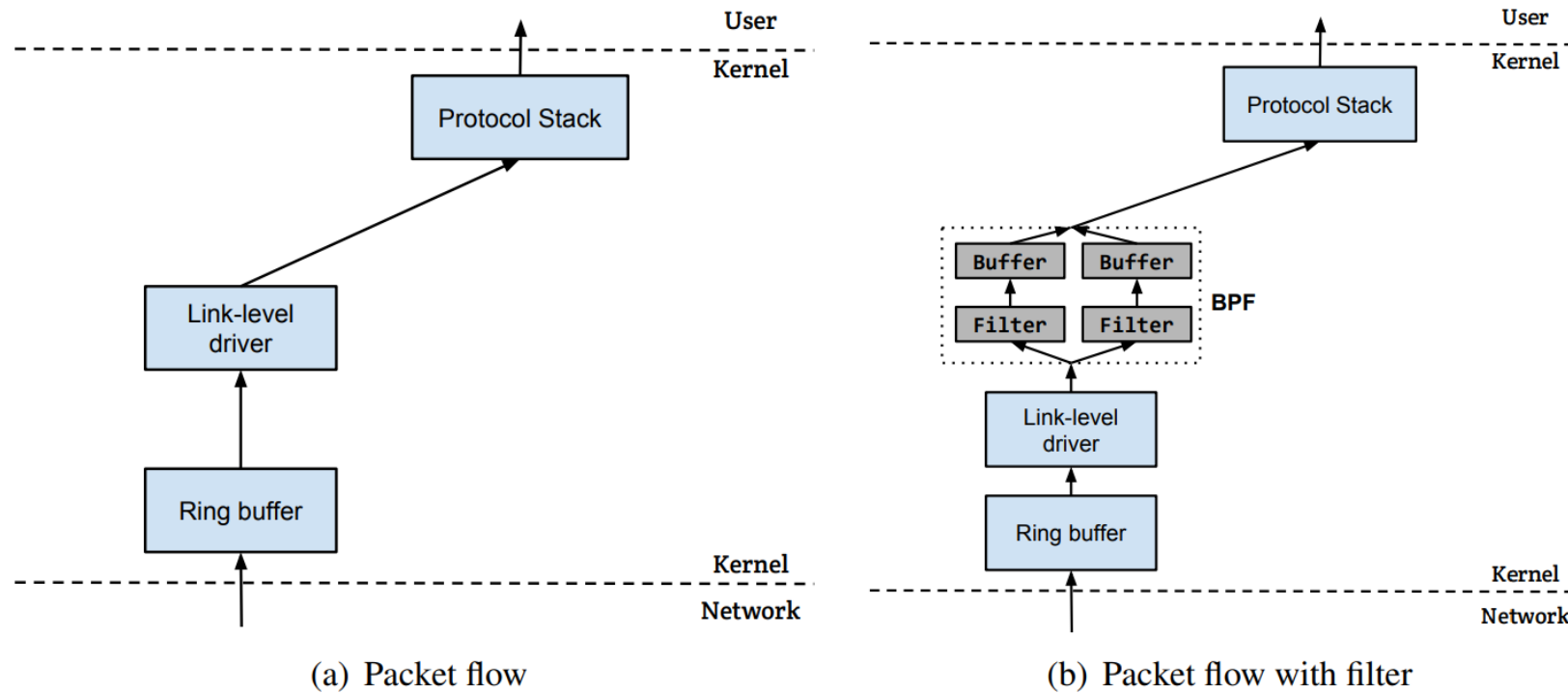
- **BPF allows a user-program to attach a filter to the socket, which tells the kernel to discard unwanted packets.**
- **An example of the compiled BPF code is shown here.**

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



Packet Sniffing

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

Let's first see how computers receive packets.

IPv4 address structure

- Port numbers and IP addresses are in ***network byte order***.

```
/* A protocol-independent address structure. */
struct sockaddr {
    short int sa_family;    /* Address family; AF_INET, AF_INET6 */
    char sa_data[14];      /* 14 bytes of protocol specific address */
};

/* An IPv4 specific address structure.*/
struct sockaddr_in {
    sa_family_t sin_family; /* Address family, AF_INET => IPv4 */
    in_port_t sin_port;    /* Port number in network byte order */
    struct in_addr sin_addr; /* Internet address */
};

/* Internet Address */
struct in_addr {
    uint32_t s_addr; /* IPv4 address in network byte order */
};
```

Addresses

- **INADDR_LOOPBACK (127.0.0.1)**
 - always refers to the local host via the loopback device
- **INADDR_ANY**
 - (0.0.0.0) means any address for binding
- **INADDR_BROADCAST**
 - (255.255.255.255) means any host

Functions

- **Create an endpoint for communication**

- `int socket(int domain, int type, int protocol);`

- **Bind a name to a socket**

- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

- **Receive message from socket**

- `ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);`

Receiving Packets Using Socket

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);
}
```

Receiving Packets Using Raw Socket

Creating a raw socket

Capture all types of packets

```
// 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");
}
```

Enable the
promiscuous
mode

Wait for packets

Limitation of the Approach

- **This program is not portable across different operating systems.**
- **Setting filters is not easy.**
- **The program does not explore any optimization to improve performance.**
- **The PCAP library was thus created.**
 - It 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";
```

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
```

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

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");
}
```

Processing Captured Packet: Ethernet Header

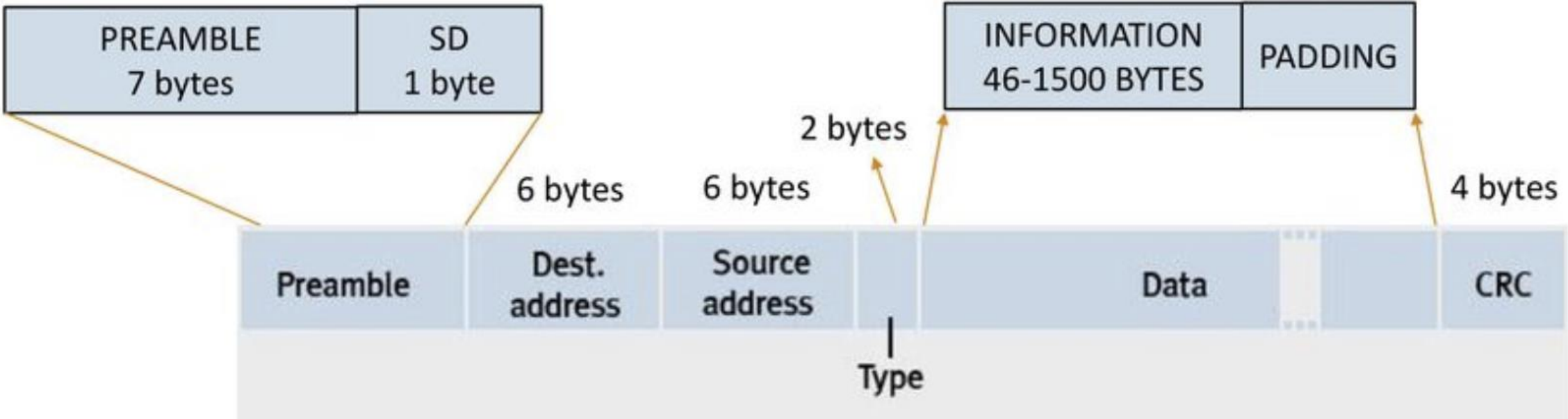
```
/* Ethernet header */
struct ethheader {
    u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
    u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
    u_short ether_type;                  /* IP? ARP? RARP? etc */
};

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

Ethernet Frame



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's see how to send packets without spoofing.**

Sending Packets Without Spoofing

```
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: Use the `netcat (nc)` command to run a UDP server on `10.0.2.5`. We then run the program on the left from another machine. We can see that the message has been 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
```

Spoofing Packets Using Raw Sockets

There are two major steps in packet spoofing:

- Constructing the packet
- Sending the packet out

Spoofing 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 Packets: 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));
```

Find the starting point
of the ICMP header,
and typecast it to the
ICMP structure

Fill in the ICMP
header fields

Spoofing Packets: 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 icmphheader));
```

Typecast the buffer to
the IP structure

Fill in the IP header
fields

Finally, send out the packet:

```
send_raw_ip_packet (ip);
```

Spoofing UDP Packets

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

/*****
  Step 1: Fill in the UDP data field.
  *****/
char *data = buffer + sizeof(struct ipheader) +
              sizeof(struct udphheader);
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 udphheader) + 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 udphheader) + 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!
```