CS241

Shayan Borhani Yazdi

December 2023

1 Introduction

This report is set to outline the methodology and the design choices made for the CS241 coursework - skeleton package extension for packet sniffing.
We will discuss different parts of the project in the same order as the objectives were mentioned in the specification.

2 PCAP_LOOP

The first task to be done was to modify the sniffing loop to utilize the pcap_loop() function instead of pcap_next() inside a while(1) loop which is far more efficient. The pcap_loop function requires a utility function to be applied to each packet received, which is why we have packet_handler_function, which calls dump (if required) and dispatch.
The consequences: The program is set to finish after an issue of cntrl+c or SIGINT, which is why we need to handle this signal in our code as well. However we need to close our pcap_handle as well, before terminating our run. As a result, we break the loop after getting a SIGINT, to then close the pcap_loop before the end of program to prevent potential loss of resources or eventual bugs.

```c
void handler(int sig){
  if (pcap_handle != NULL){
    pcap_breakloop(pcap_handle);
  }
}

// setting up signal handler
struct sigaction sa;
sa.sa_handler = handler;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
if (sigaction(SIGINT, &sa, NULL) == -1){
    perror("sigaction");
    exit(EXIT_FAILURE);
}

```

```c
pcap_loop(pcap_handle, 0, packet_handler_function, (unsigned char
    *)&verbose);
pcap_close(pcap_handle);
```

Listing 1: pcap_loop

3 Packet Analysis

In this section we will discuss the analysis of packets (used in both analyse function and dump) and how different attacks are detected.
The first part of the decoding is already done, which corresponds to packet header and packet data. In order to access Packet IP data, we need to move ETH_LEN forward through our packet:

```c
struct ip *ip_header = (struct ip *)(data + ETH_HLEN);
```

Listing 2: IP header

We can then decode the TCP header (if the ip protocol is TCP) the same way, by moving ETH_LEN and then ip_header→ip_hl * 4 (or * << 2), because the ip_hl specifies the length in bytes.

```c
if (ethernet_type == ETHERTYPE_IP && ip_header->ip_p == IPPROTO_TCP
    )
    struct tcphdr *tcp_header = (struct tcphdr *)(data + ETH_HLEN +
     ip_header->ip_hl * 4);
```

Listing 3: TCP header

3.1 analysis.c

In order to make the analysis process more robust and more efficient, we decided to introduce a new data structure analysisResponse and then change the prototype of our analyse() function to this type.

```c
struct analysisResponse {
    int isSynAttack; // 0: no, 1: yes
    char ip[16]; // only considering IPv4
    int isBlackListedURL; // 0: no, 1: yes GOOGLE 2: yes BBC
    int isARPresponse; // 0: no, 1: yes
};


struct analysisResponse * analyse(const struct pcap_pkthdr *header,
              const unsigned char *packet,
              int verbose);
```

Listing 4: analysisResponse

Now every time dispatch calls analyse, it receives a new analysisResponse containing the detection of each malicious attack, using the methods listed below:

```c
// packet structure to look for for syn attacks ehternet + ip + tcp
struct ether_header *eth_header = (struct ether_header *)packet;
if (ntohs(eth_header->ether_type) == ETHERTYPE_IP) {

  const unsigned char *ip_packet = packet + sizeof(struct ether_header);
  struct ip *ip_header = (struct ip *)ip_packet;

  if (ip_header->ip_p == IPPROTO_TCP) {

    const unsigned char *tcp_packet = ip_packet + (ip_header->ip_hl << 2);
    struct tcphdr *tcp_header = (struct tcphdr *)tcp_packet;

    // now checking for syn
    if (tcp_header->syn && !tcp_header->ack){
      response->isSynAttack = 1;

      // getting the IP address of the source
      const char *sourceIP = inet_ntoa(ip_header->ip_src);
      strcpy(response->ip, sourceIP);

    }

    if (ntohs(tcp_header->th_dport) == HTTP_PORT){
      const unsigned char *http_payload = tcp_packet + (tcp_header->th_off << 2);
      int isBlacklisted = isURLBlacklisted((const char *)http_payload);
      if (isBlacklisted){
        response->isBlackListedURL = isBlacklisted;
        const char *sourceIP = inet_ntoa(ip_header->ip_src);
        const char *destIP = inet_ntoa(ip_header->ip_dst);
        printf("==============================\n");
        printf("Blacklisted URL violation detected\n");
        printf("Source IP address: %s\n", sourceIP);
        char *site = isBlacklisted == 1 ? "google" : "bbc";
        printf("Destination IP address: %s (%s)\n", destIP, site);
        printf("==============================\n");

      }

    }
```

Listing 5: Detection process

4 Multithreading

4.1 Threadpool over One Thread Per X Model

For our choice of multithreading strategy we decided to implement a threadpool model, because we believe it to be more efficient than the other choice and more stable. As we only have to build the threads once in the beginning and destroy

them only once, the resource management is robust and prevents potential leaks more effectively.
As the program is run on a VM, we let the MAX_THREAD number to be 2, however this number can be changed at any time.

4.2 Code explanation

Multithreading is implemented mostly in sniff.c and dispatch.c. In sniff.c, we initialize our workQueue and then initialize the threadpool:

```
// Create the work queue
workQueue = create_queue();

// initialize our threadpool
initializeThreadpool();
```

Listing 6: Initializing multithreading

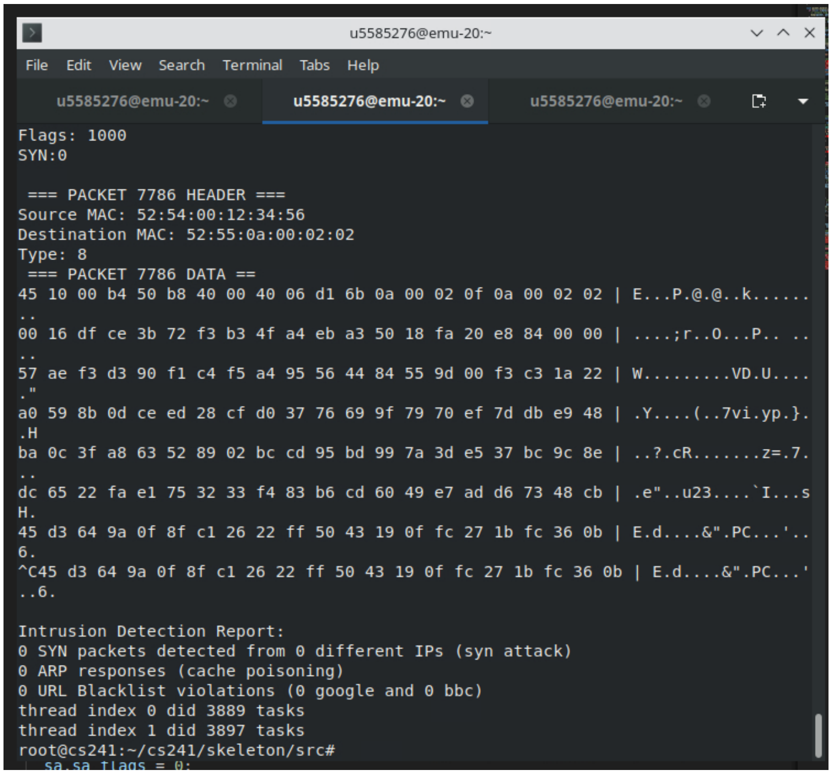
Then in dispatch.c, we add every packet we receive to the workQueue, allowing them to picked up by our workerThreads.

```
void dispatch(const struct pcap_pkthdr *header,
              const unsigned char *packet,
              int verbose) {

  // setting up signal handler
  signal(SIGINT, finalReport);

  addToQueue(header, packet, verbose);
}
```

Listing 7: filling the workQueue

4.3 Testing multithreading functionality

In order to test the multithreading functionality, we decided to keep track of the number of tasks each threads gets assigned to and print it out at the end of our report to see if there is a relative balance. This printing functionality is commented out in the final submission to respect the requested output format, however can be revoked again at any time. Below is a screenshot with an example run of the program:



To sum up, dispatch.c contains global variables related to each type of intrusion, and once the SIGINT is received, the finalReport function prints the report in the correct format, frees the resources in use and finishes the program. To test the detection process for each type, the 3 methods introduced in the specification have been used and resulted in success.

5 Final Notes

The code compiles and run with no errors or warnings. However, as it can be found on libpcap's github here, there have been reports of issues with libpcap and memory management. Our current multithreaded program perform the tasks correctly and finishes without error upon ending, however when used with valgrind, we always have memory leaks, according to OpenAI's ChatGPT (OpenAI, n.d.), caused by libpcap inner functions, mainly upon calling of pcap_compile.
This is the reason why we have included two different versions, **skeleton.zip** containing the multithreaded version with memory leaks, and **skeleton_single_threaded.zip** which is the version without multithreading.
ChatGPT was used through the final steps to analyse long valgrind reports, and to verify the source of memory leaks.

6 References

Mukhopadhyay, A. (n.d.) *Network Primer.* https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/coursework23-24/network-primer/

pcap_open_live uses uninitialized data https://github.com/the-tcpdump-group/libpcap/issues/450 [Aug 3, 2015], visited Dec 8, 2023

Die.net website for accessing linux man pages online https://linux.die.net/man/

ChatGPT 3.5 website chat.openai.com