CPE464 – Program #1 – Packet Trace

See Canvas page for Program #1 – Trace for due date and handin information

For this program you will write a packet sniffer program called *trace* in C or C++. This program will output protocol header information for a number of different types of headers. To write this program you will use a set of pcap API functions that allows you to sniff packets[1]. For this assignment you will only be "sniffing" packets from **trace files** that I already created. Your program must run on the CSL Linux servers. (I highly recommend you install the needed software on your home Linux box (e.g. Windows WSL 2) or Mac and do your development at home. BUT your program MUST work on the CSL Linux machines.)

The pcap library is an API that allows for the retrieval of packets from either a networking interface (NIC) or a file. This library is used by wireshark (you will use wireshark in the lab) to retrieve packets. Because you need root privileges to retrieve packets from the network (and you do not have root on the CSL machines!) you will write your program to retrieve packets from packet trace files previously captured by wireshark. For more information on pcap see: http://www.tcpdump.org/

Your program, called **trace**, will take one parameter. This is the file that contains the packet traces. This file will be in a format readable by the pcap library.

To run the program on one of our Unix machines (**note that the program output goes to STDOUT**):

```
-bash-4.1$ >  trace TraceFile.pcap
```

Types of headers you need to process:

- Ethernet Header
- ARP Header both Request and Reply
- IP
- ICMP Header for Echo request and Echo Reply
- TCP Header
- UDP Header

LOOK at these functions before you start!!!!

- Some pcap functions
  - pcap_open_offline(…)
  - pcap_next_ex(…)
  - pcap_close(…)

- inet_ntoa(…) (put IP address into a string for outputting)
- ether_ntoa(…) (put Mac address into a string for outputting)
- **Endianness conversion functions**: htons(…), htonl(…), ntohs(…), and ntohl(…)
- memcpy(…)

Notes:

[1] Wireshark uses the pcap library to sniff packets. Sniffing packets means reading packets on the network. In this assignment, you will read the network packets from a file – I previously read the packets off of the network using wireshark and stored them in the files.

1) Install wireshark (www.wireshark.org) onto your home computer. This will allow you to look at the trace files and help you debug your program. Note – wireshark is installed on all unix based computers in the CSL.

2) Do not use any code off the web or from other students. You may look at the sample code provided at www.tcpdump.org but do not copy code off of this site. The work you turn in must be your entirely your own. You may use the code I handed out for the Internet Checksum.

3) Do not sniff packets on an open network or in a CSL lab. You should generate your packet traces on a closed network (home network) or in the Cisco lab. Packet sniffing on an open Cal Poly lab network is against the Cal Poly Responsible Use Policy and will get you in trouble.

4) If you want to use a structure for the headers, you must create your own header structures for examble for Ethernet, IP, TCP, UDP, ARP if you need them. You cannot use structures for these headers (Ethernet, ARP, IP, TCP, UDP) created by others. Part of this assignment is for you to learn more about the different headers.

5) You cannot use any bit shifting ("<<" or ">>") in this program to handle endianness issues. You MUST use the standard conversion functions (e.g. ntohs(), htons()…) We will grep for these bit shifting operators and stop grading if we find them.

6) The output of your program must go to STDOUT (NOT a file). To capture your output use:

```
-bash-4.1$ >  trace TraceFile.pcap > anOutputFile.txt
```

7) The file checksum.c (you can find this on Canvas in the files for this program) contains the Internet Checksum function used by both IP and TCP. You must use this function. Do not write your own.

8) I have provided a sample set of packet traces. I may test your final program with additional traces. You need to create and turn in two additional test traces using wireshark.

9) You should not set a pcap filter.

10) You must provide a Makefile. This Makefile should create the program called trace. It should also provide a clean target. This clean target should delete object files and the executable file.

11) Trace files and sample output can be found on Canvas.

12) You will need to understand the *pseudo-header* to correctly verify the TCP and UDP checksums.

13) Your output must match my output. Your program will be graded by diff'ing your programs output with my programs output. Do not get creative! Do not add additional headers, options, fields… anything. My output is always correct (even if it is wrong).

14) For TCP, if the ACK flag is not set, then output <not valid> for the ACK Number.

What to turn in:

1) Makefile that correctly compiles your program on the CSL Unix machines without warnings.
2) All source code (include checksum.c and .h) needed to compile your program.
3) **Two new tracefiles that you created using wireshark.**
4) A readme file with comments for the TA. The readme should also include your lab time (9am, noon, 3 pm). This will help with grading.

Grading: (You must complete the early parts to receive any credit for a later part.)

1) (10%) Correctly read from the file and output the Ethernet headers
2) (10%) Part 1 and output the ARP headers
3) (30%) Part 2 and output the IP headers and ICMP
4) (50%) Part 3 and output the TCP and UDP headers

Note: Style points, bad programming, late points, error handing points also will be considered.

Your program must correctly parse ALL packets in the test files to receive credit for that part of the assignment.

I will provide trace files and output files on Canvas. Your program should produce output which diff's cleanly with my output. The whitespace in your output should be fairly close to my output including the spacing. For grading we will use the *–ignore-all-space* and the *–B* options when comparing your output to my sample output (so EXACT spacing is not required but it should look very close).

Note regarding the IP protocol field you should output correctly ICMP, TCP and UDP. For all other IP protocols you will just output "Unknown" and stop processing the packet.

Note for TCP/UDP Port number output – if in the input files use HTTP, Telnet, FTP, POP3 or SMTP otherwise use the port number. (see www.iana.org, look at port numbers for the correct values for these services.)

For example, output, see the output files (files with the extension *.out*) provided with this assignment in the compressed tar file.

Thoughts:

- How to start – Using wireshark open ArpTest.pcap and study wiresharks's output of the Ethernet and ARP headers. Your output is similar to wireshark.
- The .out files I provide you are the "correct" output. Your program must match these files. Even if they have typos they are "correct" and you must match them!
- For spacing many times I use "\t" or "\t\t". For example:

```
printf("\t\tProtocol: TCP\n");
```

- To move data between buffers, use memcpy(). Do NOT write your own byte copy code

.

- Use wireshark to view the .pcap files. This will allow you to look at the bytes in the packet and compare it with your output. This should help you debug you program.
- Test your program by diff'ing your output against my output. That is how I will test it.
- Declare your packet/frame buffer as an unsigned char array/pointer.
- Make sure you **ntohs() or ntohl()** when needed such as on port numbers.
- Note that the length field in the TCP pseudo header needs to be in network order
- The TCP header, specifically the checksum, takes the most time. You will receive zero (0) credit for the TCP header part of the program (part #4) without having the TCP checksum working correctly on all TCP packets.
- Note
    - The data offset field in the TCP header is sometimes called header length in documents on the web.
    - For display purposes the IP and TCP checksums should not be treated as a short. The bytes in the checksum should be treated as two separate bytes and printed out accordingly.