Final Project Report

Project Description

Participants:

- Gabriel Vainer(1007121204)
- Howard Yang(1006722478)
- Ben Wilson(1007289024)

For our final project, we built a command line packet sniffer that implements critical functionality from Wireshark.

The rationale behind this is to learn and discover how common network monitoring software (such as Wireshark) works “under the hood”, and how implementing it would be beneficial for learning about Computer Networks

Specific goals and targets in the project:

- Displaying packet information
- Protocol Identification (TCP, UDP, ICMP)
- GUI + color-coded packets
- Packet Filtering Options (by protocol type)
- Packet Exporting

Contributions/Documentation

Ben's Contribution:

TCP Stream Following:
The user can select a TCP packet among the captured packets and press 't' to follow it's TCP stream. A list of all TCP packets sent between the two host will be displayed.

is_same_tcp_stream_forward(PacketInfo *p1, PacketInfo *p2):

PacketInfo *p1: Pointer to the selected packet's information.
PacketInfo *p2: Pointer to information of the packet we're checking is in the same stream as p1.
Returns 1 if p1 and p2 belong to the same TCP stream.
Returns 0 otherwise.

Determines if two TCP packets, p1 and p2, belong to the same TCP stream where p2 is a packet captured after p1. Compares the source and destination IPs and ports of both packets to verify a match. Ensures that the sequence number of p1 is less than or equal to that of p2 or the acknowledgment sequence number of p1 is less than or equal to p2.

is_same_tcp_stream_back(PacketInfo *p1, PacketInfo *p2):

PacketInfo *p1: Pointer to the selected packet's information.
PacketInfo *p2: Pointer to information of the packet we're checking is in the same stream as p1.
Returns 1 if p1 and p2 belong to the same TCP stream.
Returns 0 otherwise.

Determines if two TCP packets, p1 and p2, belong to the same TCP stream where p2 is a packet captured before p1. Compares the source and destination IPs and ports of both packets to verify a match. Ensures that the sequence number of p1 is less than or equal to that of p2 or the acknowledgment sequence number of p1 is less than or equal to p2.

tcp_trace(PacketInfo *packet, WINDOW *win, const struct timeval *start_time):

PacketInfo *packet: Pointer to the starting packet for the trace.
WINDOW *win: A pointer to the ncurses window used for displaying the trace.
const struct timeval *start_time: Pointer to the start time of the capture for calculating elapsed time.

Traces and displays the TCP stream associated with a given packet in an interactive user interface. Synchronizes and reconstructs TCP streams by identifying related packets based on sequence and acknowledgment numbers. Provides an interactive terminal-based UI for navigation and analysis of the TCP stream. Initializes a buffer stream_packets to store packets in the TCP stream. Traverses packets backward (from the current packet) using:

- **is_same_tcp_stream_back** to identify related packets in the reverse direction.
- Stops when encountering a SYN packet or unrelated packet.

Traverses packets forward (from the current packet) using:

- **is_same_tcp_stream_forward** to identify related packets in the forward direction.
- Stops when encountering a FIN packet or unrelated packet.

Importing and Exporting:

The user can save the currently captured packest to be viewed again later.

export_to_pcapng(const char *filename):

> **const char *filename**: The path and filename where the PCAP-NG data will be saved.

Exports the captured packets currently stored in packet_list to a specified PCAP-NG file. Creates/opens the specified PCAP-NG file and writes all packets from the packet_list to it. Each packet is written with a timestamp and its full captured length. If the pcap handle or PCAP-NG file cannot be opened, an error message is displayed.

import_from_pcapng(const char *filename):

> **const char ***filename: The path and filename of the PCAP-NG file to be read and imported.

Imports packets from a given PCAP-NG file into the packet_list Opens the specified PCAP-NG file for reading. Iterates through all packets, extracting the Ethernet and IP headers. Identifies protocols (ICMP, IGMP, TCP, UDP, or OTHER). For TCP packets, extracts sequence, acknowledgment numbers, ports, and flags. Converts IP addresses to human-readable format. Packs all information into PacketInfo structures and stores them in packet_list. Continues until all packets are read, or the maximum storage limit is reached.

Howard's Contribution:

Displaying packets:

When the user presses the right arrow while selecting a packet, it will display more information in the right box. Pressing the up and down arrow keys allows scrolling.

void display_packet(WINDOW *win, PacketInfo *info)

WINDOW *win: The window in which the packet data will be displayed
PacketInfo info: The packet to be displayed

This function determines what headers are important to display by examining the protocol field of the info struct. The remainder is a while loop that displays the current packet headers, and responds to directional arrow keys to enable. To exit the while loop simply press the left arrow key.

Filtering Packets:

Pressing "F" while on the main packet list, moves the cursor to the top input text box where you are able to type specific keywords that will filter the packet list.

typedef struct FilterItem_t {
enum FilterField field; // either source ip, destination ip or protocol
int negate; // whether or not to negate
char value[MAX_SEARCH_VALUE_LEN]; // the string to filter by
} **FilterItem**;

typedef struct Filters_t {
int list_size; // the current number of filters
FilterItem filters[100]; // the list of filters
char filter_string[MAX_FILTER_LEN];
int filter_pos;
} Filters;

int match_filters(Filters *filters, PacketInfo *info)

Filters *filters: The list of filters
PacketInfo *info: The packet to check against the filters
returns True iff the packet matches all the filters

This function is a helper for determining whether or not a packet is matching the currently applied filters

void type_filter_box(WINDOW *win)

WINDOW *win: The window that will be typed

This is the function responsible for reading keyboard inputs and saving the contents of the filter string to a buffer. There is a loop that parses this string and looks for keywords such as “src=”, “dst=”, and “proto=” and the subsequent value to filter by. It also checks if the first character is a “!” and negates the following expression, so “!src=127.0.0.1” is a valid filter.

Gabe's Contribution:

Struct PacketInfo

PacketInfo is a structure that encapsulates all relevant information about a captured network packet. It includes:

- **Timestamp**: When the packet was captured.
- **Packet Number:** A unique identifier for each packet.
- **Protocol:** The type of protocol used (e.g., TCP, UDP).
- **Buffer (buf):** Pointer to the raw packet data.
- **Source and Destination IPs:** Human-readable IP addresses.
- **TCP-Specific Fields:** Sequence number, acknowledgment number, source and destination ports, and TCP flags.
- **Size:** The size of the packet in bytes.

```
PacketInfo *packet_list[MAX_PACKETS+1];
PacketInfo *filtered_packet_list[MAX_PACKETS+1];
int packet_count = 0;
int filtered_packet_count = 0;
int cursor_position = 0;  // index of the currently selected packet
int paused = 0;
int imported = 0;
FILE *log;
```

Keeps the current state of the program including a list of the processed packets, a list of filtered packets, the current packet count of the program, as well as indicators for the cursor and pause/resume state.

int main(int argc, char argv)**

Sets up a raw socket for packet capture, initializes the ncurses user interface draws the necessary components, and handles user inputs for interacting with packets. It continuously listens for packets using recvfrom(), processes them, and updates the UI for users to view or manipulate packet details, as well as determines whether we are paused (and should stop listening for packets) or to continuously listen.

process_packet(WINDOW *win, unsigned char *buffer, int size, int packet_no, const struct timeval *start_time)

WINDOW *win: The window the packet list will be outputted to.

unsigned char *buffer: The packet’s data
int size: The packet’s size
int packet_no: The packet’s number in the packet_list
const struct timeval *start_time: The Start time for the capture

Processes a captured packet by extracting its headers and creating a PacketInfo structure containing the metadata. It adds the packet to the list of packets, pplies filters if any, and updates the packet list displayed in the ncurses window.

print_packets(WINDOW *win, const struct timeval *start_time)

WINDOW *win: The window the packet list will be outputted to.
const struct timeval *start_time: The Start time for the capture

Prints the respective summary list of captured packets in the dedicated ncurses window. It supports scrolling and highlights the currently selected packet, displaying details such as the packet number, protocol, source/destination addresses, and length.

get_elapsed_time(const struct timeval *start_time, const struct timeval *current_time)

const struct timeval *start_time: The Start time for the capture .
const struct timeval *current_time: The current time.

Calculates the elapsed time in seconds between the start time and the current packet's timestamp, used for tracking relative packet arrival times.

How to run

bash ./compile.sh && sudo ./sniffer

Key binds:

- e: Export packet list to a specified file.
- i: Import packet list from a specified file.
- t: Trace the TCP stream of the currently highlighted TCP packet
- Ctrl-r: Pause current packet capture.
- Right-Arrow: Display currently highlighted packet information.
- f: filter packet list based on specified arguments.
- q: Exit program.

Analysis and Discussion

Performance:

- Our sniffer updates the packet_list and displays in real-time Real-time with minimal latency.
- Our sniffer handles up to 100,000 packets without significant loss in performance

Challenges:

- Due to how timestamps are stored in pcap we were unable to recalculate the timestamps for imported packets resulting in negative time stamps.
- Implementing accurate TCP stream tracing presented challenges due to the need to correctly match sequence and acknowledgment numbers,

Conclusion

This project provided an in-depth understanding of network protocols, packet structures, and low-level programming with raw sockets. This packet sniffer is a foundational tool that highlights the practical aspects of computer networks and their real-world applications.