Network & Web Security

Packet Sniffing and Spoofing (Part I)

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

Today

Announcements

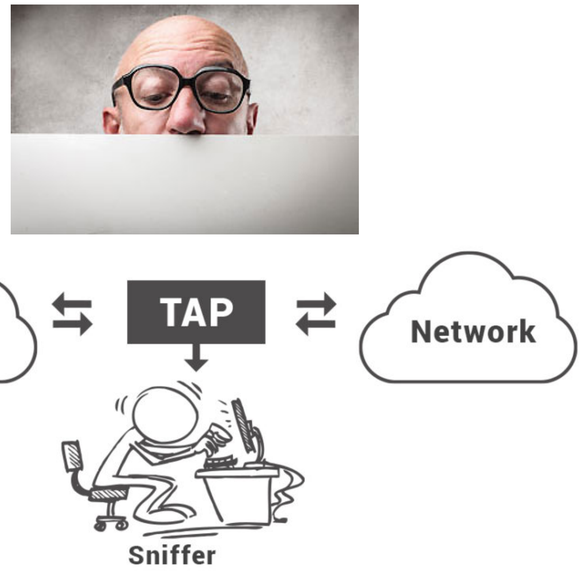
- Lab 06 Due Thursday

Goals & Learning Objectives

0. Tying up loose ends w/ SQL Injections & countermeasures

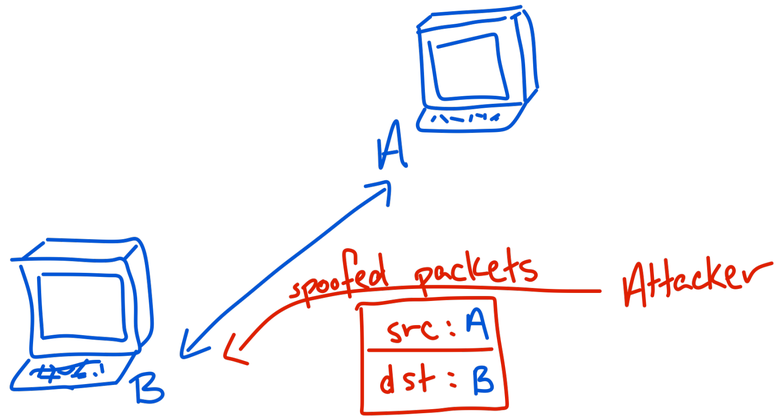
1. Network Basics

2. Packet *sniffing*



How can we build a sniffer?

3. Packet *spoofing*



How can we spoof packets?

Networks in a Nutshell

Networks can look different, but lots of concepts generalize.
E.g., Ethernet, Wi-Fi…

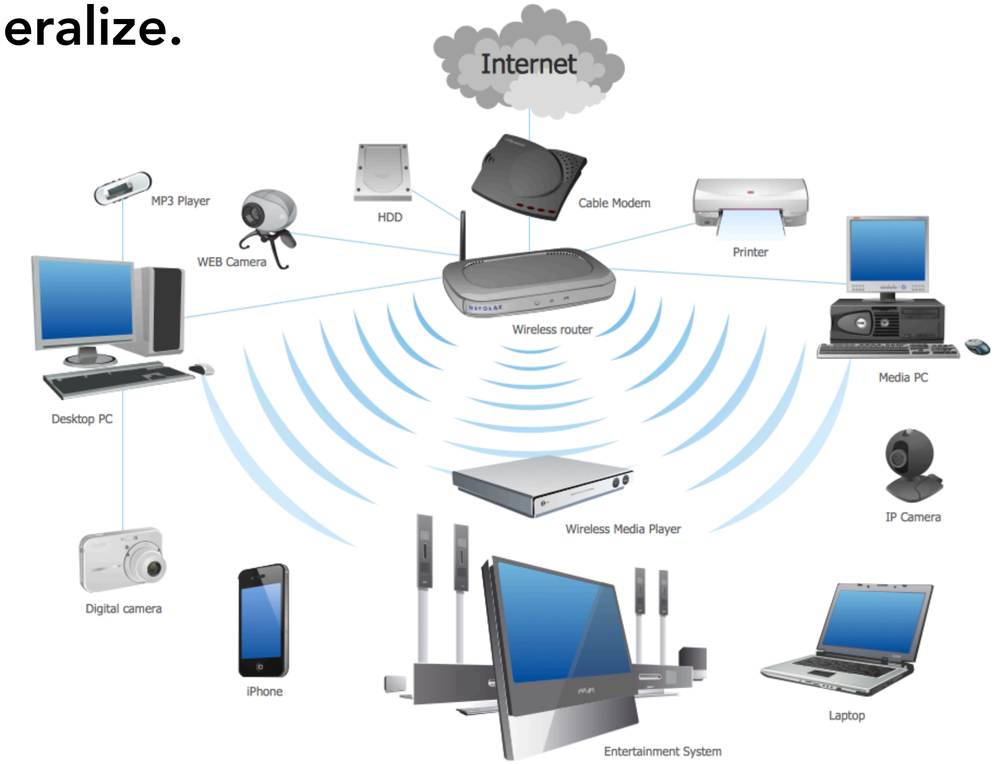
…all devices share the ***same medium***

…connect to a network via a
Network Interface Card (NIC)

…each NIC has a
Medium Access Control (MAC) address

…every NIC on the network
"hears" all the frames "on the wire" (or "in the air")

… NIC checks ***destination (dst) address****;*
"pass up" packets that match the NICs MAC address
("drop" other packets)



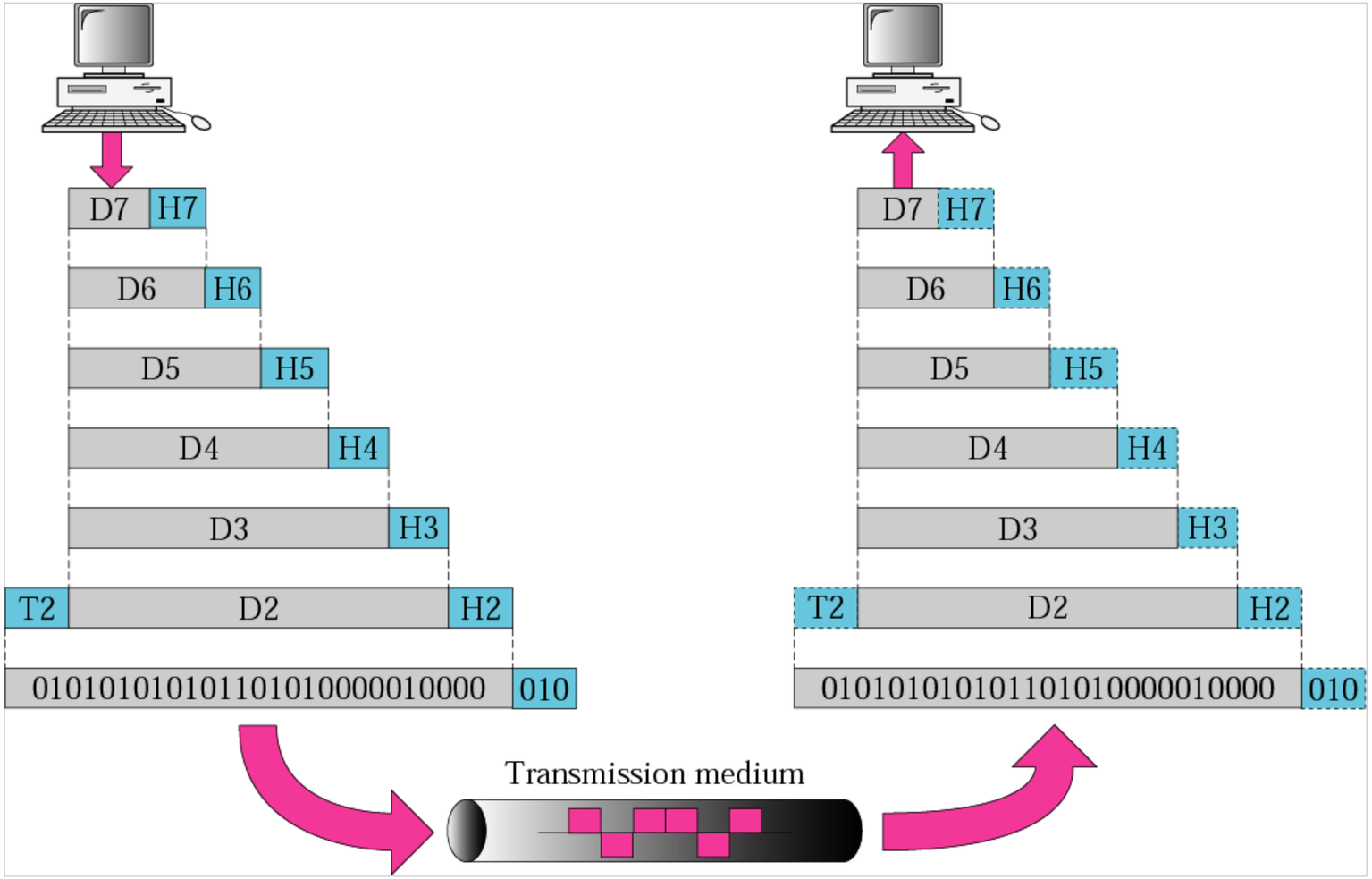
Ethernet (802.3) Frame Format

7 bytes	1 byte	6 bytes	6 bytes	2 bytes	42 to 1500 bytes	4 bytes	12 bytes
Preamble	Start of Frame Delimiter	**Destination MAC Address**	**Source MAC Address**	Type	Data (payload)	CRC	Inter-frame gap

—https://thenetworkseal.files.wordpress.com/2015/05/ethernet_wifi_frames.jpg
—https://www.conceptdraw.com/How-To-Guide/picture/network-icon/Computer-Network-Diagrams-Wireless-Router-Network-Diagram.png

Networks in a Nutshell

Packets are ***encapsulated*** in various protocol layers; each layer has a ***header*** and ***payload***



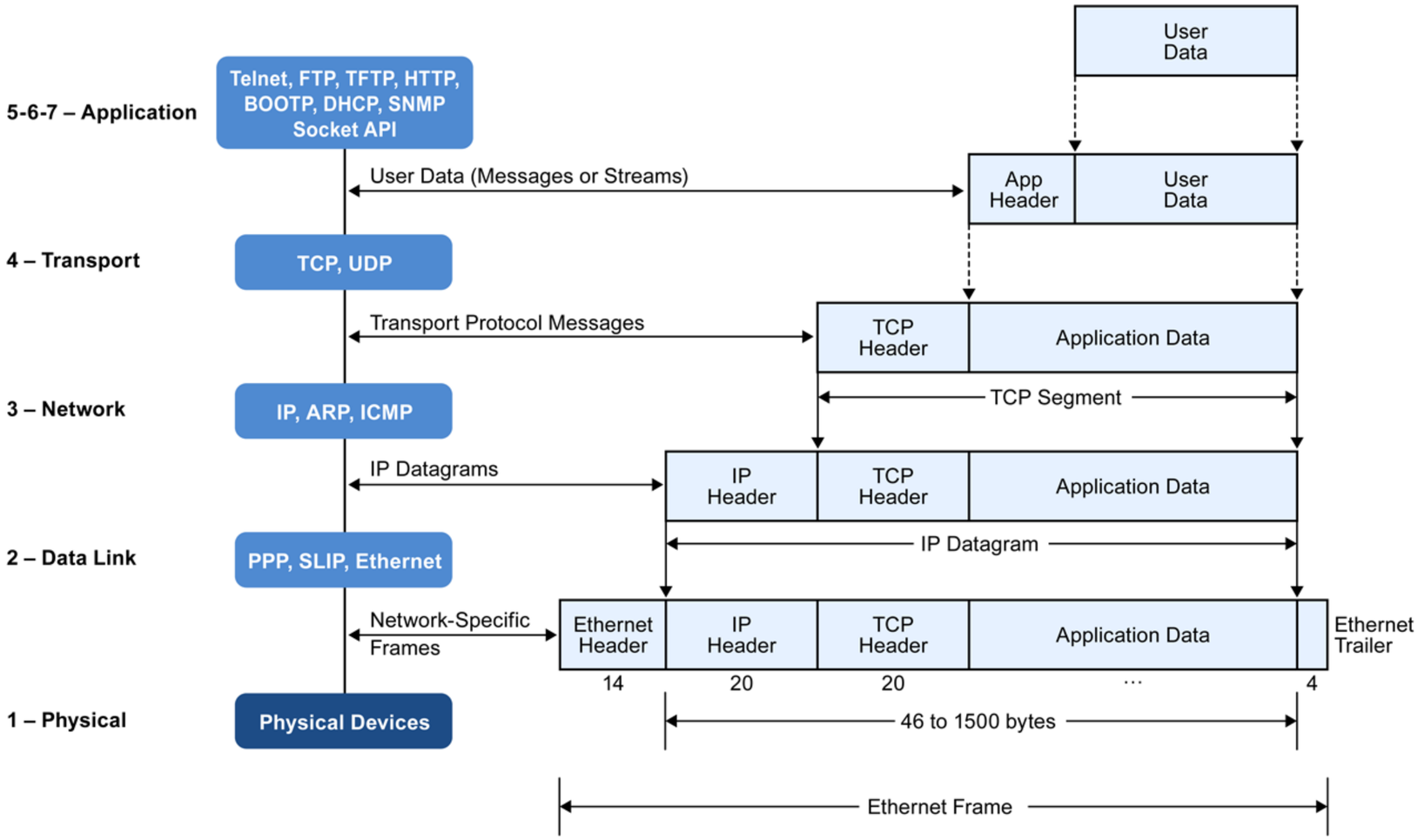
—https://computer---networking.blogspot.com/2013/04/osi-model-understanding-seven-layers-of.html

Networks in a Nutshell

OSI model…

Application (7)	Provides services directly to user applications. Because of the potentially wide variety of applications, this layer must provide a wealth of services. Among these services are establishing privacy mechanisms, authenticating the intended communication partners, and determining if adequate resources are present.
Presentation (6)	Performs data transformations to provide a common interface for user applications, including services such as reformatting, data compression, and encryption.
Session (5)	Establishes, manages, and ends user connections and manages the interaction between end systems. Services include such things as establishing communications as full or half duplex and grouping data.
Transport (4)	Insulates the three upper layers, 5 through 7, from having to deal with the complexities of layers 1 through 3 by providing the functions necessary to guarantee a reliable network link. Among other functions, this layer provides error recovery and flow control between the two end points of the network connection.
Network (3)	Establishes, maintains, and terminates network connections. Among other functions, standards define how data routing and relaying are handled.
Data-Link (2)	Ensures the reliability of the physical link established at Layer 1. Standards define how data frames are recognized and provide necessary flow control and error handling at the frame level.
Physical (1)	Controls transmission of the raw bitstream over the transmission medium. Standards for this layer define such parameters as the amount of signal voltage swing, the duration of voltages (bits), and so on.

Networking in practice…



—https://micrium.com/wp-content/uploads/2014/03/OSI-Seven-Layer-Model.png
—https://computer---networking.blogspot.com/2013/04/osi-model-understanding-seven-layers-of.html

Packet Sniffing

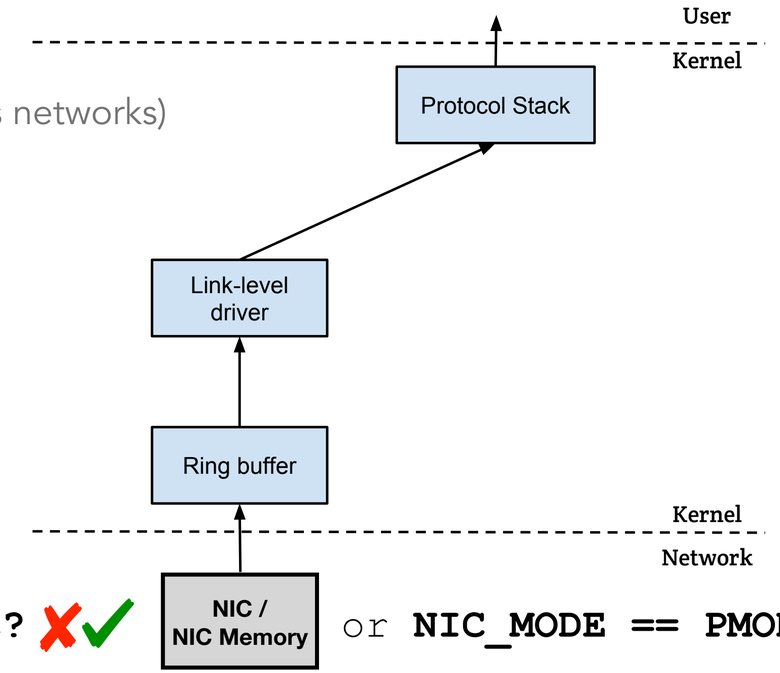
So… How Do I Access *ALLLLL* of the Network Traffic…?

Promiscuous Mode

- Frames that are not destined to a given NIC are normally discarded
- When operating in promiscuous mode,
  the 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

NOTE: In Wi-Fi, this is called **"Monitor Mode"** (Only applicable to wireless networks)

- *No association w/ an AP*
- *Capture packets only on a specific channel…*



Do we really need to sniff ***ALL*** packets?

Not necessarily! Interesting ones are
e.g., **TCP**, **UDP**, **DNS**, ...

We can **filter** out all the uninteresting packets

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,
};
```

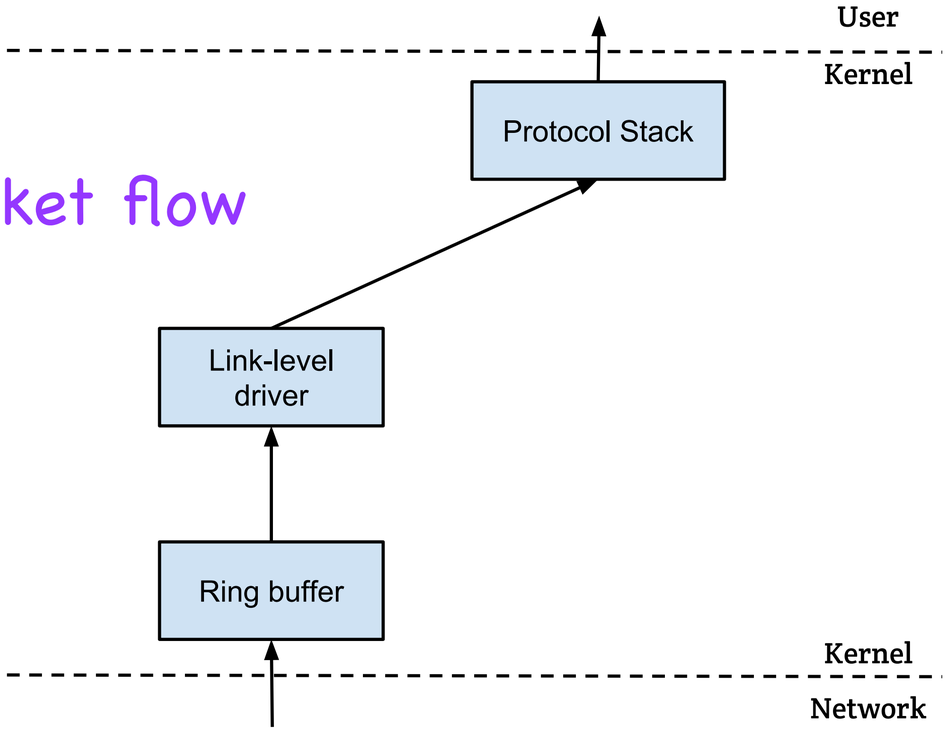
```
setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf))
```

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

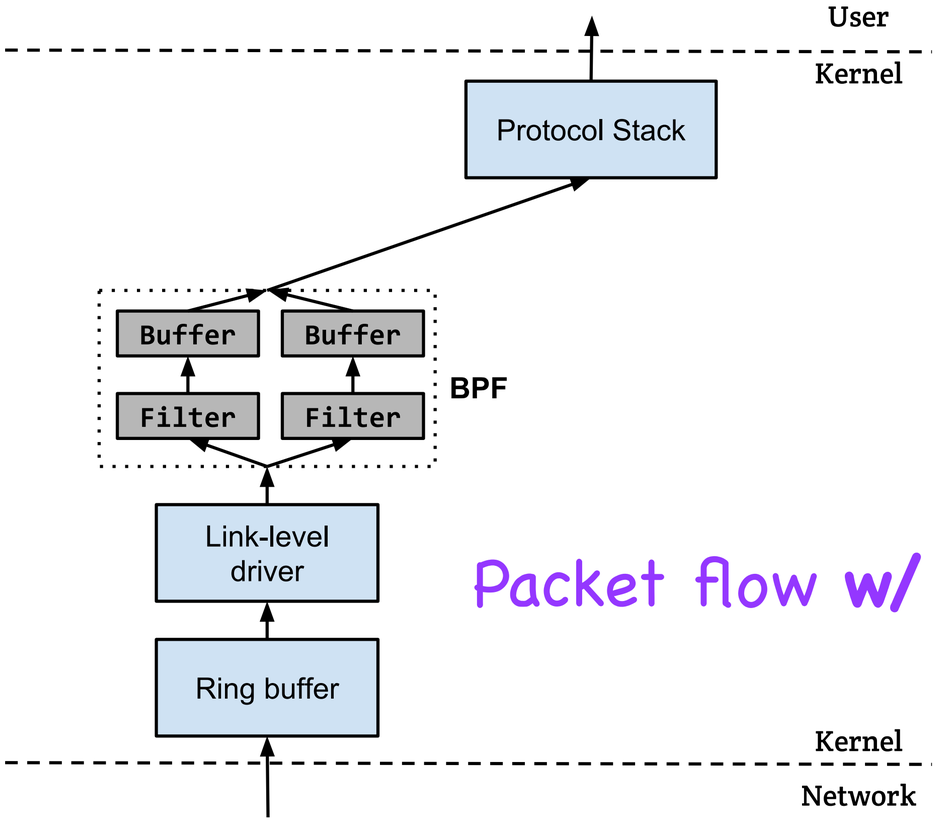
BSD Packet Filter (BPF)

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

- A compiled BPF 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.



Packet flow



Packet flow w/ filter

Receiving Packets Using **Sockets**

- Packet sniffing describes the process of ***capturing live data*** as they flow across a network
- Example: *how computers receive packet…*

Create a socket

Try: `man 7 socket`

```
NAME
       socket - Linux socket interface

SYNOPSIS
       #include <sys/socket.h>

       sockfd = socket(int socket_family, int socket_type, int protocol);
```

```c
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);
}
```

udp_server.c

Receiving Packets Using **Sockets**

- Packet sniffing describes the process of ***capturing live data*** as they flow across a network
- Example: *how computers receive packet…*

Create a socket

Provide information about server

Try: `man 7 ip`

Try: `man bind`

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

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

```
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);
}
```

udp_server.c

Receiving Packets Using **Sockets**

- Packet sniffing describes the process of ***capturing live data*** as they flow across a network
- Example: *how computers receive packet…*

Create a socket

Provide information about server

Receive packets

```
NAME
       recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       ssize_t recv(int sockfd, void *buf, size_t len, int flags);

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

```c
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);
}
```

udp_server.c

Packet Sniffing Using **Raw Sockets**

Create a raw socket

Capture copies of all types of packets

```
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));

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

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

sniff_raw.c

Enable promiscuous mode

(capture all packets, even those not destined for our machine)

Wait for packets

As we dig deeper, this approach gets complicated...

Limitations of Using Sockets Directly…

- This program is not portable across different operating systems…
- Setting filters is not easy…
- The program has no optimization to improve performance…

- As a result, the **p**acket **cap**ture (**pcap**) library was created
  - `pcap` still uses raw sockets internally, but its API is standard across all platforms.
    - OS specifics are hidden by PCAP's implementation.
    - Implementations: Linux **(libpcap)**, Windows **(WinPcap)**
  - Allows programmers to specify filtering rules using human readable ***Boolean expressions.***

E.g.,…

```
dst host 10.0.2.5
src host 10.0.2.6
host 10.0.2.6 and src port 9090
proto tcp
```

Packet Sniffing Using the **pcap** API

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

Also try…

```
dst host 10.0.2.5
src host 10.0.2.6
host 10.0.2.6 and src port 9090
proto tcp
```

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
```

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");
}
```

In reality, we want to do some real processing over interesting packets…

sniff.c

Processing Captured Packet: **Ethernet Header**

sniff_improved.c

```
/* 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**

sniff_improved.c

```
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.

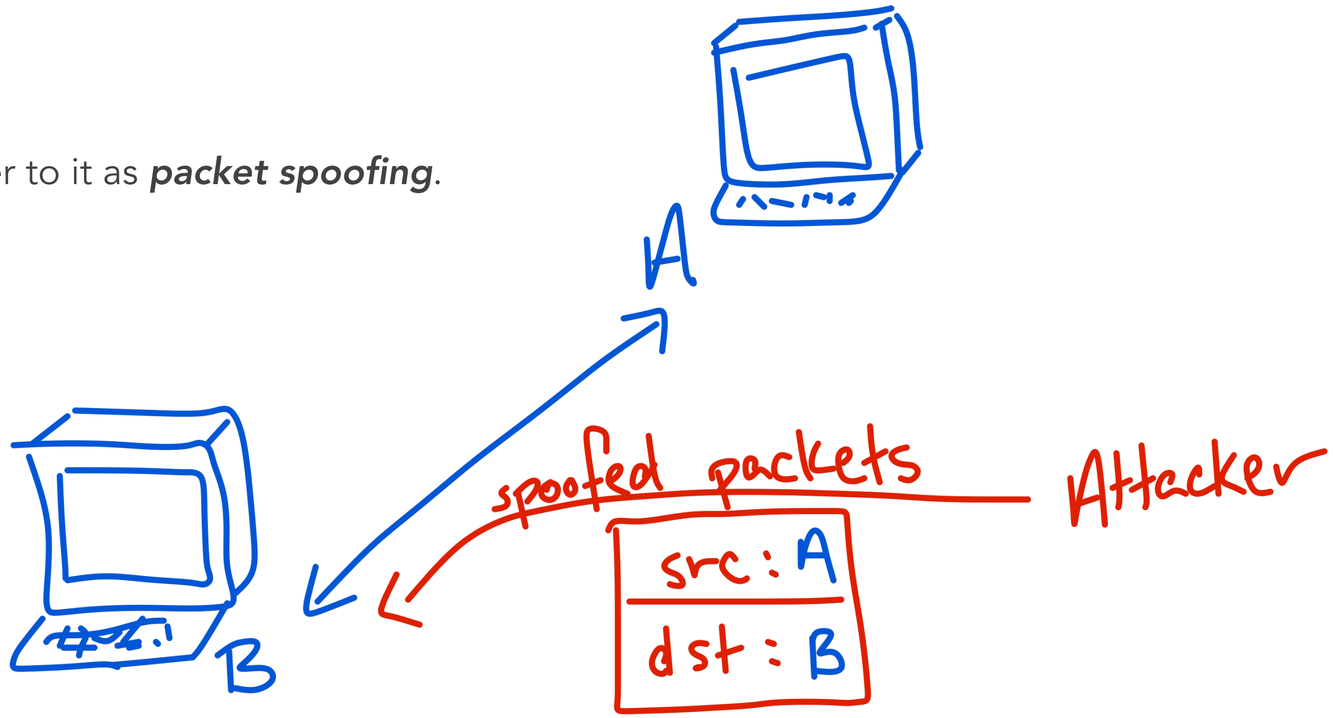
Further Processing Captured Packet

- 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.



Sending Packets *(The Normal Way… Without Spoofing)*

udp_client.c

```
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
```

Unfortunately, we don't have much control over header fields (e.g., src ip, pkt length)…

Spoofing Packets Using **Raw Sockets**

- There are two major steps to packet spoofing w/ raw sockets:
  - Constructing the packet
  - Sending the packet

Goal: Send an ICMP Echo (i.e., "ping") with a spoofed src IP address

Spoofing Packets Using **Raw Sockets** *(cont.)*

Goal: Send an ICMP Echo (i.e., "ping") with a spoofed src IP address

- There are two major steps to packet spoofing w/ raw sockets:
  - Constructing the packet
  - ***Sending the packet***

Sending the Packet

```
/*****************************************************************
  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);
}
```

spoof.c

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 Using **Raw Sockets** *(cont.)*

Goal: Send an ICMP Echo (i.e., "ping") with a spoofed src IP address

- There are two major steps to packet spoofing w/ raw sockets:
  - ***Constructing the packet***
  - Sending 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));
```

spoof_icmp.c

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

Fill in the ICMP header fields

Spoofing Packets Using **Raw Sockets** *(cont.)*

Goal: Send an ICMP Echo (i.e., "ping") with a spoofed src IP address

- There are two major steps to packet spoofing w/ raw sockets:
  - ***Constructing the packet***
  - Sending 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));
```

spoof_icmp.c

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 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.
 *********************************************************/

...... /* 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);
```

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

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 ***Then*** Spoofing

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

- Procedure (e.g., UDP)
  - Capture the packets of interests
  - Make a copy from the captured packet
  - Replace the UDP data field with new contents + swap the source and destination fields
  - Send out the spoofed reply

NOTE:

C variants of sniffing+spoofing code are covered in the text.
In the lab you will use Scapy/Python,
so we will focus our attention here now...

First, Revisiting Packing Sniffing Using Scapy

Example: Construct packets

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

a = IP()
a.show()

###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

ippkt.py

Example: Sniff for ICMP packets

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

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

def print_pkt(pkt):
    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)
```

sniff.py

First, Revisiting Spoofing ICMP & UDP Using Scapy

Example: Spoof ICMP packets

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

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

icmp_spoof.py

Example: Spoof UDP packets

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

print("SENDING SPOOFED ICMP 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)
```

udp_spoof.py