Wiretap Report

Ali Khalfan
Brennon York

October 15, 2010

Introduction

We have implemented a basic digital wiretapping device. All requirements that were stated within the problem description have been handled accordingly. Each section below briefly describes the algorithms we used to solve the problems and any issues that might have arisen. Additionally we mention any design decisions taken and why we did so when implementing the said protocol or algorithm.

Packet Structure

Packets captured in the .pcap format are held in a specified way. To handle this we needed to implement a design to easily allow us to extract the data we needed. Originally it was unknown that specified data structures for Ethernet, IP, TCP, and UDP were available so an initial implementation of our own gave us the groundwork to build up from. Once we had reasoned about the provided data structures we opted to implement those instead. From there it became quite simple to extract the data into the structures provided and manipulate it with our program.

The initial start runs the pcap_loop function to iterate over each packet. For all iterations a casting to the structure of our choice was used on the packet data to fill out structure. Since each packet traveling on a network is wrapped in the typical 'onion' fashion we needed to cast each structure after the last starting with the Ethernet and moving in. This was necessary because for each packet the size of the header was needed before we could successfully offset from the original pointer and cast the next structure. Once that was done we could traverse up the different headers extracting data from each layer until we reached the payload. The headers gave up insight (e.g. what the above layer protocol was) on how to parse the rest of the packet and call the appropriate function.
The pcap_loop function also carries another important piece of information. For each packet captured within a .pcap format network dump a format header is prepended to the front of it. The pcap_loop function gives us access to this structure as well which allows us to retrieve time stamps and payload sizes of entire packets. This information was easily obtained by accessing the individual structure elements passed.

Output

In order to fulfill the project's output requirements. A link list was build with a generic data structure that consisted of a string and an integer value. The string would represent the distinct values that were required as output (MAC address, IP address, Network Protocols, Transport protocols, and TCP and UDP ports). Since we were required to output the source and destination for some of the layers, there were 10 lists in total of this same data structure. A generic function was used to take care of all the 10 structure that would compare the new packet to existing values in the list and add it if non similar existed, or increase the count if the required value (e.g. MAC address, IP address...etc) existed in the list. The function was called whenever a specified packet was detected. For example, if an IP packet was detected and process the function would be called twice: once with the source IP link list and once with the destination IP link list as input, along with value of the IP address being processed.
A similar intuition was followed when inferring the flow of the packets. However, the structure of the items of the link list was different. The structure this time consisted of both source and destination IP addresses and ports, as well as the transaction type and a counter for both packets coming from the source (outgoing), or packets coming from the destination (outgoing).

The duration was calculated by subtracting using the *timersub* function with the timestamp of the first and last packet as input.

UDP Checksum

The checksum calculation was a defining point to this project. At first glance we had assumed that the checksum was computed merely over the header of the user datagram protocol. After searching RFC documents to understand why our calculations were coming out wrong we realized that we needed to calculate the checksum over the pseudo-header, UDP header, and the payload. The pseudo-header is made up from the source IP address, destination IP address, one zero byte, the UDP protocol (0x11), and the length of the UDP header plus payload. From there it became much easier to understand our calculation procedures.

To implement this algorithm we started by reading the class text book and realizing that a simple UDP checksum function was already given within. This function would take in a pointer to a buffer and a count that would tell the function the number of times to iterate over the buffer in 16 bit (half-word) increments. From there it would compute the 1's Compliment Addition over the entire buffer and return the 1's Compliment 16 bit result. After the analysis of the algorithm was over it merely became a process of getting the data into a single contiguous piece of memory.

To accomplish this task we began by calling calloc over a size of the UDP header (8 bytes), pseudo-header (12 bytes), and payload. Since the UDP length field, contained within the UDP header, was already storing the length of the UDP header and payload all that was needed was a pointer to the beginning of memory for the UDP header. For each item within the pseudo-header we ensured they were in network-byte order and then put them into the dynamically calloc'ed block of memory. Since the entire UDP header and payload were already in network-byte order all that was needed was a call to memcpy to put that chunk of data into our buffer. From there the buffer was filled and we could pass it along to the algorithm given in the book. Since we were verifying the checksum, and the algorithm in the book was for calculation, we needed to look for the value 0x0000 instead of 0xFFFF to verify if the checksum was indeed correct.

One issue with checksum calculations was if the payload were odd. Since 1's Compliment operations can have an infinite amount of 0's appended to the end and the same result we simply added a check if it were odd or not. If it was then we would allocate one extra byte and since we were using calloc instead of malloc all of our bytes were already zeroed out. This allowed us to be able to forget about needing to set the last byte in an odd-byte UDP packet.