

Aspects of Network Sniffing

By kay

Abstract

This writing discusses network packet sniffers. starting with some basic hardware information to designing and programming ones. Hope you find this somewhat useful.

Network basics

Some would argue that sniffing is an old issue: yes, it is. Nowadays we have asymmetrical ciphers, which are supposed to eliminate this danger. On the other hand every day thousands of shell accounts, credit card numbers and other important data leak out. Also sniffers are always good utilities for locating network protocol problems and security monitoring (IDS).

Network design, hardware and software

Some aspects of the local computer networks' design and implementation allow "sniffing" communications between different nodes on the network from all the rest being part of the same network segment. This is due to the IEEE802.3 CSMA/CD (Carrier Sense Multiple Access with Collision Detection) standard and especially its algorithm for avoiding collisions, which requires that a NIC (Network Interface Card) gets all packets from the network even those which are destined for other addresses. A collision occurs when two nodes try to "talk" simultaneously through the network. Because of hardware limitations this is impossible and causes a temporary pause for all the traffic. The algorithms themselves used to avoid these problems are different for each vendor, but that is beyond the scope of this article. In other words, at the very hardware level a NIC physically receives ALL data transmitted on its network segment.

This is pretty good, but isn't yet enough for a user-level program to read those packets from the network: usually the Operating System for each station reads only packets destined for itself and to special type of address called broadcast address (broadcast packets). Broadcast addresses are used when the caller doesn't know the address of the called node or it wants all nodes in the same network segment to receive the packet. These features are used in the DHCP and BOOTP protocols as well as some not-so-innocent Denial of Service attacks like Smurff or example. Sometimes the OS makes it difficult to read raw packets, and when you have a bunch of different OS's to deal with you face the problem.

Switched networks are partially immune to this problem as only two of all the segments will receive datagrams - the source and destination segments. On the other hand in networks with bus topology or connected via hubs all nodes would receive information as they are practically in one segment.

Devices and interfaces

In Unix each physical and logical network device is represented with an interface. For listing and configuring the interfaces on a host the 'ifconfig' command (on newer Linux systems this could be done with 'ip') is used:

```
$ /sbin/ifconfig
loLink encap:Local Loopback
inet addr:127.0.0.1Mask:255.0.0.0
UP LOOPBACK RUNNINGMTU:3924Metric:1
RX packets:249 errors:0 dropped:0 overruns:0 frame:0
TX packets:249 errors:0 dropped:0 overruns:0 carrier:0
Collisions:0

eth0Link encap:Ethernet HWaddr 00:AC:3B:71:1D:D0
inet addr:192.168.0.1Mask:255.255.255.0
MULTICAST PROMISCMTU:1500Metric:1
RX packets:5357 errors:0 dropped:0 overruns:0 frame:0
```

Aspects of Network Sniffing

By kay

```
TX packets:2397 errors:0 dropped:0 overruns:0 carrier:0
Collisions:0
Interrupt:12 Base address:0x420
```

```
ppp0Link encap:Point-to-Point Protocol
inet addr:192.168.0.100P-t-P:192.168.1.1Mask:255.255.255.255
POINTOPOINT NOARP MULTICASTMTU:1500Metric:1
RX packets:913 errors:1 dropped:0 overruns:0 frame:1
TX packets:920 errors:0 dropped:0 overruns:0 carrier:0
Collisions:0
```

In this case we are going to look at an example Linux system with one Ethernet card and a PPP interface. Notice the PROMISC flag on the eth0 interface – this indicated that eth0 receives ALL packets which reach the NIC, also it is said to be in promiscuous mode. Also note that some old network cards have to use promiscuous mode in order to receive broadcast packets. Lo stands for the loopback interface which provides us with access to our own host (the127/4 localnet, localhost, etc.).In some cases it makes sense to sniff even this interface since TCP connections and UDP messages are often used for data exchange, say between two modules of a program.

Please also note, that sniffing non-Ethernet devices such as PPP, SLIP or even the loopback is perfectly possible; a sniffer would catch all packets that go through that interface. The difference is that in an Ethernet we could "hear" what others are "talking".

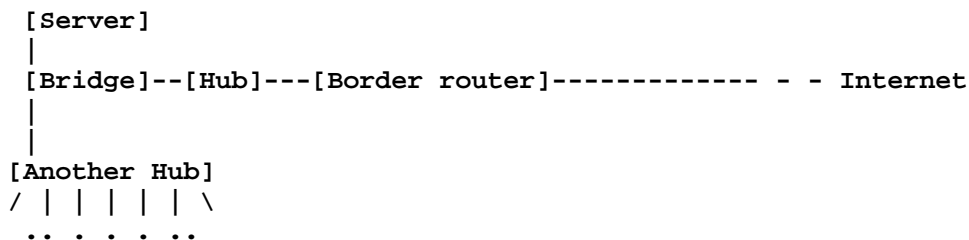
Preventing and detecting sniffers

There exist both hardware and software solutions for securing a local network. Clean design of the network topology, dividing the network into separate segments and isolating them from each other with switches, encrypting hub's, VPN (Virtual Private Networks) supporting encryption help reduce the risk of system crackers getting important information through network sniffing.

There is an excellent article on fooling poorly coded sniffers in Phrack Magazine, issue 54.

A computer program byL0pht Heavy Industries called AntiSniff is said to be able to remotely detect promiscuous interfaces by studying the network traffic and probing the IP stack of the machine. Also when probing for local promiscuous interfaces, one should not trust ifconfig as it might have been trojanized. Instead use a utility like lspromisc.c found below. Even this does not guarantee 100% that there are no hidden sniffers -see the notes below on Loadable Kernel Modules.

When building new networks engineers should avoid connecting many workstations and important servers on only one network segment like this:



This way all stations will be able to "hear" what's the neighbor department's server talking to our's through the bridge or what's Joe's email password being checked over the Internet. Instead try replacing the hubs with switches, separate the machines into groups and make VLAN for each of them, etc. etc. Remember that the most secure computers are the ones switched off.

Aspects of Network Sniffing

By kay

Also strong cryptography for all connections is always a Good Thing (tm). There are lots of free programs for implementing Virtual Private Networks, SSL tunnels, data encryption - such as FreeS/WAN, SSLeay, GnuPG. If you are still using telnet/rlogin/rsh for remote administration consider switching to SSH or OpenSSH (thanks to the OpenBSD folks!!).

Introduction to packet sniffing

As each Operating System provides its own way of accessing low level functions, it becomes hard for a programmer to write "portable" code, code that compiles without changes on a wide range of computers. There is the Berkley Packet Filter (BPF) in BSD, there is the PF_PACKET protocol family in Linux and the list grows as every major flavor of Unix has its own functions and mechanisms. The PCAP library developed in Lawrence Berkley National Laboratory provides a single and standardized API to all platforms it supports (it has even been ported to Win32, where you can use it in conjunction with a VxD device driver the same way you would use it under Unix), thus it is a wrapper of the existing system interfaces.

Operating Systems provide the applications with connection endpoints called sockets, which are used to send/receive stream of information. A TCP connection is identified by the two pairs IP_address:TCP_port. Though TCP/IP (and all the network protocols) uses packets for data transmissions and the kernel splits that stream to series of IP packets. Those packets may travel different routes to their destination; some of them may get lost and when a timeout exceeds will be retransmitted; some of them may get damaged because of hardware faults and so on. The IP stack of the OS makes this transparent to all applications. The TCP multiplexer on turn distributes the appropriate data to each socket so that different TCP connections do not get mixed.

When a sniffer is listening to the wire it will have to do its own filtering and sorting. We have to have a mini-IP stack for assembling the IP packets and then a mini-TCP-multiplexer. This is probably the most important part of the sniffer; that's why out there are so many of them, but so few are really good. Again: check out Phrack 54.

It is impossible to write a packet sniffer without good knowledge of all protocols and packets we are going to process. As most sniffers actually sniff TCP connections, we'll have to begin from the link layer frame (f.e. Ethernet), the IPv4 (usually) and then the TCP headers.

All the network protocols are separated into layers. An example TCP packet over Ethernet would look like this:

```
struct ethhdr eth;
struct iphdr ip;
struct tcphdr tcp;
[ data ]
```

Of course we can catch all the ICMP, UDP, IGMP, and non-TCP/IP protocols like IPX. Nowhere is how the structures for Ethernet, IP and TCP header are defined in Linux:

```
#define MAC_LEN 6

struct ethhdr {
u_char dst_addr[MAC_LEN]; /* Obviously - MAC addresses of sender
u_char src_addr[MAC_LEN];* and destination */
u_short protocol; /* Underlying protocol number */
};
```

Aspects of Network Sniffing

By kay

```
struct iphdr {
u_char ver_ihl;
u_char tos;
u_short total_len;
u_short id;
u_short frag_offset;
u_char ttl;
u_char protocol;
u_short checksum;
u_long src_addr;
u_long dst_addr;
};

struct tcphdr {
u_short src_port;
u_short dst_port;
u_long sequence;
u_long acq_seq;
u_short flags;
u_short window;
u_short checksum;
u_short urg_ptr;
};
```

Let's take a look at this rough algorithm of TCP sniffer written in C-pseudo code (I've always hated block diagrams, and you?):

```
while (we_are_willing_to_sniff) {
packet = read_raw_packet();

if (broken_packet(packet)) continue;

if (starts_new_connection(packet) && port_is_interesting(packet))
add_to_stack(connection(packet));

if (packet_is_part_of_tracked_connection(packet)) {
log(packet);

if (we_have_logged_enough(connection(packet)) ||
packet_closes_connection(packet))
remove_from_stack(connection(packet));
}
}
```

Broken_packet() is supposed to check if all checksums, addresses, header fields are valid.

Starts_new_connection() will check whether packet initiates a new connection. In the case with TCP/IP this is a SYN/SYN+ACK TCP packet. This is the other point you have to be very careful with Sequence numbers, checksums, etc.

The model of your sniffer will depend on what information you want to examine. The task is much simpler if you are tracking connection-less protocols such as UDP.

Aspects of Network Sniffing

By key

Example of Linux SOCK_PACKET usage

Under Linux accessing the link-layer data is done through a special type of socket - SOCK_PACKET. When we want to get all the data that passes through an interface, we have to set the promiscuous mode for it first. This is done through the ifreq structure:

```
struct ifreq
{
#define IFNAMSIZ 16
union
{
char ifrn_name[IFNAMSIZ];
} ifr_ifrn;

union {
struct sockaddr ifru_addr;
struct sockaddr ifru_dstaddr;
struct sockaddr ifru_broadaddr;
struct sockaddr ifru_netmask;
struct sockaddr ifru_hwaddr;
short ifru_flags;
int ifru_ivalue;
int ifru_mtu;
struct ifmap ifru_map;
char ifru_slave[IFNAMSIZ];
char ifru_newname[IFNAMSIZ];
char * ifru_data;
} ifr_ifru;
};

#define ifr_name ifr_ifrn.ifrn_name /* interface name */
#define ifr_hwaddr ifr_ifru.ifru_hwaddr /* MAC address */
#define ifr_addr ifr_ifru.ifru_addr /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr /* other end of p-p lnk */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_netmask ifr_ifru.ifru_netmask /* interface net mask */
#define ifr_flags ifr_ifru.ifru_flags /* flags */
#define ifr_metric ifr_ifru.ifru_ivalue /* metric */
#define ifr_mtu ifr_ifru.ifru_mtu /* mtu */
#define ifr_map ifr_ifru.ifru_map /* device map */
#define ifr_slave ifr_ifru.ifru_slave /* slave device */
#define ifr_data ifr_ifru.ifru_data /* for use by interface */
#define ifr_ifindex ifr_ifru.ifru_ivalue /* interface index */
#define ifr_bandwidth ifr_ifru.ifru_ivalue /* link bandwidth */
#define ifr_qlen ifr_ifru.ifru_ivalue /* Queue length */
#define ifr_newname ifr_ifru.ifru_newname /* New name */
```

and with the SIOCGIF_FLAGS and SIOCSIF_FLAGS (Socket I/O Control {Get,Set} Interface Flags) ioctl() calls. The only required parameter is ifr_name, others depend on what we want to do. It is possible to a list of all interfaces on a system with the ifconf structure:

```
struct ifconf
{
int ifc_len;
union
```

Aspects of Network Sniffing

By kay

```
{
char *ifcu_buf;
struct ifreq*ifcu_req;
} ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf
#define ifc_req ifc_ifcu.ifcu_req
```

and the SIOCGIFCONF call. In ifc_len is passed the size of ifcu_buf which receives the ifreq structures of all interfaces (or as much as might be stuck in to it). In ifc_len is modified there after to match the number of interfaces returned. This is necessary when we don't know what exactly interface we want to sniff and need to get information about all available ones-names, addresses, netmasks. We also might want to sniff interfaces with strange names as many software VPN implementations create a virtual interface and redirect all data from it through encrypted tunnel. This way we can get all the data just before it gets encoded.

When we want to sniff a specific interface, we use the bind() call, almost the same way we do with normal sockets. Using the sockaddr structure:

```
struct sockaddr {
unsigned short sa_family;
char sa_data[14];
};
```

we specify interface name as a null-terminated string into sa_data.

```
--<sockpacket.c>-----
/* Copyright (C) 1999 kay@phreedom.org; All rights reserved */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>
#include <linux/if.h>

#include "pdump.h"

int
main(int argc, char **argv)
{
struct ifreq ifr; /* Linux interface request control structure */
short ifr_flags_orig; /* Initial flags if interface */
int sockfd; /* Socket descriptor */
u_char sp[2000];
int err;

printf("Example of non-portable packet sniffer for Linux\n");

/* We want only Ethernet frames containing IP data */
```

Aspects of Network Sniffing

By kay

```
sockfd = socket(PF_PACKET, SOCK_PACKET, htons(ETH_P_IP));
if (sockfd < 0) {
perror("socket");
exit(1);
}

/* Make the interface promiscuous */
strcpy(ifr.ifr_name, INTERFACE);
err = ioctl(sockfd, SIOCGIFFLAGS, &ifr);
if (err < 0) {
perror("SIOCGIFFLAGS");
exit(1);
}
ifr_flags_orig = ifr.ifr_flags;
ifr.ifr_flags |= IFF_PROMISC;
err = ioctl(sockfd, SIOCSIFFLAGS, &ifr);
if (err < 0) {
perror("SIOCSIFFLAGS");
exit(1);
}

/* Read one packet */
err = read(sockfd, &sp, sizeof(sp));
if (err < 0) {
perror("read");
exit(0);
}

/* Dump what we caught */
printf("Dumping %i bytes:\n", err);
dump_eth((struct ethhdr *) &sp);
dump_ip((struct iphdr *) &sp+14L);
dump_hex((void *) &sp, err, 2, 16);
dump_ascii((void *) &sp, err, 16);
printf("\n\n");

/* Restore original interface flags */
ifr.ifr_flags = ifr_flags_orig;
if (ioctl(sockfd, SIOCSIFFLAGS, &ifr) < 0) {
perror("SIOCSIFFLAGS");
exit(1);
}
close(sockfd);

return EXIT_SUCCESS;
}

/* eof */
-</sockpacket.c>-----

-<getifconf.c>-----
/* Copyright 1999 Kay <kay@phreedom.org>. All rights Reserved */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

Aspects of Network Sniffing

By kay

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if.h>
#include <linux/if_ether.h>

int main()
{
    struct ifconf ifc;
    struct ifreq ifr_x[20];
    int sockfd, err;

    sockfd = socket(PF_PACKET, SOCK_PACKET, 0);
    if (sockfd < 0) {
        perror("socket");
        exit(1);
    }

    ifc.ifc_len = 20 * sizeof(struct ifreq);
    ifc.ifc_req = ifr_x;
    err = ioctl(sockfd, SIOCGIFCONF, &ifc);
    perror("ioctl");
    printf("retrieved info for %i interface(s)\n",
           ifc.ifc_len / sizeof(struct ifreq));
    for (err = 0; err < ifc.ifc_len / sizeof(struct ifreq); err++)
        printf("%s\n", ifr_x[err].ifr_name);
    return EXIT_SUCCESS;
}

/* eof */
-</getifconf.c>-----
```

Libpcap example

The following program does the same as the previous example. Let's first take a look at the code, as it is really simple, and then make some clarifications.

```
-<libpcap.c>-----
/* Portable packet sniffer example - needs libpcap in order to compile
 * Copyright (c) 1999 kay@phreedom.org; All rights reserved */

#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ether.h>

#include "pdump.h"

int main(int argc, char **argv)
{
    pcap_t *pcap;           /* PCAP descriptor */
    u_char *packet;        /* Our newly captured packet */
    struct pcap_pkthdr pkthdr; /* PCAP packet information structure */
```

Aspects of Network Sniffing

By kay

```
printf("Example of portable packet sniffer using Libpcap\n");

/* Obtain a descriptor for interface, capture no more than
 * 8192 octets, set interface to promiscuous mode, 1000 milliseconds
 * read timeout, No buffer for error messages */
pcap = pcap_open_live(INTERFACE, 8192, 1, 1000, NULL);
if (pcap == NULL) {
    perror("pcap_open_live");
    exit(EXIT_FAILURE);
}

/* Get the next packet from the queue */
packet = (u_char *) pcap_next(pcap, &pkthdr);

if (packet != NULL) {
    packet += OFFSET;
    /* Dump the packet in various forms */
    printf("Dumping %i bytes:\n", pkthdr.caplen);
    dump_eth((struct ethhdr *) packet);
    dump_ip((struct iphdr *) packet);
    dump_hex((void *) packet, pkthdr.caplen, 2, 16);
    dump_ascii((void *) packet, pkthdr.caplen, 16);
    printf("\n\n");
} else {
    pcap_perror(pcap, "pcap_next returned NULL");
}

/* Enough for now ... */
pcap_close(pcap);

return EXIT_SUCCESS;
}

/* eof */
-</libpcap.c>-----
```

Pcap_open_live() returns a pcap descriptor (or error if it is impossible). It takes 5 arguments:

char * device - Interface/device name (asciiz)

int snaplen- Maximum number of bytes to return. That is, if the packet size exceeds snaplen you will receive only the first snaplen bytes.

int promisc- Indicates whether to put the interface in promiscuous mode or not. Note, however, that the interface could already be in promiscuous mode before the call.

int to_ms- Read timeout (milliseconds)

char * ebuf- If non-NULL Libpcap would write there a text explanation if an error occurs PCAP_ERRBUF_SIZE characters at most). See also pcap_perror().

Obviously the main thing here is the pcap_next() function which is declared as:

```
const u_char *pcap_next(pcap_t *, struct pcap_pkthdr *);
```

Aspects of Network Sniffing

By kay

and returns the next packet from the queue. That is, it replaces the read() from a SOCK_PACKET socket. The pcap_pkthdr structure contains some information about this newly captured packet.

```
struct pcap_pkthdr {
struct timeval ts; /* time stamp */
    bpf_u_int32 caplen; /* length of portion present */
    bpf_u_int32 len; /* length of this packet (off wire) */
};
```

Caplen is the size of the packet returned. In case the real length exceeds snaplen it would be equal to snaplen and len - the packet real length. Processing the packet from here on is absolutely the same.

BPF Packet filter programs

The Berkeley Packet Filter allows us to specify a filter program which will select only packets we are interested in from all the network traffic. A program consists of array of BPF instructions "executed" on a virtual filter machine. Instructions are similar to assembly languages with some extra functionality. This is an extremely powerful mechanism but building such programs isn't always easy enough. This is an example straight out from the manpage, which accepts only IP packets between 128.3.112.15 and 128.3.112.35:

```
struct bpf_insn insns[] = {
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 26),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 30),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 30),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
BPF_STMT(BPF_RET+BPF_K, (u_int)-1), BPF_STMT(BPF_RET+BPF_K, 0),
};
```

That's why the good guys at LBL have created a nice language for testing network packets and a compiler which makes BPF programs from it. There is versatile documentation on this language in tcpdump(8). Let's take a look at this example:

```
-<pfilter.c>-----
/* Packet filter example
 * Copyright (c) 1999 kay@phreedom.org; All rights reserved */

#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ether.h>

#include "pdump.h"

int main(int argc, char **argv)
{
pcap_t *pcap;          /* PCAP descriptor */
u_char *packet;       /* Our newly captured packet */
struct pcap_pkthdr pkthdr; /* PCAP packet information structure */
```

Aspects of Network Sniffing

By kay

```
struct bpf_program fp; /* Structure to hold the compiled prog */
char pfprogram[] = "ip host 128.3.112.15 and 128.3.112.35";

printf("Example of portable packet sniffer using Libpcap\n");

/* Obtain a descriptor for interface, capture no more than
 * 8192 octets, set interface to promiscuous mode, 1000 milliseconds
 * read timeout, No buffer for error messages */
pcap = pcap_open_live(INTERFACE, 8192, 1, 1000, NULL);
if (pcap == NULL) {
pcap_perror("pcap_open_live");
exit(EXIT_FAILURE);
}

/* Compile and set the filter program */
if (pcap_compile(pcap, &fp, pfprogram, 1, 0x0) == -1) {
pcap_perror(pcap, "pcap_compile");
exit(EXIT_FAILURE);
}
if (pcap_setfilter(pcap, &fp) == -1) {
pcap_perror(pcap, "pcap_setfilter");
exit(EXIT_FAILURE);
}

/* Get the next packet from the queue */
packet = (u_char *) pcap_next(pcap, &pkthdr);

if (packet) {
/* Dump the packet in various forms */
printf("Dumping %u bytes:\n", pkthdr.caplen);
packet += OFFSET;
dump_eth((struct ethhdr *) packet);
dump_ip((struct iphdr *) packet);
dump_hex((void *) packet, pkthdr.caplen, 2, 16);
dump_ascii((void *) packet, pkthdr.caplen, 16);
printf("\n\n");
} else {
printf("Packet not captured because of filter\n");
}

/* Enough for now ... */
pcap_close(pcap);

return EXIT_SUCCESS;
}

/* eof */
-</pfilter.c>-----
```

The expression "ip host 128.3.112.15 and 128.3.112.35" is translated to a BPF program using `pcap_compile()` and then it is attached to the PCAP descriptor using `pcap_setfilter()`. Piece of cake compared to BPF's byte-code instructions.

Linux Socket Filter (or LSF) is the Linux implementation of BPF with some additions. Namely: it allows user programs to attach filters to any of their sockets and filter data to their liking; it is also simpler to attach/detach your filter. The filter programs themselves need no change to work with it.

Aspects of Network Sniffing

By kay

Loadable kernel modules

Loadable kernel modules (LKM) are were originally designed as a way to add new features to the Operating System without having it rebooted or otherwise stopping its normal operation. A LKM is able access the kernel's internal data structures and combined with different techniques to hide itself from the eyes of curious sysadmins this is an efficient way for trojanizing cracked systems.

Under Linux, kernel interfaces and module writing action to Linux internals consider reading "The Linux Kernel" and "Linux Kernel Module Programmer's Guide". This topic is widely discussed in Phrack 55.

Bibliography and additional files

Man pages: setsockopt(2), pcap(3), bpf(7)

RFC's: 791, 792, 793, 894

"TCP/IP Illustrated" by R. W. Stevens

UTSL: Linux kernel, libpcap

Libpcap, tcpdump, BPF and other fun stuff: ftp://ftp.ee.lbl.gov

Phrack Magazine: http://www.phrack.com

The paper on BPF from Usenix'93 (also from LBL's site)

Pdump.c includes some trivial routines for displaying Hex/ASCII/IP/Ethernet dumps on a terminal. Linux-specific examples have been compiled and tested on Debian2.1 and 2.2 GNU/Linux systems (kernel 2.0.36/2.2.13 and glibc 2.0.7/2.1.2 respectively). PCAP-examples have been compiled and tested under OpenBSD 2.4 GENERIC as well, all three systems using libpcap version 0.4.

```
-<pdump.h>-----
/* Packet dumping routines, Copyright (c) 1999 Kay <kay@PHREEDOM.ORG> */

void dump_eth(struct ethhdr *);
void dump_ip(struct iphdr *);
void dump_hex(void *, u_long, u_long, u_long);
void dump_ascii(void *, u_long, u_long);

/* eof */
-</pdump.h>-----

-<pdump.c>-----
/* Packet dumping routines, Copyright (c) 1999 Kay <kay@PHREEDOM.ORG> */

#include <stdio.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/ether.h>

void dump_eth(struct ethhdr *eth)
{
int cnt;

printf("\th_dest=");
```

Aspects of Network Sniffing

By key

```
for (cnt = 0; cnt < ETH_ALEN; cnt++)
    printf(" %X", eth->h_dest[cnt]);
printf("\n\th_source =");
for (cnt = 0; cnt < ETH_ALEN; cnt++)
    printf(" %X", eth->h_source[cnt]);
printf("\n\th_proto= %X;\n", eth->h_proto);
fflush(stdout);
}

void dump_ip(struct iphdr *ip)
{
    struct protoent *pp;
    struct in_addr ia;

    printf("\th_l= %X;\n", ip->ihl);
    printf("\tversion= %X;\n", ip->version);
    printf("\ttos= %X;\n", ip->tos);
    printf("\ttot_len= %X;\n", ip->tot_len);
    printf("\tid = %X;\n", ip->id);
    printf("\tfrag_off = %X;\n", ip->frag_off);
    printf("\tttl= %X;\n", ip->ttl);

    printf("\tprotocol = %X;", ip->protocol);
    pp = getprotobyname(ip->protocol);
    if (pp == NULL) printf("\n"), perror(NULL);
    else printf("\t(%s)\n", pp->p_name);

    printf("\tcheck = %X;\n", ip->check);
    ia.s_addr = ip->saddr;
    printf("\tsaddr = %X;\t(%s)\n", ip->saddr, inet_ntoa(ia));
    ia.s_addr = ip->daddr;
    printf("\tdaddr = %X;\t(%s)\n", ip->daddr, inet_ntoa(ia));
    fflush(stdout);
}

/* Its obvious: *data, how many octets, interval of spaces,
interval of '\n'-s */
void dump_hex(void *bare, u_long octets, u_long int_sp, u_long int_nl)
{
    u_long s;
    u_long spc=0, nlc=0;
    char *buf = (char *) bare;

    for (s=0; s<octets; s++) {
        if ((u_char)buf[s]<0x10) printf("0");
        printf("%X", (u_char)buf[s]);
        if (++spc==int_sp) printf(" "), spc=0;
        if (++nlc==int_nl) printf("\n"), nlc=0;
        fflush(stdout);
    }
}

int is_printable(char c)
{
    if ((c >= '1') && (c <= '0')) return 1;
    if ((c >= 'A') && (c <= 'Z')) return 1;
}
```

Aspects of Network Sniffing

By key

```
    if ((c >= 'a') && (c <= 'z')) return 1;
    return 0;
}

void dump_ascii(void *bare, u_long octets, u_long int_nl)
{
    u_long s;
    u_long nlc=0;
    char *buf = (char *) bare;

    for(s=0; s<octets; s++) {
        printf("%c", is_printable(buf[s])?buf[s]:'.');
        if (++nlc==int_nl) printf("\n"), nlc=0;
        fflush(stdout);
    }
}

/* eof */
- </pdump.c>-----

- <Makefile>-----
# Makefile for examples (c) 1999 kay <kay@phreedom.org>
# Edit to suit your system.
# In case of problems when compiling on Linux 2.0 systems, try
# replacing AF_PACKET with AF_INET.

# Set interface to sniff. Some common offsets:
# Ethernet (eth0, le0)offset 0
# Loopback (lo, lo0, ...)offset 4
# PPP link (ppp0, ppp1, ...)offset 0
DEFS=-DINTERFACE=\"lo\" -DOFFSET=4

CC=cc
RM=rm -f
CFLAGS=-O2 -Wall -pipe $(DEFS)
LIBPCAP=-lpcap

default:
    @echo "Type one of:"
    @echo "make pcap -- build only PCAP examples"
    @echo "make all-- build PCAP and Linux-specific examples"

all: libpcap pfilter sockpacket lspromisc getifconf

pcap: libpcap pfilter

.c.o: $@
    $(CC) $(CFLAGS) -c $<

sockpacket: pdump.o sockpacket.o
    $(CC) $(CFLAGS) -o sockpacket sockpacket.o pdump.o

pfilter: pdump.o pfilter.o
    $(CC) $(CFLAGS) -o pfilter pdump.o pfilter.o $(LIBPCAP)

libpcap: libpcap.o pdump.o
    $(CC) $(CFLAGS) -o libpcap libpcap.o pdump.o $(LIBPCAP)
```

Aspects of Network Sniffing

By kay

```
getifconf: getifconf.c
           $(CC) $(CFLAGS) -o getifconf getifconf.c

lspromisc: lspromisc.c
           $(CC) $(CFLAGS) -o lspromisc lspromisc.c

clean:
          $(RM) pdump.o sockpacket.o sockpacket libpcap.o \
            libpcap getifconf lspromisc pfilter pfilter.o
-----</Makefile>-----

-----<lspromisc.c>-----
/* Copyright 1999 kay@phreedom.org. All rights Reserved */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if.h>

int main()
{
    struct ifconf ifc;
    struct ifreq ifr_x[50];
    int sockfd, err, i;

    sockfd = socket(PF_PACKET, SOCK_PACKET, 0);
    if (sockfd < 0) {
        perror("socket");
        exit(1);
    }

    ifc.ifc_len = 50 * sizeof(struct ifreq);
    ifc.ifc_req = ifr_x;
    err = ioctl(sockfd, SIOCGIFCONF, &ifc);
    if (err == -1) return EXIT_FAILURE;
    for (i = 0; i < ifc.ifc_len / sizeof(struct ifreq); i++) {
        err = ioctl(sockfd, SIOCGIFFLAGS, &ifr_x[i]);
        if (err == -1) perror("SIOCGIFFLAGS: ");
        else if (ifr_x[i].ifr_flags & IFF_PROMISC)
            printf("Interface %s is promiscuous\n",
                ifr_x[i].ifr_name);
    }
    return EXIT_SUCCESS;
}

/* eof */
-----<lspromisc.c>-----

[EOF]
```