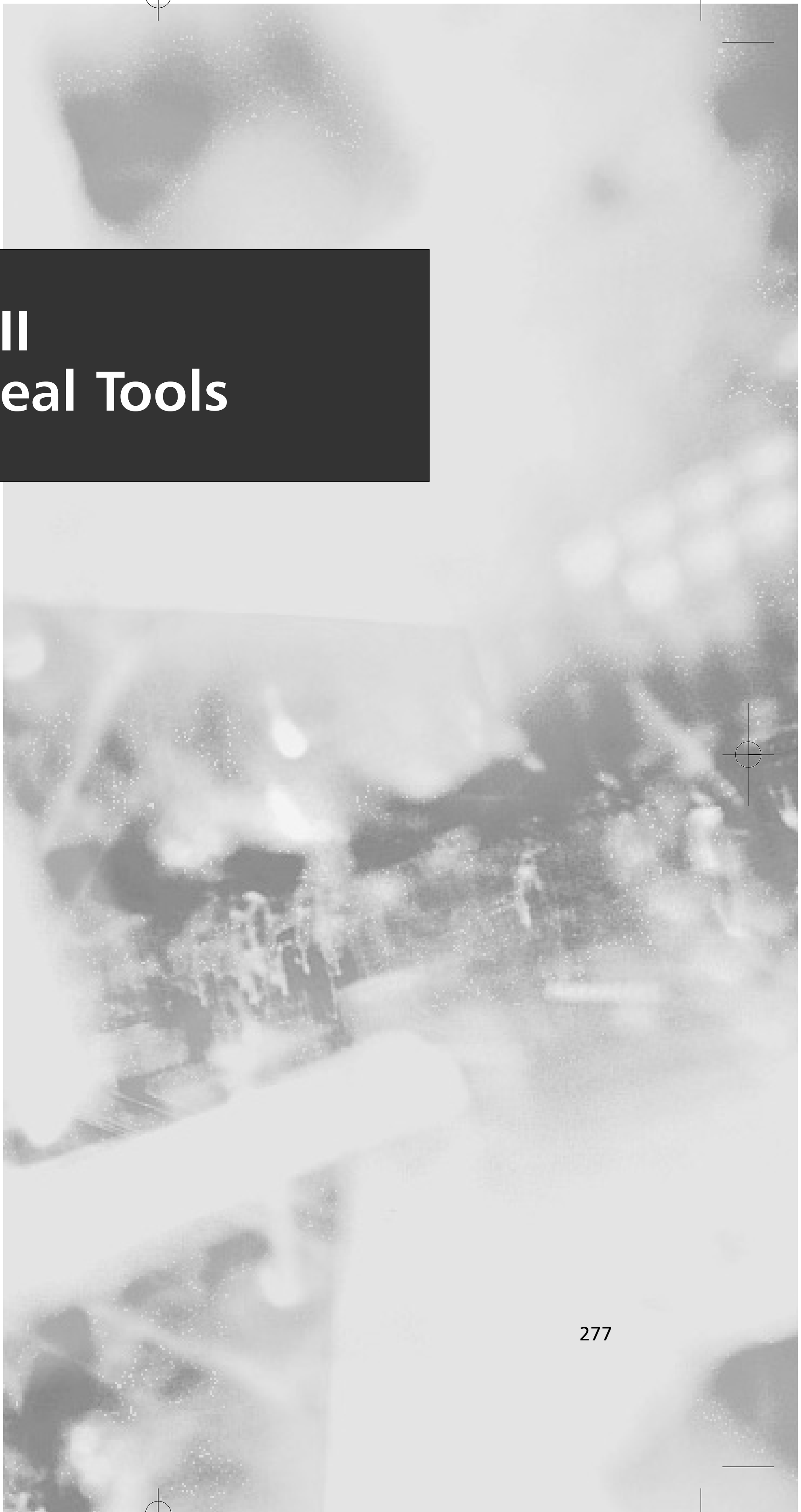
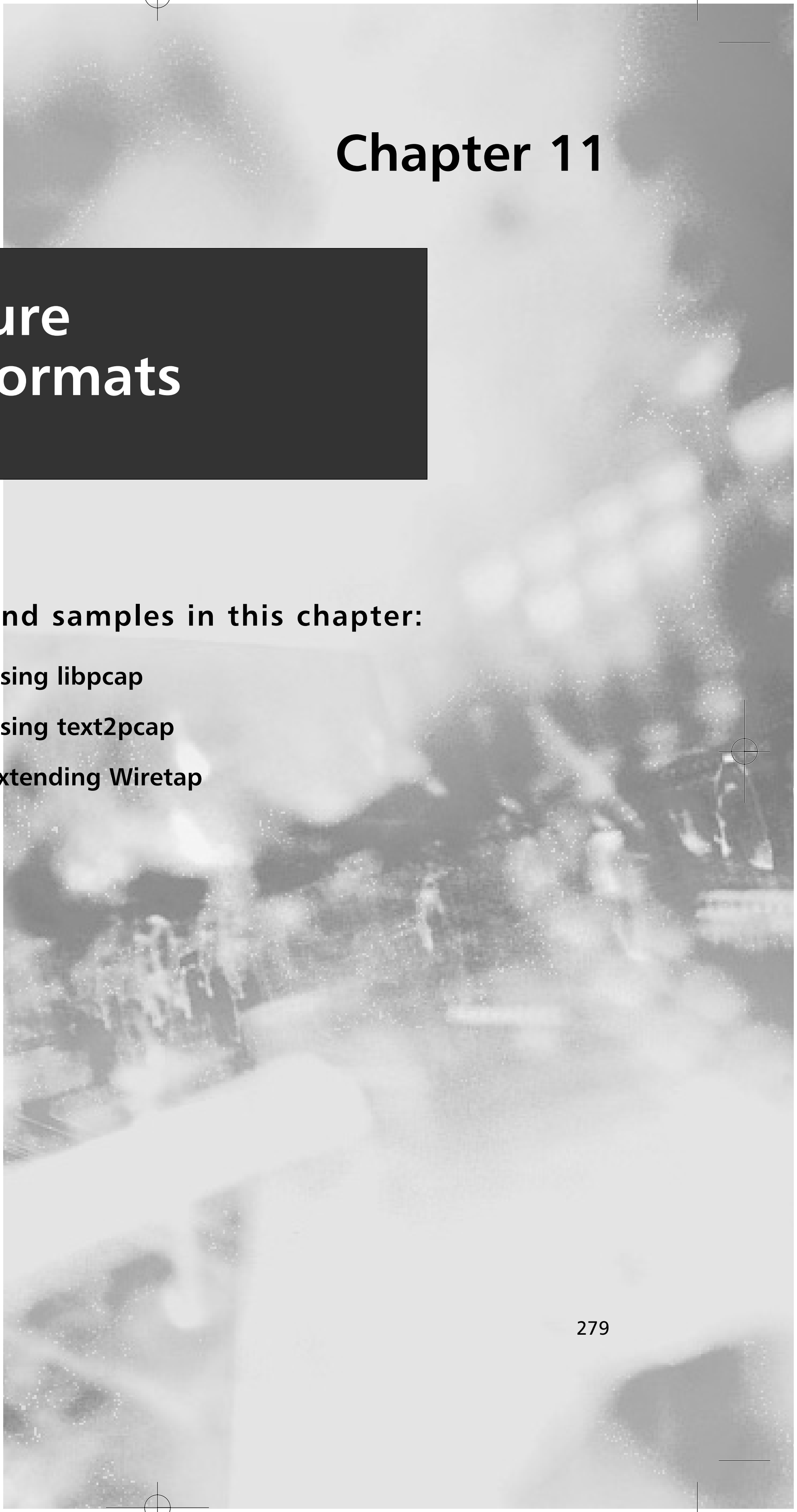


Part III Ethereal Tools



Chapter 11

Capture File Formats

Scripts and samples in this chapter:

- Using libpcap
- Using text2pcap
- Extending Wiretap

In This Toolbox

In this chapter you will learn how to enable Ethereal to read from new data sources. Programming with libpcap is introduced. You will be able to read ASCII hex dump files into Ethereal. For a more integrated solution, you will be able to teach Ethereal to read and possibly write a new file format natively.

Using libpcap

The most commonly used open source library for capturing packets from the network is libpcap, whose name is an abbreviation of *packet capture library*. Originally developed at the Lawrence Berkeley Laboratory, it is currently maintained by the same loosely knit group of people who maintain tcpdump, the venerable command-line packet capture utility. Both libpcap and tcp-dump are available online at www.tcpdump.org. A Windows version called WinPcap is available from http://winpcap.polito.it/.

Libpcap is able to save captured packets to a file. The pcap file format is unique to libpcap, but because so many open source applications use libpcap, a variety of applications can make use of these pcap files. The routines provided in libpcap allow you to save packets you have captured and to read pcap files from disk to analyze the stored data.

Selecting an Interface

The first issue to decide when capturing packets is which network interface to capture from. You can have libpcap pick a default interface for you. In that case, it picks the first active, non-loop-back interface. The *pcap_lookupdev* function picks the default interface.

```
char errbuf[PCAP_ERRBUF_SIZE];
char *default_device;
pcap_t *ph;

default_device = pcap_lookupdev(errbuf);

if (!default_device) {
    fprintf(stderr, "%s\n", errbuf);
    exit(1);
}
printf("Capturing on %s\n", default_device);
```

The *errbuf* parameter deserves a special mention. Many pcap functions make use of errbuf. It is a character array that you define in our program's address space, of at least PCAP_ERRBUF_SIZE length. The PCAP_ERRBUF_SIZE macro is defined in pcap.h, the file that provides the libpcap API (application program interface). If an error occurs in the pcap function, a description of the error is put into errbuf so that your program can present it to the user.

Alternatively, you can tell libpcap which interface to use. When starting a packet capture, the name of the interface is passed to libpcap. That is why pcap_lookupdev returns a *char*;* the function for opening an interface, *pcap_open_live*, expects the name of the interface as a string. The name of the interface differs according to the operating system. On Linux, the names of network interfaces are their simple, unadorned names, like eth0, and eth1. On BSD, the network interfaces are represented as device files, so the device filenames are given, like /dev/eth0. On Windows the names become more complicated; you shouldn't expect a user to be able to give the name of the network interface without aid.

To help users pick a network interface, libpcap provides a function that gives the calling program the list of available interfaces. It is the calling program's responsibility to present the list of interfaces to the user so that the user can choose the interface. The following example shows how to use this facility. If the program is run with no arguments, the list of interfaces is printed. If the program is run with a numeric argument, that number is used as an index to select the interface.

```
#include <stdio.h>
#include <pcap.h>

static char errbuf[PCAP_ERRBUF_SIZE];
void show_interfaces(void);
char* lookup_interface(int);

int
main(int argc, char **argv)
{
    pcap_t *ph;
    char *interface;
    int index;

    if (argc == 1) {
        /* No arguments; show the list of interfaces. */
        show_interfaces();
        exit(0);
    }
    else if (argc == 2) {
        /* Use the argument as an index */
        index = atoi(argv[1]);
        if (index < 0) {
            fprintf(stderr, "Number must be positive.\n");
            exit(1);
        }
        interface = lookup_interface(index);
        if (!interface) {
            fprintf(stderr, "No such interface: %d\n", index);
            exit(1);
        }
    }
    printf("Using %s\n", interface);
    exit(0);
}

/* Show the list of interfaces to the user. */
void
show_interfaces(void)
{
    pcap_if_t *interface_list;
    pcap_if_t *if_list_ptr;
    int result;
    int i;

    /* Ask libpcap for the list of interfaces */
    result = pcap_findalldevs(&interface_list, errbuf);
    if (result == -1) {
        fprintf(stderr, "%s\n", errbuf);
        exit(1);
    }

    /* Show them to the user */
    if_list_ptr = interface_list;
    i = 0;
    while (if_list_ptr) {
        if (if_list_ptr->description) {
            printf("%d. %s (%s)\n", i, if_list_ptr->name,
                    if_list_ptr->description);
        }
        else {
            printf("%d. %s\n", i, if_list_ptr->name);
        }
        if_list_ptr = if_list_ptr->next;
        i += 1;
    }
    pcap_freealldevs(interface_list);
}

/* Convert the user's argument to an interface name. */
char*
lookup_interface(int index)
{
    pcap_if_t *interface_list;
    pcap_if_t *if_list_ptr;
    int result;
    int i;

    /* Ask libpcap for the list of interfaces */
    result = pcap_findalldevs(&interface_list, errbuf);
    if (result == -1) {
        fprintf(stderr, "%s\n", errbuf);
        exit(1);
    }

    /* Find the right interface, according to the user's
    argument. */
    if_list_ptr = interface_list;
    i = 0;
    while (if_list_ptr) {
        if (i == index) {
            pcap_freealldevs(interface_list);
            return if_list_ptr->name;
        }
        if_list_ptr = if_list_ptr->next;
        i += 1;
    }
    /* If we reached here, then there's no such interface. */
    pcap_freealldevs(interface_list);
    return NULL;
}
```

As you can tell from the preceding example, the *pcap_findalldevs* function returns a linked list of pcap_if_t structures. The pcap_if_t structure definition can be found in pcap.h, the header file that defines the interface for libpcap.

```
/*
 * Item in a list of interfaces.
 */
typedef struct pcap_if pcap_if_t;
struct pcap_if {
    struct pcap_if *next;
    char *name;     /* name to hand to "pcap_open_live()" */
    char *description;  /* textual description of interface, or NULL */
    struct pcap_addr *addresses;
    bpf_u_int32 flags;  /* PCAP_IF_ interface flags */
};
```

Table 11.1 summarizes the functions provided by libpcap to help identify the interface where packets will be captured on. Of course, these functions don't have to be used if you wish to hard code the interface name into your program or have the user type the full name of the interface.

Table 11.1 Selecting Network Interfaces

Function	Use
pcap_lookup_dev	Return the name of the default network interface
pcap_find_alldevs	Return a list of available interfaces
pcap_free_alldevs	Free the memory allocated by pcap_findalldevs

Opening the Interface

Once your program has decided which interface to use, proceeding to capture packets is easy. The first step is to open the interface with pcap_open_live.

```
pcap_t  *pcap_open_live(const char *device, int snaplen,
                        int promisc, int to_ms, char *errbuf);
```

The *device* is the name of the network interface. The number of bytes you wish to capture from the packet is indicated by *snaplen*. If your intent is to look at all the data in a packet, as a general packet analyzer like Ethereal would do, you should specify the maximum value for snaplen, which is *65535*. The default behavior of other programs, like tcpdump, is to return only a small portion of the packet, or a *snapshot* (thus the term *snaplen*). Tcpdump's original focus was to analyze TCP (Transmission Control Protocol) headers, so capturing all the packet data was a waste of time.

The *promisc* flag should be *1* or *0*. It tells libpcap whether to put the interface into promiscuous mode. A zero value does not change the interface mode; if the interface is already in promiscuous mode because of another application, libpcap simply uses the interface as is. Capturing packets in promiscuous mode lets you see all the packets that the interface can see, even those destined for other machines. Nonpromiscuous mode captures only let you see packets destined for your machine, which includes broadcast packets and multicast packets if your machine is part of a multicast group.

A timeout value can be given in *to_ms*, which stands for *timeout, milliseconds*. The time-out mechanism tells libpcap how long to wait for the operating system kernel to queue received packets, even if a packet has been seen. Then libpcap can efficiently read a buffer full of packets from the kernel in one call. Not all operating systems support such a read time-out value. A zero value for to_ms tells the operating system to wait as long as necessary to read enough packets to fill the packet buffer, if it supports such a construct. For what it's worth, ethereal passes 1,000 as to_ms value.

Finally, *errbuf* is the same errbuf seen in other pcap functions. It points to space for libpcap to store an error or warning message.

Upon success, a pcap_t pointer is returned. On failure, a NULL value is returned.

Capturing Packets

There are two ways to capture packets from an interface in libpcap. The first method is to ask libpcap for a packet at a time, and the second is to start a loop in libpcap that calls your callback function when packets are ready.

There are two functions that deliver the packet-at-a-time approach:

```
const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h);
int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header,
        const u_char **pkt_data);
```

If you look closely at the two functions, you will notice that there are two types of information relevant for the captured packet. One is the *pcap_pkthdr*, or the packet header. The other is the *u_char* array of packet data. The u_char array is the actual data of the packet, whereas the packet header is the metadata about the packet. The definition of *pcap_pkthdr* is found in pcap.h.

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

The time stamp, *ts*, is the time at which that packet was captured. The *caplen* is the number of bytes captured from the packet. Remember that the *snaplen* parameter used when opening the interface may limit the portion of a packet that we capture. The number of bytes in the *u_char* array will be *caplen*. The last field in a pcap_pkthdr is *len*, which is the size of the packet on the wire. Thus, caplen will always be less than or equal to len, because we may always capture part or all of a packet, but never more than a packet.

The *pcap_next* function is very basic. If a problem occurs during the capture, a NULL pointer is returned; otherwise a pointer to the packet data is returned. Unfortunately, a problem may not always mean an error. A NULL can mean that no packets were read during a time-out period, if a time-out period is supported on that platform. To rectify this uncertain return code, *pcap_next_ex*, where *ex* is an abbreviation for *extended*, was added to the libpcap API. Its return value does specify exactly what happened during the capture, as shown in Table 11.2.

Table 11.2 pcap_next_ex Return Codes

Return Code	Meaning
1	Success
0	The timeout expired during a live capture.
-1	An error occurred while reading the packet.
-2	There are no more packets to read from a file.

The other way to capture packets with libpcap is to set up a callback function and have libpcap process packets in a loop. Your program can break the execution of that loop when a condition is met, like when the user presses a key or clicks a button. This callback method is the way most packet analyzers utilize libpcap. As before, there are two libpcap functions for capturing packets in this manner. They differ in how they handle their count (*cnt*) parameters.

```
int pcap_dispatch(pcap_t *p, int cnt,
        pcap_handler callback, u_char *user);
int pcap_loop(pcap_t *p, int cnt,
        pcap_handler callback, u_char *user);
```

In both cases, the callback function, which is defined in your program, has the same function signature, as both pcap functions expect a callback to be of the *pcap_handler* type.

```
typedef void (*pcap_handler)(u_char *user,
    const struct pcap_pkthdr *pkt_header,
    const u_char *pkt_data);
```

The *user* parameter is there for your program to pass arbitrary data to the callback function. Libpcap does not interpret this data or add to it in anyway. The same user value that was passed by your program to *pcap_dispatch* or *pcap_loop* is passed to your callback function. The *pkt_header* and *pkt_data* parameters are the same as we saw in the discussion of *pcap_next* and *pcap_next_ex*. These two fields point to the packet metadata and data, respectively.

The *cnt* parameter to *pcap_dispatch* specifies the maximum number of packets that libpcap will capture before stopping the execution of the loop and returning to your application, while honoring the time-out value you set for that interface. This is different from *pcap_loop*, which uses its *cnt* parameter to specify the number of packets to capture before returning.

In both cases, a *cnt* value of -1 has special meaning. For *pcap_dispatch*, a *cnt* of -1 tells libpcap to process all packets received in one buffer from the operating system. For *pcap_loop*, a *cnt* of -1 tells libpcap to continue capturing packets ad infinitum, until your program breaks the execution of the loop with *pcap_breakloop*, or until an error occurs. This is summarized in Table 11.3.

Table 11.3 cnt Parameter for pcap_dispatch and pcap_loop

Function	cnt parameter	Meaning
pcap_dispatch	> 0	Maximum number of packets to capture during time-out period
pcap_dispatch	-1	Process all packets received in one buffer from the operating system
pcap_loop	> 0	Capture this many packets
pcap_loop	-1	Capture until an error occurs, or until the program calls pcap_breakloop

The following example shows a simple example of using *pcap_loop* with a *pcap_handler* callback function to capture 10 packets. When you run this on UNIX, make sure you have the proper permissions to capture on the default interface. You can run the program as the *root* user to ensure this.

```
#include <stdio.h>
#include <pcap.h>

void
pcap_handler_cb(u_char *user, const struct pcap_pkthdr *pkt_header,
        const u_char *pkt_data)
{
    printf("Got packet: %d bytes captured:",
        pkt_header->caplen);

    if (pkt_header->caplen > 2) {
        printf("%02x %02x ... \n", pkt_data[0], pkt_data[1]);
    }
    else {
        printf("...\n");
    }
}

#define NUM_PACKETS 10

int
main(void)
{
    char errbuf[PCAP_ERRBUF_SIZE];
    char *default_device;
    pcap_t* ph;

    default_device = pcap_lookupdev(errbuf);

    if (!default_device) {
        fprintf(stderr, "%s\n", errbuf);
        exit(1);
    }

    printf("Opening  %s\n", default_device);
    ph = pcap_open_live(default_device, BUFSIZ, 1, 0, errbuf);

    printf("Capturing on %s\n", default_device);
    pcap_loop(ph, NUM_PACKETS, pcap_handler_cb, NULL);

    printf("Done.\n");
    exit(0);
}
```

Swiss Army Knife

Filtering Packets

The libpcap library is also famous for providing a packet filtering language that lets your application capture only the packets that the user is interested in. The syntax to the filter language is documented in the tcpdump man (manual) page. There are three functions you need to know to use filters.

To compile a filter string into bytecode, use **pcap_compile**. To attach the filter to your pcap_t object, use **pcap_setfilter**. Finally, to free the space used by the compiled bytecode, use **pcap_freecode**. This can be called immediately after a *pcap_setfilter* call.

Saving Packets to a File

To save packets to a file, libpcap provides a struct named *pcap_dumper_t* that acts as a file handle for your output file. There are five functions dealing with the *dump file*, or the *pcap_dumper_t* struct. They are listed in Table 11.4.

Table 11.4 pcap_dumper_t Functions

Function	Use
pcap_dump_open	Create an output file and pcap_dumper_t object
pcap_dump	Write a packet to the output file
pcap_dump_flush	Flush buffered packets immediately to output file
pcap_dump_file	Return the FILE member of the pcap_dumper_t struct
pcap_dump_close	Close the output file

Because of its function prototype, the *pcap_dump* function can be used directly as a callback to *pcap_dispatch* or *pcap_loop*. Although the first argument is *u_char**, in reality *pcap_dump* expects a *pcap_dumper_t** argument.

```
void    pcap_dump(u_char *, const struct pcap_pkthdr *, const u_char *);
```

In the following example, the *pcap_handler_cb* function is kept as the callback, and *pcap_dump* is called after printing information to *stdout*.

```
#include <stdio.h>
#include <pcap.h>

void
pcap_handler_cb(u_char *pcap_out, const struct pcap_pkthdr *pkt_header,
        const u_char *pkt_data)
{
    printf("Got packet: %d bytes captured:", pkt_header->caplen);

    if (pkt_header->caplen > 2) {
        printf("%02x %02x ... \n", pkt_data[0], pkt_data[1]);
    }
    else {
        printf("...\n");
    }
    pcap_dump(pcap_out, pkt_header, pkt_data);
}

#define NUM_PACKETS 10

int
main(void)
{
    char errbuf[PCAP_ERRBUF_SIZE];
    char *default_device;
    pcap_t* ph;
    pcap_dumper_t *pcap_out;

    default_device = pcap_lookupdev(errbuf);

    if (!default_device) {
        fprintf(stderr, "%s\n", errbuf);
        exit(1);
    }

    printf("Opening  %s\n", default_device);
    ph = pcap_open_live(default_device, BUFSIZ, 1, 0, errbuf);

    printf("Writing to out.cap\n");
    pcap_out = pcap_dump_open(ph, "out.cap");
    if (!pcap_out) {
        printf("Could not open out.cap for writing.\n");
        exit(1);
    }
    printf("Capturing on %s\n", default_device);
    pcap_loop(ph, NUM_PACKETS, pcap_handler_cb, (u_char*) pcap_out);

    printf("Done.\n");
    pcap_dump_close(pcap_out);
    exit(0);
}
```

Master Craftsman

Writing pcap Files without Capturing Packets

The *pcap_dump_open* function requires a *pcap_t* object. What if you want to write pcap files using libpcap, but the source of your packets is not the libpcap capture mechanism? Libpcap provides the *pcap_open_dead,* which will return a *pcap_t* object as if you had opened an interface, but does not open any network interface. The *pcap_open_dead* function requires two parameters: the link layer type (a *DLT* value defined in pcap-bpf.h) and the *snaplen*, or how many bytes of each packet you intended to capture. It's safe to set *snaplen* to its maximum value, 65535. That maximum value comes from the filter bytecode compiler, which uses a two-byte integer to report packet lengths. With those two values, link layer type and *snaplen*, libpcap can write the file header for the generated pcap file.

Using text2pcap

text2pcap is a command-line tool that comes with Ethereal that helps you convert ASCII hex dump files to pcap files that can be loaded into Ethereal. While flexible about the style of hex dump it reads, it does expect a certain format of hex dump file.

text2pcap Hex Dumps

Another way to analyze a file that Ethereal cannot read is to convert it to a file format that Ethereal does know how to read. This can be done by using an ASCII hex dump file as an intermediate representation, and using text2pcap, supplied in the Ethereal distribution, to convert the hex dump file to a pcap file. The hex dump format is a useful intermediate representation because many packet analyzers can produce a hex dump in addition to saving their data in their proprietary file format. The hex dump format is also easy to produce with other tools.

The hex dump format that text2pcap expects is a hexadecimal offset, starting at 0, followed by hexadecimal bytes. Of course, a single packet can consist of multiple lines of hex dump, but the offset must increase correctly. The following is a valid hex dump for text2pcap.

```
000000 ff ff ff ff ff ff 00 09
000008 6b 50 f9 ed 08 06 00 01
000010 08 00 06 04 00 01 00 09
000018 6b 50 f9 ed 0a 0a 0a 39
000020 00 00 00 00 00 00 0a 0a
000028 0a 04
```

The offsets are the first field in the line. They are more than two hexadecimal digits wide, to distinguish them from the data bytes. The data bytes are fields of two hexadecimal digits. text2pcap is flexible in the format of the hex dump that it accepts. The offsets do not have to increase by 0x08, but by any value that you wish. For example, each line of hexadecimal digits can have 16 data bytes.

```
0000  ff ff ff ff ff ff 00 09 6b 50 f9 ed 08 06 00 01
0010  08 00 06 04 00 01 00 09 6b 50 f9 ed 0a 0a 0a 39
0020  00 00 00 00 00 00 0a 0a 0a 04
```

You can have more than one packet in a file by separating their data by a blank line.

```
0000  ff ff ff ff ff ff 00 09 6b 50 f9 ed 08 06 00 01
0010  08 00 06 04 00 01 00 09 6b 50 f9 ed 0a 0a 0a 39
0020  00 00 00 00 00 00 0a 0a 0a 04

0000  00 09 6b 50 f9 ed 00 50 e8 01 42 ec 08 06 00 01
0010  08 00 06 04 00 02 00 50 e8 01 42 ec 0a 0a 0a 04
0020  00 09 6b 50 f9 ed 0a 0a 0a 39 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00 00 00 00 00
```

Many packet analyzers print extra characters to the right of the hex dump to show the ASCII equivalent of the hexadecimal bytes. For hexadecimal values that don't have a printable ASCII character, many packet analyzers print a period. text2pcap ignores these extra characters automatically.

```
0000  ff ff ff ff ff ff 00 09 6b 50 f9 ed 08 06 00 01   ........kP......
0010  08 00 06 04 00 01 00 09 6b 50 f9 ed 0a 0a 0a 39   ........kP.....9
0020  00 00 00 00 00 00 0a 0a 0a 04                     ..........

0000  00 09 6b 50 f9 ed 00 50 e8 01 42 ec 08 06 00 01   ..kP...P..B.....
0010  08 00 06 04 00 02 00 50 e8 01 42 ec 0a 0a 0a 04   .......P..B.....
0020  00 09 6b 50 f9 ed 0a 0a 0a 39 00 00 00 00 00 00   ..kP.....9......
0030  00 00 00 00 00 00 00 00 00 00 00 00               ............
```

Packet Metadata

The hex dump file format is a simple way to feed packet data into text2pcap, but it has no means of providing packet metadata to text2pcap. The minimal metadata that is interesting includes the time stamp of the arrival of the packet and the data link type, which indicates the first protocol in the packet. By default, text2pcap will invent a time stamp for the packets, incrementing the time stamp by one second. This is a workable, if not elegant solution. text2pcap's **-l** option lets you specify the data link type for all the packets in the hex dump. The libpcap file format allows one data link type for the entire file; you cannot have packets with different data link types in the same file.

Table 11.5 shows some of the more useful data link types from libpcap, defined in the pcap-bpf.h file in the libpcap distribution. These are the values that are passed into text2pcaps with the **-l** option. The pcap-bpf.h has many more values defined; read the file to find more if you need to.

Table 11.5 Some libpcap Data Link Type Values

Value	Meaning
1	Ethernet
6	IEEE 802 Networks (Token-Ring)
8	SLIP
9	PPP
10	FDDI
12	Raw IP
14	Raw IP (on OpenBSD)
19	Classical IP over ATM, on Linux
104	Cisco HDLC
107	Frame Relay
120	Aironet link-layer

Suppose your hex dump file is named hex.txt, and the packets have an Ethernet data link type. The text2pcap command line to convert the hex dump file to a pcap file would be written as follows:

```
$ text2pcap -l 1 hex.txt newfile.cap
```

Or, since Ethernet is the default data link type, the **-l 1** can be removed.

```
$ text2pcap hex.txt newfile.cap
```

A new file, newfile.cap, is produced and can be loaded into Ethereal as you would any other capture file.

Swiss Army Knife

Using text2cap for Higher Protocol Layers

The text2pcap tool has options to prepend fake data to each packet. This is useful if your hex dump shows data at a higher layer than the link layer. The **–e**, **-i**, **-T**, **-u**, **-s**, and **–S** options prepend dummy Ethernet, IP (Internet Protocol), TCP, UDP (User Datagram Protocol), and SCTP (Stream Control Transmission Protocol) headers. Both the **–s** and **–S** options prepend SCTP headers in different ways.

Continued

This functionality is useful for application developers whose programs can save their socket data as a hex dump to a file. A program that acts as an HTTP (Hypertext Transfer Protocol) proxy, for example, could save the HTTP data to a hex dump file. Then text2pcap could prepend a TCP header to each HTTP packet. The packets would have to be in one direction, going to the server or going to the client, since the text2pcap command-line indicates source and destination TCP ports. The text2pcap command line includes the following code:

```
$ text2pcap –T 2000,80 input.hex output.cap
```

Converting Other Hex Dump Formats

text2pcap is flexible in its ability to read hex dump files. It skips extra white space and ignores extra characters. But it does expect the offsets to exist before the data and to be more than two hex digits long. Sometimes you are presented with a hex dump format that does not meet the minimum requirements of text2pcap. However, it is simple to write a script to convert your current hex dump format to a hex dump format that text2pcap will read. For example, the hex dump produced by Juniper Network's NetScreen firewall product is not compatible with text2pcap because it does not have offsets. The following example was posted to the ethereal-dev mailing list on October 19, 2004. You can get it from www.ethereal.com/lists/ethereal-dev/200410/msg00295.html.

```
13301.0: 0(i):0003ba0f9adf->0010db621640/0800
              10.0.33.254->10.0.33.35/1, tlen=84
              vhl=45, tos=00, id=59119, frag=4000, ttl=255
              icmp:type=8, code=0
              00 10 db 62 16 40 00 03 ba 0f 9a df 08 00 45 00
              00 54 e6 ef 40 00 ff 01 3d 98 0a 00 21 fe 0a 00
              21 23 08 00 30 d1 14 f2 00 00 41 75 40 c1 00 07
              44 fc 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15
              16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
              26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
              36 37

13301.0: 0(o):0010db621640->0003ba0f9adf/0800
              10.0.33.35->10.0.33.254/1, tlen=84
              vhl=45, tos=00, id=3662, frag=0000, ttl=64
              icmp:type=0, code=0
              00 03 ba 0f 9a df 00 10 db 62 16 40 08 00 45 00
              00 54 0e 4e 00 00 40 01 15 3b 0a 00 21 23 0a 00
              21 fe 00 00 38 d1 14 f2 00 00 41 75 40 c1 00 07
              44 fc 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15
              16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
              26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
              36 37
```

The format is almost exactly what text2pcap needs; individual packets are separated by a blank line and the data bytes are represented by two hex digits. A script can convert this format to a text2pcap-compatible format if it can provide the offsets to the data lines. We might also want to throw away the packet information that precedes the actual packet data. Even though text2pcap will ignore them, there's a small chance that some packet could have extra information that looks like packet data, and that would confuse text2pcap.

The program in the following example is a small Python script that will convert the NetScreen hex dump format to a format that is readable by text2pcap. It uses regular expressions to find the data lines in the hex dump, and then it counts the hexadecimal pairs in those lines to produce correct offsets. It prints just the offsets and the data, ignoring the additional packet information and the ASCII characters to the right of the hex dump.

```
#!/usr/bin/env python
import sys
import re

# This regular expression pattern finds a sequence of
# 1 - 16 pairs of hexadecimal digits, separated by spaces.
re_hex_line = re.compile(r"(?P<hex>([0-9a-f]{2} ){1,16})")

def main():
    datalines = []

    for line in sys.stdin.xreadlines():
        m = re_hex_line.search(line)
        if m:
            # If we see a valid data line, append
            # it to our list of data lines.
            datalines.append(m)
        else:
            # If we don't see a data line, then
            # check to see if we have already found some
            # data lines. If so, then we reached the end of
            # the packet, so print the packet, and reset
            # the list of datalines.
            if datalines:
                print_datalines(datalines)
                datalines = []

    # We've reached the end of the file. See if we have
    # any data lines to print (in the case of an EOF being
    # reached directly after a packet, instead of finding blank
    # lines after a packet) and print the packet.
    if datalines:
        print_datalines(datalines)

def print_datalines(datalines):
    offset = 0
    for hexgroup in datalines:
        # Retrieve the substring that has the hex digits.
        hexline = hexgroup.group("hex")

        # Create an array by splitting the substring on
        # whitspace.
        hexpairs = hexline.split()
        # Print the data
        print "%08x   %s" % (offset, hexline)

        # Increae the offset by the number of bytes
        # that were represented in the data line.
        offset += len(hexpairs)

    # Print a blank line
    print

if __name__ == "__main__":
    main()
```

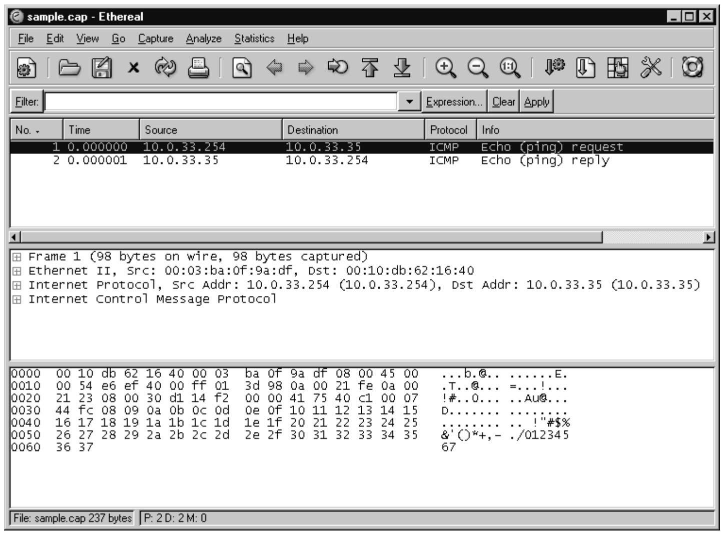
This is what is produced when the Python script is run.

```
00000000   00 10 db 62 16 40 00 03 ba 0f 9a df 08 00 45 00
00000010   00 54 e6 ef 40 00 ff 01 3d 98 0a 00 21 fe 0a 00
00000020   21 23 08 00 30 d1 14 f2 00 00 41 75 40 c1 00 07
00000030   44 fc 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15
00000040   16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
00000050   26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
00000060   36 37

00000000   00 03 ba 0f 9a df 00 10 db 62 16 40 08 00 45 00
00000010   00 54 0e 4e 00 00 40 01 15 3b 0a 00 21 23 0a 00
00000020   21 fe 00 00 38 d1 14 f2 00 00 41 75 40 c1 00 07
00000030   44 fc 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15
00000040   16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
00000050   26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
00000060   36 37
```

You can then run textpcap on this output and produce a pcap file. Figure 11.1 shows what that pcap file looks like when loaded in Ethereal.

Figure 11.1 Ethereal Reading the Generated pcap File



No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.0.33.254	10.0.33.35	ICMP	Echo (ping) request
2	0.000001	10.0.33.35	10.0.33.254	ICMP	Echo (ping) reply

```
⊞ Frame 1 (98 bytes on wire, 98 bytes captured)
⊞ Ethernet II, Src: 00:03:ba:0f:9a:df, Dst: 00:10:db:62:16:40
⊞ Internet Protocol, Src Addr: 10.0.33.254 (10.0.33.254), Dst Addr: 10.0.33.35 (10.0.33.35)
⊞ Internet Control Message Protocol
```

```
0000  00 10 db 62 16 40 00 03  ba 0f 9a df 08 00 45 00   ...b.@.. ......E.
0010  00 54 e6 ef 40 00 ff 01  3d 98 0a 00 21 fe 0a 00   .T..@... =...!...
0020  21 23 08 00 30 d1 14 f2  00 00 41 75 40 c1 00 07   !#..0... ..Au@...
0030  44 fc 08 09 0a 0b 0c 0d  0e 0f 10 11 12 13 14 15   D....... ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                              67
```

File: sample.cap 237 bytes | P: 2 D: 2 M: 0

Extending Wiretap

A more powerful way to have Ethereal read a new file format is to teach Ethereal how to read it natively. By integrating this code with Ethereal, you will no longer have to go through the transformation step of running textp2cap before you can read your file. This approach is most useful if you intend to use Ethereal on your new file format very often.

The Wiretap Library

Ethereal uses a library called *wiretap*, which comes with the Ethereal source code, to read and write many packet analyzer file formats. Most people don't realize that Ethereal uses libpcap only for capturing packets. It does not use libpcap for reading pcap files. Ethereal's wiretap library reads pcap files. The reason wiretap reimplemented the pcap-reading code is that wiretap has to read many variations of the pcap file format. There are various vendors that have modified the pcap format, sometimes without explicitly changing the version number inside the file. Wiretap uses heuristics to determine which format the pcap file is, and it is generally successful.

Wiretap currently reads these file formats (this list is from the Ethereal Web site at www.ethereal.com/introduction.html):

- libpcap
- NAI's Sniffer (compressed and uncompressed) and Sniffer Pro
- NetXray
- Sun snoop and atmsnoop
- Shomiti/Finisar Surveyor
- AIX's iptrace
- Microsoft's Network Monitor
- Novell's LANalyzer
- RADCOM's WAN/LAN Analyzer
- HP-UX nettl
- i4btrace from the ISDN4BSD project
- Cisco Secure IDS iplog
- pppd log (pppdump-format)
- The AG Group's/WildPacket's EtherPeek/TokenPeek/AiroPeek
- Visual Networks' Visual UpTime
- Lucent/Ascend WAN router traces
- Toshiba ISDN routers traces
- VMS's TCPIPtrace utility's text output
- DBS Etherwatch utility for VMS

Because Wiretap makes use of *zlib*, a compression library, any of these files can be compressed with gzip, and wiretap will automatically decompress them while reading them. It doesn't save the uncompressed version of the file; it decompresses the portion of the file, in memory, that it is currently reading.

Reverse Engineering a Capture File Format

To teach Ethereal how to read a new file format, you will add a module to the wiretap library. To do this, you must understand enough of your file format to be able to find the packet data. It's easiest, of course, if you have documentation for the file format, or if you designed the file format yourself. But in the case of a lack of documentation, it is usually relatively easy to reverse engineer a packet file format, simply because you can examine the packets in the tool that created that file. By using the original tool, you know the data in each packet. By creating a hex dump of the file, you can look for the same packet data. The non-data portion of the packet is the metadata, part of which you may be able to decode. Not all packet file formats save the packet data unadulterated. For example, the Sniffer tool can save packets with its own compression algorithm, which makes reverse engineering a more difficult task. But the great majority of tools simply save packet data as is.

Understanding Capture File Formats

Commonly, packet trace files have simple formats. First there is a file header that indicates the type and perhaps version of the file format. Then the packets themselves will follow, each with a header giving metadata, followed by the packet data shown in the following example:

```
File Header
Packet #1 Header
Packet #1 Data
Packet #2 Header
Packet #2 Data
Packet #3 Header
Packet #3 Data
etc.
```

There are sometimes variations that allow different record types to be stored in the file so that not each record is a packet. These are commonly called TLV format, for type, length, value, as those are the three fields that are necessary in order to have variable record type and sizes.

The next example shows a TLV capture file format. Usually, by correlating your packet analyzer's analysis with the contents of the trace file, you can determine enough of the file format so that the wiretap library can read the file.

```
File Header
Record #1 Type
Record #1 Length
Record #1 Value         Packet Header and Data
Record #2 Type
Record #2 Length
Record #2 Value         Other Data
etc.
```

As an example of reverse engineering, we take a look at an iptrace file produced on an old AIX 3 machine. On this operating system there were two programs related to packet capturing. The iptrace program captured packets into a file. The ipreport program read one of these trace files and produced a protocol dissection in text format. The first step in reverse engineering the file format is to produce the protocol dissection so that you know which bytes belong to which packet. The next example shows the protocol dissection of the first three packets in a trace file.

```
ETHERNET packet : [ 08:00:5a:cd:ba:52 -> 00:e0:1e:a6:dc:e8 ]  type 800  (IP)
IP header breakdown:
        < SRC =  192.168.225.132 >
        < DST =  192.168.129.160 >
        ip_v=4, ip_hl=20, ip_tos=0, ip_len=84, ip_id=20884, ip_off=0
        ip_ttl=255, ip_sum=859e, ip_p = 1 (ICMP)
ICMP header breakdown:
        icmp_type=8 (ECHO_REQUEST)  icmp_id=9646  icmp_seq=0
00000000     383e3911 00074958 08090a0b 0c0d0e0f     |8>9...IX........|
00000010     10111213 14151617 18191a1b 1c1d1e1f     |................|
00000020     20212223 24252627 28292a2b 2c2d2e2f     | !"#$%&'()*+,-./|
00000030     30313233 34353637                       |01234567        |

=====( packet received on interface en0 )=====Fri Nov 26 07:38:57 1999
ETHERNET packet : [ 00:e0:1e:a6:dc:e8 -> 08:00:5a:cd:ba:52 ]  type 800  (IP)
IP header breakdown:
        < SRC =  192.168.129.160 >
        < DST =  192.168.225.132 >
        ip_v=4, ip_hl=20, ip_tos=0, ip_len=84, ip_id=47965, ip_off=0
        ip_ttl=251, ip_sum=1fd5, ip_p = 1 (ICMP)
ICMP header breakdown:
        icmp_type=0 (ECHO_REPLY)  icmp_id=9646  icmp_seq=0
00000000     383e3911 00074958 08090a0b 0c0d0e0f     |8>9...IX........|
00000010     10111213 14151617 18191a1b 1c1d1e1f     |................|
00000020     20212223 24252627 28292a2b 2c2d2e2f     | !"#$%&'()*+,-./|
00000030     30313233 34353637                       |01234567        |

=====( packet transmitted on interface en0 )=====Fri Nov 26 07:38:58 1999
ETHERNET packet : [ 08:00:5a:cd:ba:52 -> 00:e0:1e:a6:dc:e8 ]  type 800  (IP)
IP header breakdown:
        < SRC =  192.168.225.132 >
        < DST =  192.168.129.160 >
        ip_v=4, ip_hl=20, ip_tos=0, ip_len=84, ip_id=20890, ip_off=0
        ip_ttl=255, ip_sum=8598, ip_p = 1 (ICMP)
ICMP header breakdown:
        icmp_type=8 (ECHO_REQUEST)  icmp_id=9646  icmp_seq=1
00000000     383e3912 00074d6c 08090a0b 0c0d0e0f     |8>9...Ml........|
00000010     10111213 14151617 18191a1b 1c1d1e1f     |................|
00000020     20212223 24252627 28292a2b 2c2d2e2f     | !"#$%&'()*+,-./|
00000030     30313233 34353637                       |01234567        |
```

The next step is to produce a hex dump of the packet trace file. It is useful to print this hex dump to paper so that you can make scribbles on it as you analyze the format. A good tool for producing hex dumps from a file is xxd, a command-line program that comes with the vim editor package available from www.vim.org. As you see in the following line of code, using xxd is simple:

```
$ xxd input-file output-file
```

By default, xxd prints bytes in groups of two. The following code shows these two groups:

```
0000000: 6970 7472 6163 6520 312e 3000 0000 7838  iptrace 1.0...x8
0000010: 3e39 1100 0000 0065 6e00 0001 4575 1001  >9.....en...Eu..
```

The following example shows the first 25 lines of the hex dump for the trace file that corresponds to the protocol analysis in the preceding example. The offset values were added to the top of the hex dump after the fact to aid you in reading the data.

```
 offset  00   02   04   06   08   0a   0c   0e
 offset  01   03   05   07   09   0b   0d   0f

0000000: 6970 7472 6163 6520 312e 3000 0000 7838  iptrace 1.0...x8
0000010: 3e39 1100 0000 0065 6e00 0001 4575 1001  >9.....en...Eu..
0000020: 4594 5000 0000 0006 0100 e01e a6dc e808  E.P.............
0000030: 005a cdba 5208 0045 0000 5451 9400 00ff  .Z..R..E..TQ....
0000040: 0185 9ec0 a8e1 84c0 a881 a008 002c a025  .............,.%
0000050: ae00 0038 3e39 1100 0749 5808 090a 0b0c  ...8>9...IX.....
0000060: 0d0e 0f10 1112 1314 1516 1718 191a 1b1c  ................
0000070: 1d1e 1f20 2122 2324 2526 2728 292a 2b2c  ... !"#$%&'()*+,
0000080: 2d2e 2f30 3132 3334 3536 3700 0000 7838  -./01234567...x8
0000090: 3e39 1108 000e 0065 6e00 0001 4575 1001  >9.....en...Eu..
00000a0: 4594 5000 0000 0006 0008 005a cdba 5200  E.P........Z..R.
00000b0: e01e a6dc e808 0045 0000 54bb 5d00 00fb  .......E..T.]...
00000c0: 011f d5c0 a881 a0c0 a8e1 8400 0034 a025  .............4.%
00000d0: ae00 0038 3e39 1100 0749 5808 090a 0b0c  ...8>9...IX.....
00000e0: 0d0e 0f10 1112 1314 1516 1718 191a 1b1c  ................
00000f0: 1d1e 1f20 2122 2324 2526 2728 292a 2b2c  ... !"#$%&'()*+,
0000100: 2d2e 2f30 3132 3334 3536 3700 0000 7838  -./01234567...x8
0000110: 3e39 1200 0000 0065 6e00 0001 4575 1001  >9.....en...Eu..
0000120: 4594 5000 0000 0006 0100 e01e a6dc e808  E.P.............
0000130: 005a cdba 5208 0045 0000 5451 9a00 00ff  .Z..R..E..TQ....
0000140: 0185 98c0 a8e1 84c0 a881 a008 0028 8a25  .............(.%
0000150: ae00 0138 3e39 1200 074d 6c08 090a 0b0c  ...8>9...Ml.....
0000160: 0d0e 0f10 1112 1314 1516 1718 191a 1b1c  ................
0000170: 1d1e 1f20 2122 2324 2526 2728 292a 2b2c  ... !"#$%&'()*+,
0000180: 2d2e 2f30 3132 3334 3536 3700 0000 7838  -./01234567...x8
```

Finding Packets in the File

The first step is to find the locations of the packet data. The locations are easy to find because the protocol dissection shows the packet data as hex bytes, and the hex dump shows the same thing. However, the ipreport protocol dissection is tricky. The hex data shown is not the entire packet data; it is only the packet payload. The protocol information that the report shows as *header breakdown* is not shown in the hex dump in the report. At this point it is important to realize that these packets are Ethernet packets, and that Ethernet headers, like many link layers, begin by listing the source and destination Ethernet addresses (also known as hardware addresses, or MAC addresses). In the case of Ethernet, the destination Ethernet address is listed first, followed by the source destination address. Luckily for us, the Ethernet hardware addresses in the report are represented by sequences of six hex digits. To find the beginning of the packet in our hex dump, we have to find the sequences of hex digits in Table 11.6.

Table 11.6 Bytes to Look For

Packet Number	Starts with (Destination)	Followed by (Source)	Soon Followed by (Payload)	Ends with (Payload)
1	00:e0:1e:a6:dc:e8	08:00:5a:cd:ba:52	383e3911 00074958	30313233 34353637
2	08:00:5a:cd:ba:52	00:e0:1e:a6:dc:e8	383e3911 00074958	30313233 34353637
3	00:e0:1e:a6:dc:e8	08:00:5a:cd:ba:52	383e3912 00074d6c	30313233 34353637

Searching for these sequences of bytes in the hex dump, we find the offsets listed in Table 11.7.

Table 11.7 Packet Data Start and End Offsets

Packet Number	Data Start Offset	Data End Offset
1	0x29	0x8a
2	0xa9	0x10a
3	0x129	0x18a

To determine the size of the packet metadata, we look at the number of bytes preceding each packet. At first we don't consider the space before the first packet because we are guessing that it contains both a file header and a packet header. To calculate the size of the packet header, we find the difference between the two offsets and subtract 1 because we want the number of bytes between the offsets, not including either of the offsets.

```
(Beginning of Packet) - (End of Previous Packet) - 1
```

From this formula, we see in Table 11.8 that the packet headers for packets 2 and 3 are the same length.

Table 11.8 Computed Packet Lengths

Between Packet Numbers	Equation (hex)	Equation (decimal)	Result (decimal)
1 and 2	0xa9 - 0x8a - 1	169 - 138 – 1	30
2 and 3	0x129 - 0x10a - 1	297 - 266 – 1	30

There are 30 bytes between the packets, so the packet header is probably 30 bytes long. The initial packet starts at offset 0x29, or 41 decimal. If we guess that the initial packet also has a packet header of 30 bytes, then the remaining space must be the file header, which will be 11 bytes long (41 - 30 == 11). Our proposed file format is beginning to take shape in Table 11.9.

Table 11.9 File Format Proposal

Item	Length
File header	11 bytes
Packet #1 header	30 bytes
Packet #1 data	n bytes
Packet #2 header	30 bytes
Packet #2 data	n bytes
Packet #3 header	30 bytes
Packet #3 data	n bytes

Master Craftsman

Reverse Engineering for text2pcap

Once enough of the file format is reverse engineered so that you know where packets begin and end, you could write a script that would read the file, pass over all packet headers, and simply write the packet data to a file in hex dump format. Then text2pcap could be run on that file, making sure to set the right encapsulation type via the **–l** option. The resulting pcap file could then be loaded into Ethereal. This is a viable option for people who don't have a development environment set up on their computer to enable them to re-build Ethereal from sources.

First we attack the file header. What data does our first 11 bytes contain? Look at bytes 0x00 through 0x0a in the hex dump.

```
 offset  00   02   04   06   08   0a   0c   0e
 offset  01   03   05   07   09   0b   0d   0f

0000000: 6970 7472 6163 6520 312e 3000 0000 7838  iptrace 1.0...x8
```

The first 11 bytes of the file are a string giving the tool name and version that created this file: iptrace 1.0. This type of identifying information is exactly what we would expect to find in a file header. It lets a tool, like the wiretap library, uniquely identify the format of this file.

Now we must understand the contents of the packet header. We know that four types of information must be in the packet header. The length of the packet data must exist so that the ipreport tool can know how much data to read for each packet. In addition, the following data are in the dissection produced by ipreport, so they must exist in the packet data:

- Time stamp
- Interface name
- Direction (transmit/receive)

We might also expect to find a field that identifies the link layer of the capture, although it is possible that the ipreport tool could infer this from the name of the interface. The only way we will determine that, however, is to have an iptrace file for two different link layers. This trace was made on and Ethernet interface. We would need to have an iptrace file for something else, like Token-Ring or FDDI (Fiber Distributed Data Interface), to see which field varied along with the link layer type.

We can organize what we know. Table 11.10 calculates the packet data length by using the data offsets. This time the equation is as follows:

```
(End Offset) - (Start Offset) + 1
```

We add 1 to the difference because we want the number of bytes between the offsets, but this time we include the offsets in the count. Doing this calculation in Table 11.10, we see that each byte is 98, or 0x62, bytes long.

Table 11.10 Computed Packet Data Lengths

Packet Number	Data Start Offset	Data End Offset	Equation	Answer (Hexa-decimal)	Answer (Decimal)
1	0x29	0x8a	0x8a - 0x29 + 1	0x62	98
2	0xa9	0x10a	0x10a - 0xa9 + 1	0x62	98
3	0x129	0x18a	0x18a - 0x129 + 1	0x62	98

Table 11.11 shows the packet length and time stamp of each packet. Table 11.12 shows the header data.

Table 11.11 All Metadata Summarized

Packet Number	Data Length	Time Stamp	Interface	Direction
1	0x62	Fri Nov 26 07:38:57 1999	en0	Transmit
2	0x62	Fri Nov 26 07:38:57 1999	en0	Receive
3	0x62	Fri Nov 26 07:38:58 1999	en0	Transmit

Table 11.12 All Packet Header Data Bytes

Packet Number	Header Data
1	00 00 00 78 38 3e 39 11 00 00 00 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 01

Continued

Table 11.12 continued All Packet Header Data Bytes

Packet Number	Header Data
2	00 00 00 78 38 3e 39 11 08 00 0e 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 00
3	00 00 00 78 38 3e 39 12 00 00 00 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 01

We can see right away that the packet data length is not represented verbatim in the packet header. Each packet is 0x62 bytes long, but there is not a 0x62 value in any of the headers. Unfortunately, these first three packets do not have enough variation in them to make analysis easy. We must pick some data from another packet that has a different length. We use the same analysis technique we have used so far to find another packet. An interesting packet later in the trace file, packet number 7, is shown in the following example:

```
=====( packet transmitted on interface en0 )=====Fri Nov 26 07:39:05 1999
ETHERNET packet : [ 08:00:5a:cd:ba:52 -> 00:e0:1e:a6:dc:e8 ]  type 800  (IP)
IP header breakdown:
        < SRC =  192.168.225.132 >
        < DST =  192.168.129.160 >
        ip_v=4, ip_hl=20, ip_tos=16, ip_len=44, ip_id=20991, ip_off=0
        ip_ttl=60, ip_sum=4847, ip_p = 6 (TCP)
TCP header breakdown:
        <source port=4257, destination port=25(smtp) >
        th_seq=b6bfbc01, th_ack=0
        th_off=6, flags<SYN |>
        th_win=16384, th_sum=f034, th_urp=0
00000000     020405b4                                |...´            |

 offset  00   02   04   06   08   0a   0c   0e
 offset  01   03   05   07   09   0b   0d   0f

0000300: 2d2e 2f30 3132 3334 3536 3700 0000 5238  -./01234567...R8
0000310: 3e39 1900 0000 0065 6e00 0001 4575 1001  >9.....en...Eu..
0000320: 4594 5000 0000 0006 0100 e01e a6dc e808  E.P............
0000330: 005a cdba 5208 0045 1000 2c51 ff00 003c  .Z..R..E..,Q...<
0000340: 0648 47c0 a8e1 84c0 a881 a010 a100 19b6  .HG............
0000350: bfbc 0100 0000 0060 0240 00f0 3400 0002  .......`.@..4...
0000360: 0405 b400 0000 0000 5238 3e39 1908 000e  ........R8>9....
```

To be sure, we also find another interesting packet, packet 10, shown in the next example. It's important to use packets that have lengths that vary, to make it easier to determine which field in the packet header indicates length.

```
=====( packet received on interface en0 )=====Fri Nov 26 07:39:05 1999
ETHERNET packet : [ 00:e0:1e:a6:dc:e8 -> 08:00:5a:cd:ba:52 ]  type 800  (IP)
IP header breakdown:
        < SRC =  192.168.129.160 >
        < DST =  192.168.225.132 >
        ip_v=4, ip_hl=20, ip_tos=0, ip_len=60, ip_id=48148, ip_off=0(don't fragment)
        ip_ttl=60, ip_sum=9e31, ip_p = 6 (TCP)
TCP header breakdown:
        <source port=1301, destination port=113(auth) >
        th_seq=eeb744f6, th_ack=0
        th_off=10, flags<SYN |>
        th_win=32120, th_sum=ab9a, th_urp=0
00000000     020405b4 0402080a 0151fff8 00000000     |...´.....Q.ø....|
00000010     01030300                                |....            |

 offset  00   02   04   06   08   0a   0c   0e
 offset  01   03   05   07   09   0b   0d   0f

0000410: f600 0000 0000 0000 0000 0000 6038 3e39  ............`8>9
0000420: 1908 000e 0065 6e00 0001 4575 1001 4594  .....en...Eu..E.
0000430: 5000 0000 0006 0008 005a cdba 5200 e01e  P........Z..R...
0000440: a6dc e808 0045 0000 3cbc 1440 003c 069e  .....E..<..@.<..
0000450: 31c0 a881 a0c0 a8e1 8405 1500 71ee b744  1...........q..D
0000460: f600 0000 00a0 027d 78ab 9a00 0002 0405  .......}x.......
0000470: b404 0208 0a01 51ff f800 0000 0001 0303  ......Q.........
0000480: 0000 0000 5238 3e39 1900 0000 0065 6e00  ....R8>9.....en.
```

Looking at the hex dumps we see the string *en* in the ASCII shown to the right. Because *en0* is the name of the interface for each packet, our suspicion is that bytes 13 and 14 are involved with recording the interface name. However, the number of the interface, 0 for en0, is not visible in the ASCII. Perhaps the hex values after en, or byte 15, is the number of the interface. It would require more packet capture files, with varying interface names and numbers to confirm this suspicion.

The analysis of the data locations and size calculation is not shown, but the results, showing the first three packets, and packets 7 and 10, are shown in Table 11.13. The header data is summarized in Table 11.14.

Table 11.13 All Metadata Summarized

Packet Number	Data Length	Time Stamp	Interface	Direction
1	0x62	Fri Nov 26 07:38:57 1999	en0	Transmit
2	0x62	Fri Nov 26 07:38:57 1999	en0	Receive
3	0x62	Fri Nov 26 07:38:58 1999	en0	Transmit
7	0x3c	Fri Nov 26 07:39:05 1999	en0	Transmit
10	0x4a	Fri Nov 26 07:39:05 1999	en0	Receive

Table 11.14 All Packet Header Data Bytes

Packet Number	Header Data
1	00 00 00 78 38 3e 39 11 00 00 00 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 01

Continued

Table 11.14 continued All Packet Header Data Bytes

Packet Number	Header Data
2	00 00 00 78 38 3e 39 11 08 00 0e 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 00
3	00 00 00 78 38 3e 39 12 00 00 00 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 01
7	00 00 00 52 38 3e 39 19 00 00 00 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 01
10	00 00 00 60 38 3e 39 19 08 00 0e 00 65 6e 00 00 01 45 75 10 01 45 94 50 00 00 00 00 06 00

Immediately some interesting facts show themselves. Table 11.15 shows that byte 8 in the header differs between each packet by the number of seconds that differ between the time stamps in each packet. There's a good chance byte 8 is involved in recording the time stamp.

Table 11.15 Time Stamp Differences

Packet	Time Stamp	Seconds Since Previous Time Stamp	Byte 8	Difference
1	Fri Nov 26 07:38:57 1999	n/a	0x11	n/a
2	Fri Nov 26 07:38:57 1999	0	0x11	0
3	Fri Nov 26 07:38:58 1999	1	0x12	1
7	Fri Nov 26 07:39:05 1999	7	0x19	7
10	Fri Nov 26 07:39:05 1999	0	0x19	0

Table 11.16 shows that the last byte in the header, byte 30, toggles between 0x00 and 0x01, with the same pattern of the transmit and receive values.

Table 11.16 Direction Values

Packet	Direction	Byte 30
1	Transmit	01
2	Receive	00
3	Transmit	01
7	Transmit	01
10	Receive	00

Byte 4 in the header is the same for the first three packets, but different for the last packets. The difference between the values in byte 4 is the same as the difference between the packet data lengths (Table 11.17).

Table 11.17 Length Field Differences

Packet	Data Length	Difference from Previous Data Length	Byte 4	Difference from Previous Byte 4
1	0x62	n/a	0x78	n/a
2	0x62	0	0x78	0
3	0x62	0	0x78	0
7	0x3c	-0x26	0x52	-0x26
10	0x4a	0xe	0x60	0xe

The difference between byte 4 values is constant, in the same way that the difference between data lengths is constant. It appears that that byte 4 encodes the packet data length as the data length plus some constant:

```
(Data Length) + (Some Unknown Constant) = (Value of Byte 4)
```

To find the unknown constant, simply subtract the value of byte 4 from the packet data length for each packet (see Table 11.18):

```
(Some Unknown Constant) = (Value of Byte 4) - (Data Length)
```

Table 11.18 Data Length Constant Calculations

Packet	Byte 4 Value	Data Length	Calculated Constant
1	0x78	0x62	0x16
2	0x78	0x62	0x16
3	0x78	0x62	0x16
7	0x52	0x3c	0x16
10	0x60	0x4a	0x16

Our suspicion is confirmed. Byte 4 stores the length of the packet data, plus 0x16. Table 11.19 shows what we know so far of the format of the packet header.

Table 11.19 Packet Header Information

Byte(s)	Use
4	Data length + 0x16
8	Time stamp
13 – 14	Interface name
30	Direction

To further map out the format of the packet header, we need to remember how computers store integer values. Each byte can hold 256 values, from 0x00 to 0xff, or 0 to 255. To count higher than 255, a number has to be stored in multiple bytes. Table 11.20 shows the number of values that a particular number of bytes can represent.

Table 11.20 Integer Sizes

Bytes	Formula	Number of Values
1	2^{8}	256
2	2^{16}	65,536
3	2^{24}	16,777,216
4	2^{32}	4,294,967,296

Since packets can have more than 256 bytes of data, we know that byte 4 in the packet header cannot be the only byte that is used to represent the length of the packet. Furthermore, it is easy to see from the hex dumps that bytes 5 through 7 have a nonzero value that seems to be constant across packets. Those bytes seem to be part of a number whose last byte, byte 8, varies with the number of seconds. These facts, plus the fact that we know that using 4 bytes to represent an integer is very common because many processors are 32-bit CPUs, where 32-bits means 4 bytes, allows us to guess the following field lengths in Table 11.21.

Table 11.21 Hypothesized Field Lengths

Bytes	Use
1 – 4	Data length
5 – 8	Time stamp

Table 11.22 focuses in on those bytes in the sample packets.

Table 11.22 Length and Time Stamp Bytes

Packet	Data Length	Time Stamp	Header Bytes 1-8
1	0x62	Fri Nov 26 07:38:57 1999	00 00 00 78 38 3e 39 11
2	0x62	Fri Nov 26 07:38:57 1999	00 00 00 78 38 3e 39 11
3	0x62	Fri Nov 26 07:38:58 1999	00 00 00 78 38 3e 39 12
7	0x3c	Fri Nov 26 07:39:05 1999	00 00 00 52 38 3e 39 19
10	0x4a	Fri Nov 26 07:39:05 1999	00 00 00 60 38 3e 39 19

If bytes 1 through 4 represent a single 32-bit (4-byte) integer, then we can tell that the integer is *big endian*. This shouldn't come as a surprise to us, as the PowerPC processors that run AIX are big-endian CPUs. To understand the term big endian and its opposite, *little endian*, you must understand how computers store multiple-byte integers into memory. A 32-bit number 0x78, can be stored in memory in two ways, shown in Table 11.23.

Table 11.23 0x78 Stored Two Ways

Number	Big Endian	Little Endian
0x78	00 00 00 78	78 00 00 00

Choosing a big endian representation in our file format makes bytes 1 through 4 work for us. To be sure, you must find a packet that has more than 256 bytes of data in it and see what bytes 1 through 4 look like. An example won't be shown here, but suffice it to say that our hypothesis is correct. Applying that fact to bytes 5 through 8, we surmise that the time stamps are also big-endian integers, shown in Table 11.24.

Table 11.24 Time Stamp Integers

Packet	Time Stamp	Header Bytes 5-8	Big Endian Integer
1	Fri Nov 26 07:38:57 1999	38 3e 39 11	943,601,937
2	Fri Nov 26 07:38:57 1999	38 3e 39 11	943,601,937
3	Fri Nov 26 07:38:58 1999	38 3e 39 12	943,601,938
7	Fri Nov 26 07:39:05 1999	38 3e 39 19	943,601,945
10	Fri Nov 26 07:39:05 1999	38 3e 39 19	943,601,945

It is obvious that the 4-byte integer that represents the time stamp is an offset from some time in the past. Since the ipreport analysis of the iptrace file suggests that the time resolution is only 1 second, and our integer value indicates one-second differences, the time stamp integer must represent the number of seconds since some beginning point in time. The C library has routines to store as the number of seconds since the Epoch, which is 00:00:00 UTC, January 1, 1970. Our first guess as to what time zero in the iptrace file is should be the C library Epoch, because iptrace runs on UNIX computers, and they use the C library. To test this hypothesis, we will use a small program that loads the time stamp value from packet 1 into a variable and runs the C library *ctime* command to see the character representation of the time stamp.

```
#include <stdio.h>
#include <time.h>

int
main(void)
{
    char *text;
    time_t ts;

    ts = 0x383e3911;
    text = ctime(&ts);

    printf("%u is %s\n", ts, text);

    return 0;
}
```

Running this program returns a result that is almost the expected value:

```
$ ./test-timestamp
943601937 is Fri Nov 26 01:38:57 1999
```

You have to be sure to set your time zone to UTC. The *ctime* function reports a perfect match in that case:

```
$ TZ=UTC ./test-timestamp
943601937 is Fri Nov 26 07:38:57 1999
```

Luckily for us, the iptrace time stamp is compatible with the C library *time_t* value. It's the number of seconds since the Epoch. That will make writing our wiretap module to read iptrace files that much easier.

Adding a Wiretap Module

Ethereal uses the wiretap library to read a capture file in three distinct steps. Its useful to know that Ethereal keeps metadata from all packets in memory, but the packet data is only read when needed. That's why the wiretap module must provide the ability to read a packet capture file in a random-access fashion:

1. The capture file is opened. Wiretap determines the file type.
2. Ethereal reads through all packets sequentially, recording metadata for each packet. If color filters or read filters are set, the packet data is dissected at this time, too.
3. As the user selects packets in the GUI (graphical user interface), in a random access fashion, Ethereal will ask wiretap to read that packet's data.

To add a new file format to the wiretap library, you create a new C file in the wiretap directory of the Ethereal source distribution. This new wiretap module plugs into wiretap's mechanism for detecting file types. The new module is responsible for being able to recognize the file format by reading a few bytes from the beginning of the file. The wiretap library distinguishes file formats by examining the contents at the beginning of the file, instead of using a superficial method like using a file name suffix as a key to the file type.

To start, add a new file type macro to the list of *WTAP_FILE* macros in the wtap.h file. Choose a name that is related to your file, and set its value to be one greater than the last *WTAP_FILE* macro. Also increase the value of *WTAP_NUM_FILE_TYPES* by one.

The module_open Function

In your new module, write a routine for detecting the file type. The functions in your new module should be prefixed with a name that distinguishes your module from others. The function that detects file types is called the *open* function in wiretap, so your module's *open* function should be named *module_open*, where *module* is the prefix you choose for the functions. For example, the functions in the iptrace.c wiretap module are prefixed with the name *iptrace*.

You should have a module.h file that gives the prototype for your *open* function. To plug your new module into wiretap, you must modify the file_access.c file in wiretap. First *#include* your module.h file from file_accesss.c. Then add your module's *open* routine to the array *open_routines*.

The comments inside that array identify two sections to the array. The first part of the list has the modules that look for identifying values at fixed locations in the file. The second part of the list has modules that scan the beginning of the file looking for certain identifying values somewhere in the file. You should list your module's *open* routine in the appropriate section.

Then modify the *dump_open_table* array in file_access.c. It contains, in order, names and pointers for each file format. The structure is as follows:

```
    const char *name;
    const char *short_name;
    int (*can_write_encap)(int);
    int (*dump_open)(wtap_dumper *, gboolean, int *);
```

The *name* field gives a long descriptive name that is useful in a GUI. The *short_name* field gives a short unique name that is useful in a command-line-based program. The *can_write_encap* and *dump_open functions* are used if your wiretap module can write files. This chapter won't describe writing files, as the intent is to have wiretap read new file formats. But if you are extending your wiretap module to write files, the *can_write_encap* function lets Ethereal know if your file format can handle a particular encapsulation type. The *dump_open* function is the function in your module that opens a file for writing.

Your open routine has this function prototype:

```
int module_open(wtap *wth, int *err, gchar **err_info);
```

The return value of *module_open* is one of three values (see Table 11.25).

Table 11.25 module_open Return Values

Value	Meaning
-1	An I/O error occurred. Wiretap will discontinue trying to read the file.
0	No I/O error occurred, but the file is not of the right format.
1	The file format is correct for this module.

The *wtap* struct is the data structure that wiretap uses to store data about a capture file. The *err* variable is for your function to return error codes to the program that called wiretap. The *err_info* variable is the way for the error code returned in *err* to be accompanied by additional information.

The layout of the *wtap* struct is as follows:

```
struct wtap {
    FILE_T          fh;
    int             fd;           /* File descriptor for cap file */
    FILE_T          random_fh;    /* Secondary FILE_T for random access */
    int             file_type;
    int             snapshot_length;
    struct Buffer       *frame_buffer;
    struct wtap_pkthdr  phdr;
    union wtap_pseudo_header pseudo_header;

    long            data_offset;
    union {
        libpcap_t       *pcap;
        lanalyzer_t     *lanalyzer;
        ngsniffer_t     *ngsniffer;
        i4btrace_t      *i4btrace;
        nettl_t         *nettl;
        netmon_t        *netmon;
        netxray_t       *netxray;
        ascend_t        *ascend;
        csids_t         *csids;
        etherpeek_t     *etherpeek;
        airopeek9_t     *airopeek9;
        erf_t           *erf;
        void            *generic;
    } capture;

    subtype_read_func   subtype_read;
    subtype_seek_read_func  subtype_seek_read;
    void            (*subtype_sequential_close)(struct wtap*);
    void            (*subtype_close)(struct wtap*);
    int         file_encap; /* per-file, for those
                            file formats that have
                            per-file encapsulation
                            types */
};
```

When wiretap is attempting to identify a capture file format, it will call all the functions listed in the *open_routines* array in file_access.c. When your *module_open* function is called, it will be able to use the *fh* member of the *wtap* struct. It is an open file handle set at the beginning of the file. The *FILE_T* type is a special file handle type. It is used like the C library *FILE* type, but if Ethereal, and thus wiretap, is linked with the zlib compression library, which it normally is, then the *FILE_T* type gives wiretap the ability to read compressed files. The zlib compression library decompresses the file on the fly, passing decompressed chunks to wiretap. The functions to use *FILE_T* types are similar to those for using *FILE* types, but the functions are prefixed with *file_* instead of *f*. These functions are listed in file_wrappers.h, and are summarized in Table 11.26.

Table 11.26 FILE_T Functions

stdio FILE function	Wiretap FILE_T function
open	file_open
dopen	filed_open
seek	file_seek
read	file_read
write	file_write
close	file_close
tell	file_tell
getc	file_getc

Continued

Table 11.26 continued FILE_T Functions

stdio FILE function	Wiretap FILE_T function
gets	file_gets
eof	file_eof
n/a	file_error

The *file_error* function is specific to wiretap. It returns a wiretap error code for an I/O (input/output) stream. If no error has occurred, it returns 0. If a file error occurs, an *errno* value is returned. Any other error causes *file_error* to return a *WTAP_ERR* code, which is defined in wtap.h.

To read the iptrace 1.0 file format, for example, the first 11 bytes of the file must be read and compared with the string iptrace 1.0. That's easy. The more difficult part is remembering to check for errors while reading the file and to set all appropriate error-related variables. To be safe, use the standard boilerplate code that sets *errno*, calls *file_read*, then checks for either an error condition or simply the fact that the file was too small to contain the requested number of bytes.

```
/* Sets errno in case we return an error */
errno = WTAP_ERR_CANT_READ;

/* Read 'num_recs' number of records, each 'rec_size' bytes long. */
bytes_read = file_read(destination, rec_size, num_recs, wth->fh);

/* If we didn't get 'size' number of bytes... */
if (bytes_read != size) {
    *err = file_error(wth->fh);
    /* ...if there was an error, return -1 */
    if (*err != 0)
        return -1;
    /* ...otherwise, the file simply didn't have 'size' number of bytes.
    It can't be our file format, so return 0. */
    return 0;
}
```

To see how this works in practice, the following example shows how *iptrace_open* would look. Notice how the *data_offset* member of wtap is incremented after the call to *file_read*. The *data_offset* variable will be used during the sequential read of the capture file. If *iptrace_open* detects that the file is an iptrace 1.0 file, then three members of the *wtap* struct are set: *file_type*, *subtype_read*, and *subtype_seek_read*.

```
#define IPTRACE_VERSION_STRING_LENGTH   11

int
iptrace_open(wtap *wth, int *err, gchar **err_info)
{
    int bytes_read;
    char name[12];

    errno = WTAP_ERR_CANT_READ;
    bytes_read = file_read(name, 1, IPTRACE_VERSION_STRING_LENGTH, wth->fh);
    if (bytes_read != IPTRACE_VERSION_STRING_LENGTH) {
        *err = file_error(wth->fh);
        if (*err != 0)
            return -1;
        return 0;
    }
    wth->data_offset += IPTRACE_VERSION_STRING_LENGTH;
    name[IPTRACE_VERSION_STRING_LENGTH] = 0;

    if (strcmp(name, "iptrace 1.0") == 0) {
        wth->file_type = WTAP_FILE_IPTRACE_1_0;
        wth->subtype_read = iptrace_read;
        wth->subtype_seek_read = iptrace_seek_read;
        wth->file_encap = WTAP_ENCAP_PER_PACKET;
    }
    else {
        return 0;
    }

    return 1;
}
```

Some capture file formats allow each packet to have a separate link layer, or encapsulation type. Other file formats allow only one type per file. Since the interface name is given in the packet header in the iptrace file format that we investigated, the encapsulation type in this file format is per-packet. So we set the file encapsulation type to *WTAP_ENCAP_PER_PACKET* to indicate that.

The module_read Function

The *subtype_read* function is used when the capture file is initially opened. Ethereal will read all packet records in the capture file, sequentially. The *subtype_seek_read* function is the random access function that is called when an Ethereal user selects a packet in the GUI.

The following code represents the *subtype_read* function prototype:

```
static gboolean
module_read(wtap *wth, int *err, gchar **err_info, long *data_offset);
```

The first three arguments are the same as in *module_open*. The *long* data_offset* argument is the way for *module_read* to send the offset of the packet record to Ethereal. It should point to the packet's record, including metadata, within the capture file. This offset will be passed to the random access function later, if the user selects the packet in the GUI.

Additional metadata about the packet is returned to Ethereal via the *phdr* member of the *wtap* struct. The *phdr*, or packet header, member is a *wtap_pkthdr* struct. Its definition is as follows:

```
struct wtap_pkthdr {
    struct timeval ts;  /* Timestamp */
    guint32 caplen;     /* Bytes captured in file */
    guint32 len;        /* Bytes on wire */
    int pkt_encap;      /* Encapsulation (link-layer) type */
};
```

The time stamp value records when the packet was recorded. The *timeval* struct that is used is defined in system header files as a two-member struct, recording seconds and microseconds.

```
struct timeval {
    int32_t tv_sec;     /* seconds since Epoch */
    int32_t tv_usec;    /* microseconds since second*/
};
```

The *caplen* member represents how many bytes of the packet are present in the capture file. This value will be less than or equal to the *len* value, which is the number of bytes of the packet present on the wire. The reason for two separate length values is that some tools, like tcpdump, allow you to capture only a portion of the packet. This is useful if you want to capture many packets, but only need the first few bytes of them, perhaps to analyze TCP headers, but not the payload.

The *pkt_encap* variable signifies the first protocol in the packet payload. This can be called the link layer, or more generally, the encapsulation type. This value should be a *WTAP_ENCAP* value. These are defined in wtap.h. The *pkt_encap* value is the value that Ethereal uses to begin dissection of the packet data.

The *module_read* function returns *TRUE* if a packet was read, or *FALSE* if not. A *FALSE* may be returned on an error, or if the end of a file has been reached.

A *module_read* function template looks like this:

```
/* Read the next packet */
static gboolean
module_read(wtap *wth, int *err, gchar **err_info,
    long *data_offset)
{
    /* Set the data offset return value */
    *data_offset = wth->data_offset;

    /* Read the packet header */
    /* Read the packet data */
    /* Set the phdr metadata values */

    return TRUE;
}
```

To handle reading the packet header and data, a helper function will be used that reads data and sets the error codes appropriately. This function returns -1 on an error, 0 on end of file, and 1 on success.

```
static int
iptrace_read_bytes(FILE_T fh, guint8 *dest, int len, int *err)
{
    int bytes_read;

    errno = WTAP_ERR_CANT_READ;
    bytes_read = file_read(dest, 1, len, fh);
    if (bytes_read != len) {
        *err = file_error(fh);
        if (*err != 0)
```

```
            return -1;
        if (bytes_read != 0) {
            *err = WTAP_ERR_SHORT_READ;
            return -1;
        }
        return 0;
    }
    return 1;
}
```

Then we define some helpful macros values to aid in reading the iptrace `packet header.`

```
#define IPTRACE_1_0_PHDR_LENGTH_OFFSET      0
#define IPTRACE_1_0_PHDR_TVSEC_OFFSET       4
#define IPTRACE_1_0_PHDR_IF_NAME_OFFSET     12
#define IPTRACE_1_0_PHDR_DIRECTION_OFFSET   29

#define IPTRACE_1_0_PHDR_SIZE               30

#define IPTRACE_1_0_PHDR_LENGTH_CONSTANT    0x16

#define ASCII_e                             0x65
#define ASCII_n                             0x6e
```

We define the offset macros instead of defining a struct, which corresponds to the packet header because the architecture of the machine that is reading the iptrace file may not be the same as that of the machine that wrote the file. You never know what the compiler is going to do to your struct with regards to field alignments. It's safer to pull the values out of the header one by one than trying to align a struct to the header layout.

To read the packet header, our function evolves to the following:

```
/* Read the next packet */
static gboolean
iptrace_read(wtap *wth, int *err, gchar **err_info,
    long *data_offset)
{
    int ret;
    guint8  header[IPTRACE_1_0_PHDR_SIZE];

    /* Set the data offset return value */
    *data_offset = wth->data_offset;

    /* Read the packet header */
    ret = iptrace_read_bytes(wth->fh, header,
        IPTRACE_1_0_PHDR_SIZE, err);
    if (ret <= 0) {
        /* Read error or EOF */
        return FALSE;
    }
    wth->data_offset += IPTRACE_1_0_PHDR_SIZE;

    /* Read the packet data */
    /* Set the phdr metadata values */

    return TRUE;
}
```

Now that the packet header has been read into the header array, we can read the packet length from the header. To convert the series of 4 bytes, arranged in big endian order, also known as network order, use the *pntohl* macro. The letters *pntohl* stand for pointer, network to host, long. By *long*, the macro means 32 bits, or 4 bytes. The abbreviations used to name the macros are listed in Table 11.27. The collection of macros in wtap-int.h is summarized in Table 11.28.

Table 11.27 Pointer-to-Integer Macro Abbreviations

Abbreviation	Meaning
p	Pointer
n	Network order, big endian
le	Little endian
to	"to"
h	Host order, usable by the host CPU
s	Short, 2 bytes
24	24 bytes, or 3 bytes
l	Long, 4 bytes
ll	Double long, 8 bytes

Table 11.28 Pointer-to-Integer Macros

Bytes	Big Endian	Little Endian
2	pntohs	pletohs
3	pntoh24	pletoh24
4	pntohl	pletohl
8	pntohll	pletohll

To extend our read function to read packet data, we convert the packet length with *pntohl*, subtract the constant 0x16 that is added to the length, and read that number of bytes. The bytes for the packet data are read into the *frame_buffer* member of the *wtap* struct. The *frame_buffer* member is a *Buffer* struct, a resizable array of bytes that is part of the wiretap library. To deal with the *frame_buffer*, you need to know only two functions (see Table 11.29).

Table 11.29 Buffer Functions

Function	Use
buffer_assure_space	Ensures that there's enough free space in the buffer for new data of a known length to be copied to it.
buffer_start_ptr	Returns the pointer where you can start copying data into it.

Combining the pointer-to-integer macros and the buffer function calls, our *iptrace_read* function can now read data.

```
/* Read the next packet */
static gboolean
iptrace_read(wtap *wth, int *err, gchar **err_info,
    long *data_offset)
{
    int ret;
    guint8  header[IPTRACE_1_0_PHDR_SIZE];
    guint32 packet_len;
    guint8  *data_ptr;

    /* Set the data offset return value */
    *data_offset = wth->data_offset;

    /* Read the packet header */
    ret = iptrace_read_bytes(wth->fh, header,
        IPTRACE_1_0_PHDR_SIZE, err);
    if (ret <= 0) {
        /* Read error or EOF */
        return FALSE;
    }
    wth->data_offset += IPTRACE_1_0_PHDR_SIZE;

    /* Read the packet data */
    packet_len = pntohl(&header[IPTRACE_1_0_PHDR_LENGTH_OFFSET]) -
        IPTRACE_1_0_PHDR_LENGTH_CONSTANT;

    buffer_assure_space(wth->frame_buffer, packet_len);
    data_ptr = buffer_start_ptr(wth->frame_buffer);

    ret = iptrace_read_bytes(wth->fh, data_ptr, packet_len, err);
    if (ret <= 0) {
        /* Read error or EOF */
        return FALSE;
    }
    wth->data_offset += packet_len;

    /* Set the phdr metadata values */

    return TRUE;
}
```

Finally, the metadata is set in the *phdr* member of the *wtap* struct. Because the iptrace file doesn't distinguish between the number of bytes originally in a packet and the number of bytes captured from the packet, the *len* and *caplen* values are set to the same value. We haven't investigated enough iptrace files to fully know how the encapsulation type is encoded, but so far we know that if the interface name begins with *en* then the encapsulation type is Ethernet. In the future, when we investigate iptrace files of other encapsulation types, we can refine the iptrace_read function. The following example shows the final evolution of the *iptrace_read* function. Notice how we can set the time stamp value without any modification because the time stamp is already the integer number of seconds since the C library Epoch. The iptrace file does not have microsecond resolution, so *tv_usec* is set to 0.

```
/* Read the next packet */
static gboolean
iptrace_read(wtap *wth, int *err, gchar **err_info,
    long *data_offset)
{
    int ret;
    guint8  header[IPTRACE_1_0_PHDR_SIZE];
    guint32 packet_len;
    guint8  *data_ptr;

    /* Set the data offset return value */
    *data_offset = wth->data_offset;

    /* Read the packet header */
    ret = iptrace_read_bytes(wth->fh, header,
        IPTRACE_1_0_PHDR_SIZE, err);
    if (ret <= 0) {
        /* Read error or EOF */
        return FALSE;
    }
    wth->data_offset += IPTRACE_1_0_PHDR_SIZE;

    /* Read the packet data */
    packet_len = pntohl(&header[IPTRACE_1_0_PHDR_LENGTH_OFFSET]) -
        IPTRACE_1_0_PHDR_LENGTH_CONSTANT;

    buffer_assure_space(wth->frame_buffer, packet_len);
    data_ptr = buffer_start_ptr(wth->frame_buffer);

    ret = iptrace_read_bytes(wth->fh, data_ptr, packet_len, err);
    if (ret <= 0) {
        /* Read error or EOF */
        return FALSE;
    }
    wth->data_offset += packet_len;

    /* Set the phdr metadata values */
    wth->phdr.len = packet_len;
    wth->phdr.caplen = packet_len;
    wth->phdr.ts.tv_sec = pntohl(&header[IPTRACE_1_0_PHDR_TVSEC_OFFSET]);
    wth->phdr.ts.tv_usec = 0;

    if (header[IPTRACE_1_0_PHDR_IF_NAME_OFFSET] == ASCII_e &&
        header[IPTRACE_1_0_PHDR_IF_NAME_OFFSET+1] == ASCII_n) {

        wth->phdr.pkt_encap = WTAP_ENCAP_ETHERNET;
    }
    else {
        /* Unknown encapsulation type */
        wth->phdr.pkt_encap = WTAP_ENCAP_UNKNOWN;
    }

    return TRUE;
}
```

The module_seek_read Function

The *subtype_seek_read* function in a module provides the means for Ethereal to request a specific packet in the capture file. The prototype for the *subtype_seek_read* function is substantially different from that of the *subtype_read* function.

```
static gboolean
module_seek_read(wtap *wth, long seek_off,
    union wtap_pseudo_header *pseudo_header, guchar *pd, int packet_size,
    int *err, gchar **err_info);
```

Table 11.30 lists the meanings of those arguments.

Table 11.30 subtype_seek_read Arguments

Argument	Meaning
wth	The wtap struct that represents the file.
seek_off	The offset of the packet record that is being requested.
pseudo_header	A structure that holds additional data for some encapsulation types that have to send more information to Ethereal.
pd	The byte array where the packet data should be copied.
packet_size	The size of the packet data. This was recorded during the run of the subtype_read function.
err	Means to pass error condition to caller.
err_info	Means to pass error string to caller.

The return value of *module_seek_read* is either *TRUE* or *FALSE*, indicating success or failure.

A *module_seek_read* function template looks like this.

```
/* Seek and read a packet */
static gboolean
module_seek_read(wtap *wth, long seek_off,
    union wtap_pseudo_header *pseudo_header, guchar *pd, int packet_size,
    int *err, gchar **err_info);
{
    /* Seek to the proper file offset */
    /* Read the packet header if necessary */
    /* Read the packet data */
    /* Fill in the pseudo_header, if necessary */

    return TRUE;
}
```

In the *module_seek_read* function, the *random_fh FILE_T* variable is used instead of the *fh FILE_T* variable. This allows the user to select packets to look at while Ethereal is also capturing packets and updating its GUI to show them. The functions for reading from *random_fh* are the same as those for reading from *fh*. This code shows how we seek and read.

```
/* Seek and read a packet */
static gboolean
iptrace_seek_read(wtap *wth, long seek_off,
    union wtap_pseudo_header *pseudo_header, guchar *pd, int packet_size,
    int *err, gchar **err_info)
{
    int ret;
    guint8          header[IPTRACE_1_0_PHDR_SIZE];
    int pkt_encap;

    /* Seek to the proper file offset */
    if (file_seek(wth->random_fh, seek_off, SEEK_SET, err) == -1)
        return FALSE;

    /* Read the packet header if necessary. We need to read it to find
    the encapsulation type for this packet. */
    ret = iptrace_read_bytes(wth->random_fh, header,
        IPTRACE_1_0_PHDR_SIZE, err);
    if (ret <= 0) {
        /* Read error or EOF */
        if (ret == 0) { /* EOF */
            *err = WTAP_ERR_SHORT_READ;
        }
        return FALSE;
    }

    /* Read the encapsulation type.
    if (header[IPTRACE_1_0_PHDR_IF_NAME_OFFSET] == ASCII_e &&
        header[IPTRACE_1_0_PHDR_IF_NAME_OFFSET+1] == ASCII_n) {

        pkt_encap = WTAP_ENCAP_ETHERNET;
    }
    else {
        /* Unknown encapsulation type */
        return FALSE;
    }

    /* Read the packet data. We'll use 'packet_size' instead of
    retrieving the packet length from the packet header. */
    ret = iptrace_read_bytes(wth->random_fh, pd, packet_size, err);
    if (ret <= 0) {
        /* Read error or EOF */
        if (ret == 0) { /* EOF */
            *err = WTAP_ERR_SHORT_READ;
        }
        return FALSE;
    }

    /* Fill in the pseudo_header, if necessary */
    return TRUE;
}
```

Wiretap's pseudo-header mechanism allows the encapsulation protocol to return additional information to Ethereal. The definition of the *wtap_pseudo_header* union, in wtap.h, lists the different encapsulations that have such additional information.

```
union wtap_pseudo_header {
    struct eth_phdr     eth;
    struct x25_phdr     x25;
    struct isdn_phdr    isdn;
    struct atm_phdr     atm;
    struct ascend_phdr  ascend;
    struct p2p_phdr     p2p;
    struct ieee_802_11_phdr ieee_802_11;
    struct cosine_phdr  cosine;
    struct irda_phdr    irda;
};
```

The Ethernet protocol has a pseudo-header. That pseudo header struct is also defined in wtap.h.

```
/* Packet "pseudo-header" information for Ethernet capture files. */
struct eth_phdr {
    gint    fcs_len;    /* Number of bytes of FCS - -1 means "unknown" */
};
```

The FCS (frame check sequence) bytes are extra bytes that are added to the actual transmission over the Ethernet cable to detect transmission errors. In most cases the host operating system strips those bytes before the packet analyzer program sees them, but some packet analyzers do record the FCS bytes. The Ethernet pseudo-header lets Ethereal know if there are any of these extra bytes. The iptrace file does not contain them, so we must set fcs_len to 0. The following example shows the final version of iptrace_seek_read:

```
/* Seek and read a packet */
static gboolean
iptrace_seek_read(wtap *wth, long seek_off,
    union wtap_pseudo_header *pseudo_header, guchar *pd, int packet_size,
    int *err, gchar **err_info)
{
    int ret;
    guint8          header[IPTRACE_1_0_PHDR_SIZE];
    int pkt_encap;

    /* Seek to the proper file offset */
    if (file_seek(wth->random_fh, seek_off, SEEK_SET, err) == -1)
        return FALSE;

    /* Read the packet header if necessary. We need to read it to find
    the encapsulation type for this packet. */
    ret = iptrace_read_bytes(wth->random_fh, header,
        IPTRACE_1_0_PHDR_SIZE, err);
    if (ret <= 0) {
        /* Read error or EOF */
        if (ret == 0) { /* EOF */
            *err = WTAP_ERR_SHORT_READ;
        }
        return FALSE;
    }

    /* Read the encapsulation type. We don't have to return this
    to Ethereal, because it already knows it. But we don't have
    that information handy. We have to re-retrieve that value
    from the packet header. */
    if (header[IPTRACE_1_0_PHDR_IF_NAME_OFFSET] == ASCII_e &&
        header[IPTRACE_1_0_PHDR_IF_NAME_OFFSET+1] == ASCII_n) {

        pkt_encap = WTAP_ENCAP_ETHERNET;
    }
    else {
        /* Unknown encapsulation type */
        return FALSE;
    }

    /* Read the packet data. We'll use 'packet_size' instead of
    retrieving the packet length from the packet header. */
    ret = iptrace_read_bytes(wth->random_fh, pd, packet_size, err);
    if (ret <= 0) {
        /* Read error or EOF */
        if (ret == 0) { /* EOF */
            *err = WTAP_ERR_SHORT_READ;
        }
        return FALSE;
    }

    /* Fill in the pseudo_header, if necessary */
    if (pkt_encap == WTAP_ENCAP_ETHERNET) {
        pseudo_header->eth.fcs_len = 0;
    }

    return TRUE;
}
```

If your *module_read* or *module_seek_read* functions need additional information about the file in order to process packets, the wtap struct can be extended by defining a structure type and adding it to the capture union. The capture union in the struct wtap shows that many file formats do save extra information.

```
    union {
        libpcap_t       *pcap;
        lanalyzer_t     *lanalyzer;
        ngsniffer_t     *ngsniffer;
        i4btrace_t      *i4btrace;
        nettl_t         *nettl;
        netmon_t        *netmon;
        netxray_t       *netxray;
        ascend_t        *ascend;
        csids_t         *csids;
        etherpeek_t     *etherpeek;
        airopeek9_t     *airopeek9;
        erf_t           *erf;
        void            *generic;
    } capture;
```

The module_close Function

When your file format allocates memory in this capture union, your wiretap module has to provide *close* functions to properly free that memory. As there were two *open* functions, one for sequential and one for random access, there are two *close* functions:

```
void    (*subtype_sequential_close)(struct wtap*);
void    (*subtype_close)(struct wtap*);
```

If your module does not need them, as the iptrace module does not, then those two fields in the wtap struct are left alone. If your module needs them, then during the *module_open* function, they should be set to point to your functions, in the same manner as *subtype_read* and *subtype_seek_read* are dealt with.

Building Your Module

To integrate your new wiretap module into the wiretap library, it must be added to the list of files to be built. Edit the makefile.common file in the wiretap directory of the Ethereal distribution. Add your module.c file to the *NONGENERATED_C_FILES* list and add your module.h file to the *NONGENERATED_HEADER_FILES* list. Both the UNIX build and the Windows build use the lists in makefile.common. You can use the normal Ethereal build procedure; wiretap will build and include your module.

Final Touches

You have learned three ways of feeding data into Ethereal. If you have an application that has the opportunity to deal with network interfaces, you can use libpcap to capture packets and save them to a file. text2pcap is a tool that will convert from hex dumps to the pcap format. You have seen the range of hex dump formats that text2pcap will accept, and how to produce a hex dump format from another file. Finally, you not only learned how to extend the wiretap library so that Ethereal can read a new file format natively but also saw a practical example of how to reverse engineer a packet capture file format for which you had no documentation.