Packet Sniffing and Spoofing

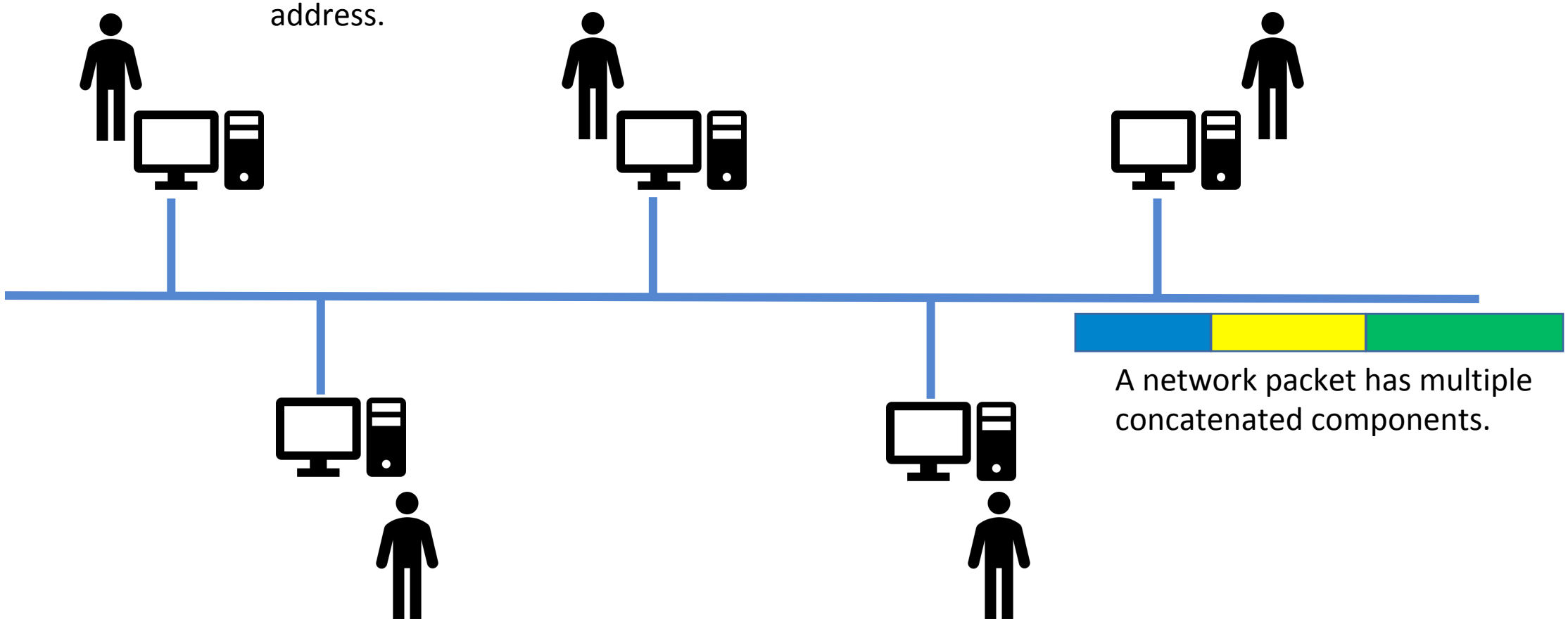
Chester Rebeiro

IIT Madras

Some of the slides borrowed from the book ‘Computer Security: A Hands on Approach’ by Wenliang Du

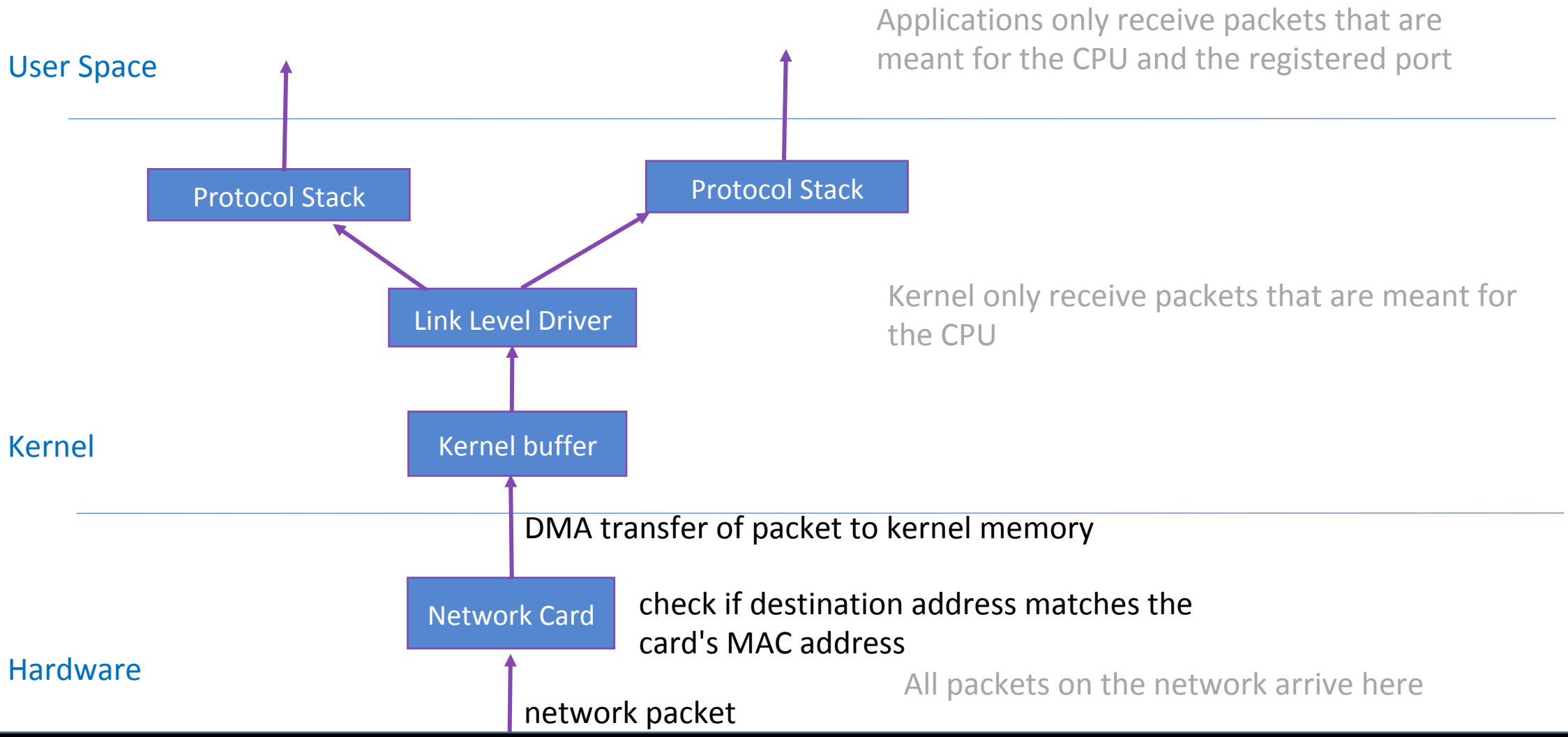
Shared Networks

Every network packet reaches every computer's network Interface card, which then filters packets based on the MAC address.



A network packet has multiple concatenated components.

Packet Flow in the System



From the Software

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);
}
```

From Software

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);
}
```

Domain: IPV4. Other alternatives are AF_INET6 and many more

Type: datagram, connectionless, fixed length, unreliable

associate an address with the socket with the bind call

From Software

Create the socket

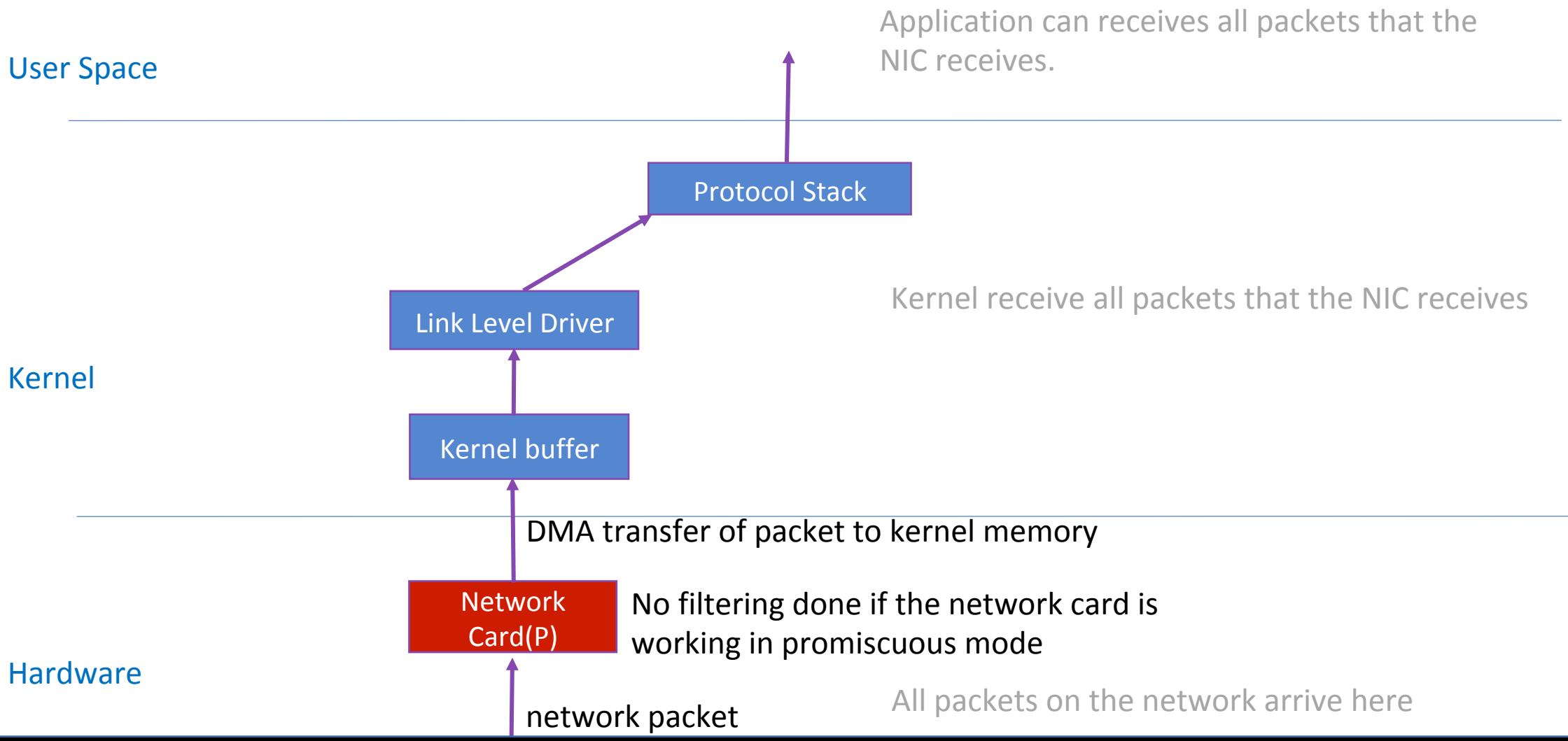
Provide information about server

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
    error("ER

// Step ③
while (1) {
    bzero(buf
    recvfrom(sock, buf, 1500-1, 0,
             (struct sockaddr *) &client, &clientlen);
    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

Promiscuous Mode



Packet Sniffers

- Applications that register with the kernel so as to capture all packets seen in the network.
- Typically requires superuser permissions

Packet Sniffers

Specify that the socket you want to create is a RAW socket.

Protocol family: AF_PACKET implies low level protocol

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

Packet Sniffers

What type of packets should we receive? ETH_P_ALL, implies all protocols. Other options are for instance, ETH_P_IP, for only IP 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");
}
```

Packet Sniffers

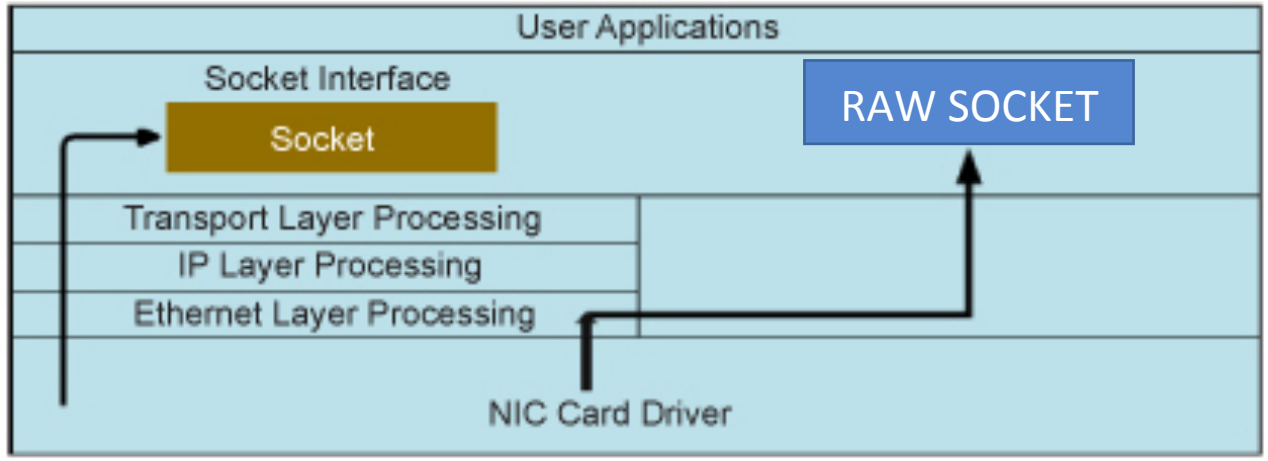
Configure the NIC to ensure that all packets are accepted and passed to the kernel. Ignore the destination field in the packets.

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

Packet Sniffers

Specify that the socket you want to create is a RAW socket.

```
// 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
```



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

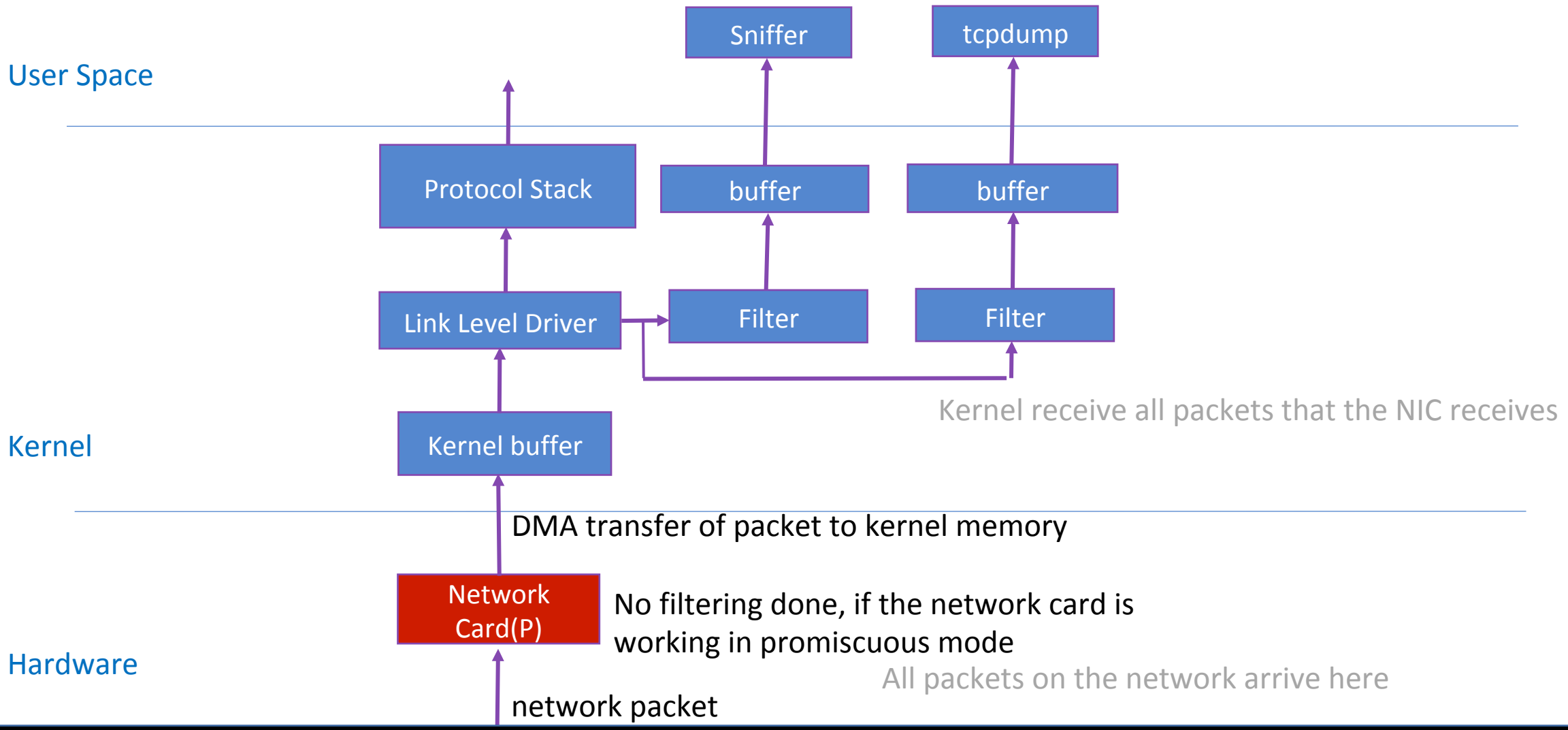
- Applications that register with the kernel so as to capture all packets seen in the network.
- Typically, sniffers are only interested in a small subset of packets, all the other packets are discarded.
  - Improves performance considerably (less processing time)
  - Would require much less expensive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

Filter Requirements

- Must be programmable
  - Each sniffer may be interested in a different set of packets.
- Must be as close to the NIC as possible (filter as early as possible)
  - Rules out user-space filtering
  - Kernel based filtering
  - Hardware based filtering

Operating System Filters

Sniffer only receives all packets that the NIC receives AND that pass the filter.



BSD Packet Filters (BPF)

- 1992, Steven McCanne and Van Jacobson from Lawrence Berkeley Laboratory
- Incorporated in Linux kernel in 1997
  - Variants still used in latest versions
- JIT engine
  - Low level language defined
  - User level application writes filter rules using this language and attaches it to a socket
  - The kernel, verifies sanity of these rules and then applies them to all packets it receives.

bpf architecture

Architecture

Element	Description
A	32 bit wide accumulator
X	32 bit wide X register
M[]	16 x 32 bit wide misc registers aka "scratch memory store", addressable from 0 to 15

https://www.kernel.org/doc/Documentation/networking/filter.txt

bpf architecture

Element	Description
A	32 bit wide accumulator
X	32 bit wide X register
M[]	16 x 32 bit wide misc registers aka "scratch memory store", addressable from 0 to 15

Instruction Set

Instruction	Addressing mode	Description
ld	1, 2, 3, 4, 12	Load word into A
ldi	4	Load word into A
ldh	1, 2	Load half-word into A
ldb	1, 2	Load byte into A
ldx	3, 4, 5, 12	Load word into X
ldxi	4	Load word into X
ldxb	5	Load byte into X
st	3	Store A into M[]
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <x>
jneq	9, 10	Jump on A != <x>
jne	9, 10	Jump on A != <x>
jlt	9, 10	Jump on A < <x>
jle	9, 10	Jump on A <= <x>
jgt	7, 8, 9, 10	Jump on A > <x>
jge	7, 8, 9, 10	Jump on A >= <x>
jset	7, 8, 9, 10	Jump on A & <x>
add	0, 4	A + <x>
sub	0, 4	A - <x>
mul	0, 4	A * <x>
div	0, 4	A / <x>
mod	0, 4	A % <x>
neg		!A
and	0, 4	A & <x>
or	0, 4	A | <x>
xor	0, 4	A ^ <x>
lsh	0, 4	A << <x>
rsh	0, 4	A >> <x>

https://www.kernel.org/doc/Documentation/networking/filter.txt

bpf architecture

Element	Description
A	32 bit wide accumulator
X	32 bit wide X register
M[]	16 x 32 bit wide misc registers aka "scratch memory store", addressable from 0 to 15

Addressing Modes

Addressing mode	Syntax	Description
0	x/%x	Register X
1	[k]	BHW at byte offset k in the packet
2	[x + k]	BHW at the offset X + k in the packet
3	M[k]	Word at offset k in M[]
4	#k	Literal value stored in k
5	4*([k]&0xf)	Lower nibble * 4 at byte offset k in the packet
6	L	Jump label L
7	#k,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
8	x/%x,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
9	#k,Lt	Jump to Lt if predicate is true
10	x/%x,Lt	Jump to Lt if predicate is true
11	a/%a	Accumulator A
12	extension	BPF extension

Instruction	Addressing mode	Description
ld	1, 2, 3, 4, 12	Load word into A
ldi	4	Load word into A
ldh	1, 2	Load half-word into A
ldb	1, 2	Load byte into A
ldx	3, 4, 5, 12	Load word into X
ldxi	4	Load word into X
ldxb	5	Load byte into X
st	3	Store A into M[]
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <x>
jneq	9, 10	Jump on A != <x>
jne	9, 10	Jump on A != <x>
jlt	9, 10	Jump on A < <x>
jle	9, 10	Jump on A <= <x>
jgt	7, 8, 9, 10	Jump on A > <x>
jge	7, 8, 9, 10	Jump on A >= <x>
jset	7, 8, 9, 10	Jump on A & <x>
add	0, 4	A + <x>
sub	0, 4	A - <x>
mul	0, 4	A * <x>
div	0, 4	A / <x>
mod	0, 4	A % <x>
neg		!A
and	0, 4	A & <x>
or	0, 4	A | <x>
xor	0, 4	A ^ <x>
lsh	0, 4	A << <x>
rsh	0, 4	A >> <x>

https://www.kernel.org/doc/Documentation/networking/filter.txt

bpf architecture

Extensions

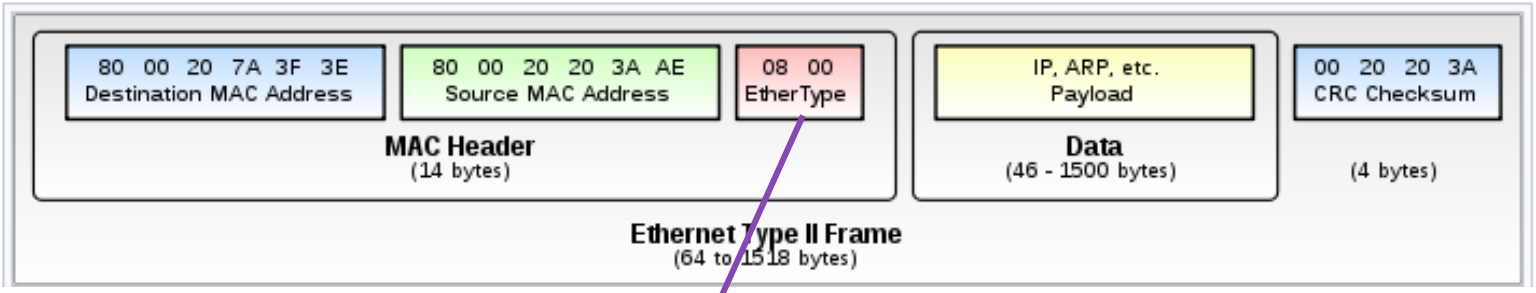
Extension	Description
len	skb->len
proto	skb->protocol
type	skb->pkt_type
poff	Payload start offset
ifidx	skb->dev->ifindex
nla	Netlink attribute of type X with offset A
nlan	Nested Netlink attribute of type X with offset A
mark	skb->mark
queue	skb->queue_mapping
hatype	skb->dev->type
rxhash	skb->hash
cpu	raw_smp_processor_id()
vlan_tci	skb_vlan_tag_get(skb)
vlan_avail	skb_vlan_tag_present(skb)
vlan_tpid	skb->vlan_proto
rand	prandom_u32()

https://www.kernel.org/doc/Documentation/networking/filter.txt

bpf asm example

```
** IPv4 TCP packets:

  ldh [12]
  jne #0x800, drop
  ldb [23]
  jneq #6, drop
  ret #-1
  drop: ret #0
```

Load 2 bytes (half word) from the 12th offset in the packet



A value of 0x0800 indicates that data is an IPv4 packet

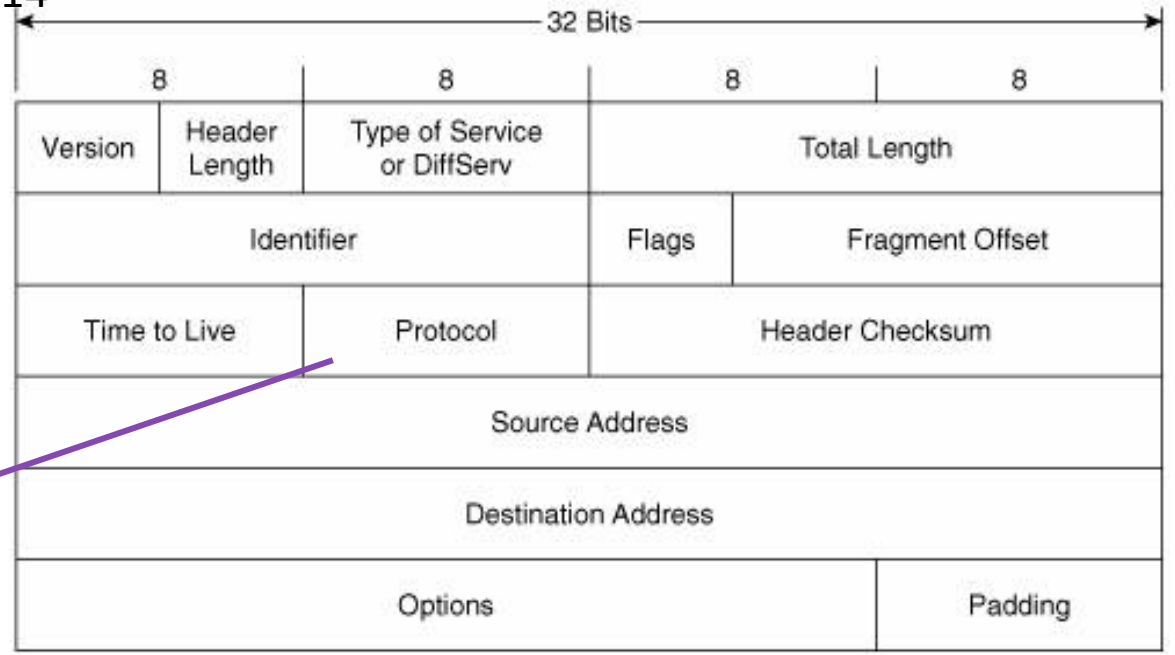
https://en.wikipedia.org/wiki/EtherType

bpf asm example

```
** IPv4 TCP packets:

  ldh [12]
  jne #0x800, drop
  ldb [23]
  jneq #6, drop
  ret #-1
  drop: ret #0
```

Reaches here only if it is an IPv4 packet.
We now check if it is a TCP packet

14



IPV4 Header

At offset 23, a value of 6 indicates that data is a TCP packet

https://en.wikipedia.org/wiki/EtherType

bpf asm another example

Randomly sample 25% of the ICMP packets

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #1, drop
# get a random uint32 number
ld rand
mod #4
jneq #1, drop
ret #-1
drop: ret #0
```

bpf_asm

```
ld [4]
jne #0xc000003e, bad
ld [0]
jeq #15, good
jeq #231, good
jeq #60, good
jeq #0, good
jeq #1, good
jeq #5, good
jeq #9, good
jeq #14, good
jeq #13, good
jeq #35, good
bad: ret #0
good: ret #0x7fff0000
```

Bpf assembly


```
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  1, 0x00000806 },
{ 0x06,  0,  0, 0xffffffff },
{ 0x06,  0,  0, 0000000000 },
```

Bpf opcode

bpf in the Linux kernel

- JIT compiler built into the Linux kernel
- Can be enabled as follows:

```
echo 1 > /proc/sys/net/core/bpf_jit_enable
```

- Internally 64-bit kernels use an enhanced BPF (eBPF) format
- Internally 32-bit kernels use the classical BPF format

Usage in Linux

```
struct sock_filter {    /* Filter block */
        __u16   code;   /* Actual filter code */
        __u8    jt;     /* Jump true */
        __u8    jf;     /* Jump false */
        __u32   k;      /* Generic multiuse field */
};
```

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

sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
if (sock < 0)
        /* ... bail out ... */

ret = setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf));
```

Create a raw socket and attach the filter.

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <linux/if_ether.h>
/* ... */

/* From the example above: tcpdump -i em1 port 22 -dd */
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 },
};
```

filter to dump packets on interface em1 port 22.

setsockopt

- SO_ATTACH_FILTER: attach a filter to a socket
- SO_DETACH_FILTER: detach a filter from a socket.
- SO_LOCK_FILTER: lock a filter to a socket. The filter cannot be detached or modified. Any attempt to detach a locked filter will result in an error.

Enhanced BPF

- Instructions looks more like that of the native architecture (makes coding simpler)
- 10 registers (R0 to R9) instead of 2 registers (A, X) with each register 64 bit instead of 32 bit
- A Frame Register (R10)

```
* R0        - return value from in-kernel function, and exit value for eBPF program
* R1 - R5   - arguments from eBPF program to in-kernel function
* R6 - R9   - callee saved registers that in-kernel function will preserve
* R10       - read-only frame pointer to access stack
```

Enhanced BPF

- Restricted C compiled to eBPF (C->eBPF->native code).
- Closer (1-to-1) mapping from eBPF to native code
- Instructions looks more like that of the native architecture (makes coding simpler)
  - 10 registers (R0 to R9) instead of 2 registers (A, X) with each register 64 bit instead of 32 bit
  - A Frame Register (R10)
  - jt/jf replaced with jf/fall-through
  - bpf_call instruction which can call other kernel functions

Checks in the Kernel

- Before attaching a filter, the following checks need to be performed.
- BPF program terminates (does not have any loops)
  - Depth first search of the program's control flow graph
  - Unreachable instructions are prohibited
- Verify by single stepping through each line in the BPF program
  - Ensure virtual machine state and check if the stack is valid
  - Prevent out-of-bound jumps and out-of-range data
- Ensure no pointer arithmetic
- Ensure registers are not read before being accessed

Limitations

- Not portable. Programs written for one operating system may not work on another OS (No common API)
- Optimizations in the filtering not easily achieved. The JIT compiler in the OS cannot extract optimizations.
- Usability is not easy. Programmers would need to efficiently develop BPF code.

PCap (**P**acket **Cap**ture)

- It is a library that provides APIs for packet capture.
- Has a compiler (*pcap_compile*) that
  - Takes as input filtering rules using human readable Boolean expressions.
  - Converts the Boolean expressions into BPF pseudo-code, which can be used by the kernel.
- Well defined APIs available on many platforms:
  - Port in Linux is called *libpcap*
  - Port in Windows is called *WinPCap*.
    (APIs are common across ports)

PCap filter expressions

Three types of qualifiers: type, dir, proto

1. **type**: identifier of a machine, port number etc.

   Options include: host, net, port, portrange

   Examples:

   host iitm.ac.in

   port 5000

   portrange 5000-6000

https://linux.die.net/man/7/pcap-filter

PCap filter expressions

Three types of qualifiers.

2. **dir**: transfer directions to or from the id.

   Options include: **src**, **dst**, **src or dst**, **src and dst**,

   Examples:

   src host iitm.ac.in

   src or dst port 5000 (equivalent to port 5000)

   portrange 5000-6000

https://linux.die.net/man/7/pcap-filter

PCap filter expressions

Three types of qualifiers.

3. **proto**: transfer directions to or from the id.

   Options include: **ether**, **fddi**, **tr**, **wlan**, **ip**, **ip6**, **arp**, **rarp**, **decnet**, **tcp** and **udp**

Examples:

- ether src foo : all ethernet packets where the source address is host foo
- arp net 128.3 : all arp packets to network 128.3
- tcp port 21 : all tcp packets to port 21
- udp portrange 7000-7009

https://linux.die.net/man/7/pcap-filter

PCap Filter examples

- Examples:

host foo and not port ftp and not port ftp-data

Any traffic from/to the host name foo except traffic on ftp and ftp-data ports

gateway snup and (port ftp or ftp-data)

All FTP traffic through the gatewap snup

gateway snup and ip[2:2] > 576

All gateway traffic through snup with size greater than 576 bytes

ether[0] & 1 = 0 and ip[16] >= 224

IP broadcast or multicast traffic that were not sent via Ethernet broadcast/multicast
Byte 0 LSB 1 in Ethernet frame indicates a broadcast
IP broadcast have destination address 224.0.0.0 to 239.255.255.255

https://linux.die.net/man/7/pcap-filter

PCap Filter examples

- Examples:

```
host helios and \( hot or ace \)
```

```
ip and not net localnet
```

```
tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net localnet
```

https://linux.die.net/man/7/pcap-filter

PCap Filter examples

- Examples:

  `host helios and \( hot or ace \)`

  Any traffic from the host name helios and with destination hot or ace will be logged.

  `ip and not net` *`Localnet`*

  Traffic that is not sourced or destined for local hosts

  `tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net` *`Localnet`*

  start and end packets (the SYN and FIN packets) of each TCP conversation that involves a non-local host.

https://linux.die.net/man/7/pcap-filter

tcpdump (uses PCap library)

Output the BPF code for the input predicate

Filter IP and UDP packets

```
$ sudo tcpdump -p -ni eth0 -d "ip and udp"
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 5
(002) ldb      [23]
(003) jeq      #0x11            jt 4    jf 5
(004) ret      #65535
(005) ret      #0
```

Low level BPF output

```
$ sudo tcpdump -p -ni eth0 -ddd "ip and udp"|tr "\n" ","
6,40 0 0 12,21 0 3 2048,48 0 0 23,21 0 1 17,6 0 0 65535,6 0 0 0,
```

Packet Sniffing using 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.

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

```
struct pcap_pkthdr {
        struct timeval ts;
        bpf_u_int32 caplen;
        bpf_u_int32 len;
};
```

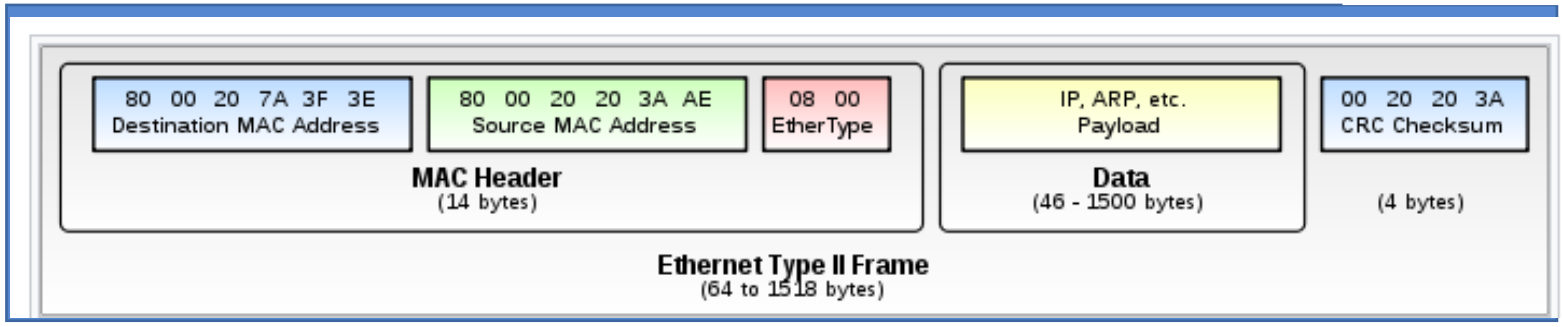
Invoke this function for every captured packet

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

Is filled with the packet Received. This contains the raw ICMP packet

Processing 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 Ethernet Header

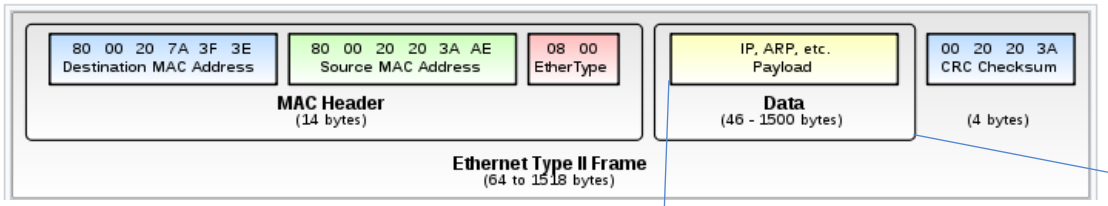
```
/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};

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.

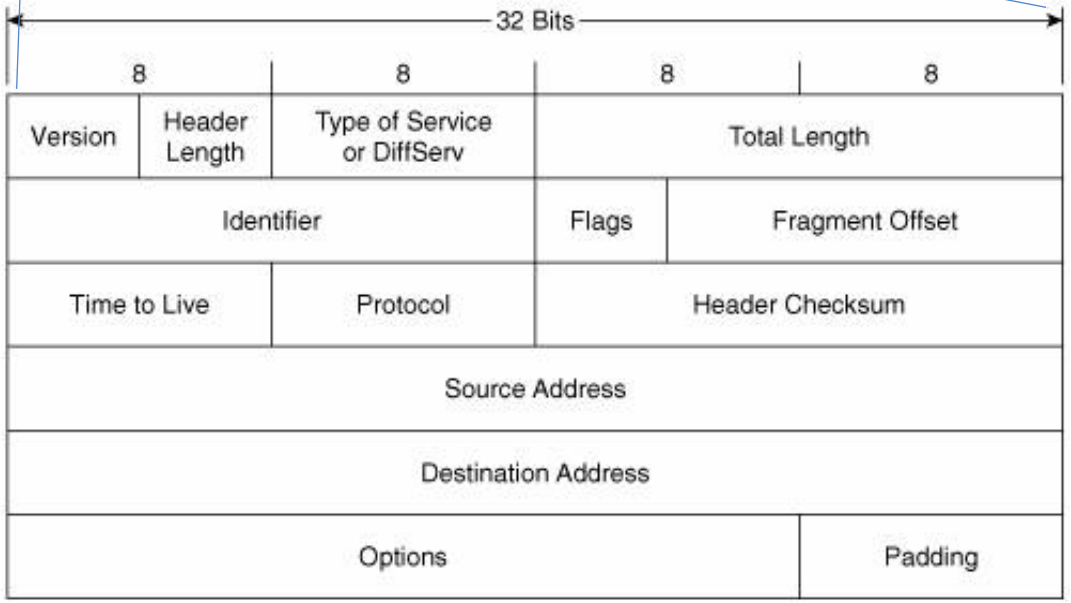
Now we can access the field of the structure

Processing IP Packet

*packet




```
/* 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
};
```



*(packet + sizeof(struct ethheader))

Processing 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 of 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

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: 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:

-luv: listen for incoming UDP packets, verbose

```
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.
- Example
  - Destination IP address can be set
  - Source IP address is not set:
    - Operating system, will automatically fill these fields before tranmitting the packet to the hardware
- Spoofing
  - Permits manipulation of critical fields in the packet headers
  - Can create unrealistic / bogus packets. For example:
    - Transmit a TCP packet with SYN and FIN bits turned on
    - The response from the receiver is unpredictable and depends on the OS
  - Used in many network attacks like
    - TCP SYN Flooding, TCP session hijacking, DNS cache poisoning attack
    - Supplied information depends on the type of attack being carried out

Spoofing Tools

- Netwox
- Scapy
- Spoofing from first principles
  - Two Major Steps

  (1) constructing the packet in a buffer
  (this step is going to depend on the type of packet)

  (2) sending the packet out

Constructing an ICMP Ping Packet

STEP 1

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));
```

Ping request (echo request)

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

Fill in the ICMP header fields

Constructing an ICMP Ping Packet

STEP 1

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);
```

Sending Spoofed Packets Using Raw Sockets

STEP 2

```
/*****************************************************************
  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

```
/*************************************************************
   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!
```

MAC Address Spoofing?

How to spoof MAC addresses?
Needs hardware and OS support

```
# ip link set dev eth0 down

# ip link set dev eth0 address XX:XX:XX:XX:XX:XX

# ip link set dev eth0 up
```

MAC is restricted to local networks.
Thus MAC spoofing is only a problem with insider threats

Sniffing and Spoofing

Threat: Man in the middle attacks
Sniff a packet. Spoof the response

- Procedure
  - 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

Sniffing and Spoofing a UDP Example

```
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);
```

why *4?

Sniffing and Spoofing a UDP Example

```
    // 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);
}
```