REPORT

CS2410

u1719840

Atom Deutsch-Filippi

Introduction

In this report we will explain the design, implementation and testing of a packet sniffer program which analyses the packets it receives and takes notes of all potentially malicious packets. In order to keep up with large amounts of packets, the program uses a thread pool model.

I. DESIGN

Designing this software was made easier thanks to the detailed specification[1] provided. The project was split into three parts defined in the specification. The three parts are parsing packets, analysing said packets and improving the dispatching method using multithreading.

Firstly, we will discuss the Part 1 in the specification. We were asked to parse packet headers, more specifically Ethernet, IP or ARP and TCP headers with a bit of payload formating. What is interesting with these packets is that some of these had fixed sizes such as the Ethernet header while others had variable sizes which could change with each packets such as the IP header and TCP headers because of optional information included in these packets. And so you have to calculate the size of each header for each incoming packet. To get access to the information stored in these packets we format them by casting structs to the raw data (see Implementation for details) and shifting the pointer a certain number of bytes (the length of the headers before the header we want to analyse) to get the next header. In the ethernet header we get access to the type of protocol the layer underneath it uses i.e IP or ARP and we also get the details of the MAC addresses of the source and destination of the packet. Next is either the IP or ARP header depending on the type of packet. For the IP packet, we get the IP address of the source and destination of the packet. After the IP header there is the TCP header which gives us access to the TCP port used. If the packet isn t an TCP/IP packet then it could be an ARP packet. The fact that it is an ARP packet gives us the information we need. Finally, we can display everything that can be converted to an ASCII characters which will unable us in the case of an HTTP payload to find the host website.

Then, in Part 2 of the specification, our task was to analyse packets thanks to the parsing done in Part 1. There were three types of malicious attacks we had to consider: Cache poisoning, Xmas packets and Url blacklist violations. To spot the first attack, we had to report all of the ARP packet replies which flowed through the system. This was done by counting the number of ARP replies in the packets we analysed. For the next attack, we checked whether

[1] https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/coursework18-19/

the flags FIN, URG and PSH were set and counted the packets who did. For the last attack, we took a look inside of the payload after all the headers to check for the destination of the http request and checked if it was blacklisted. When the program is forced to stop, we needed to display the number of attacks. To do so, we used signal processing to catch any CTRL + C signal (SIGINT) in order to force our program to stop and display the report.

In the last part of the specification, we needed to use multithreading to speed up the pace at which we analyse packets. Using a threadpool model was the better choice as it resisted better in bursts of request. A threadpool is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application[2] and it consists of a number of threads which take packets from a queue and wait when nothing is in the queue.

[2] https://docs.microsoft.com/en-us/windows/desktop/procthread/thread-pools

II. IMPLEMENTATION

We will approach the implementation section by studying the files we needed to modify: sniff.c, dispatch.c and analyse.c. The main.c file was off limit as we weren't allowed to edit it.

In sniff.c there was little change, the first change is changing the packet fetching loop with pcap's own loop[3] (Code 1.1). It takes a the handle created in the given code, a number of packets to send (0 means infinite), a function (Code 1.2) to send the packet to and a u_char pointer which can point to some data we optionally could provide. The function pre_dispatch is useful as it serves as a passing point to make the dispatch take the arguments we want it to take and not the ones imposed by pcap_loop. The second change is the free_pcap_handle function which uses a global defined handle to free it easily.

In dispatch.c is where the bulk of the multithreading code is. To implement the queue we need a node struct to contain all the information we need (Code 1.3). Then on the first call of dispatch we create THREAD number of worker threads. And finally enqueue all of the received packets. The enqueuing is straight forward however specifying the multithreaded part is of interest. We use a mutex lock (pthread_mutex_t) as well as a condition lock. The regular lock is used when threads access global variables and the condition lock is used to stop threads from trying to dequeue when the queue is empty thanks to pthread_cond_wait(). We then make sure to prevent memory leaks when taking the next packet and to mutex lock the threads. The packets are then sent to analysis.c to be analysed. The final part of dispatch.c is the signal catching (which should be in main.c but it is prohibited to edit main.c, this causes a bug where another packet needs to be sent after the signal in order to terminate the program), the signal handler just sets a global variable to 1 in order to signal to dispatch to end and call cleaning methods and print the report (Code 1.4).

In analysis.c we analyse the packets received. We firstly parse the packets with these structs making sure to shift the pointer after each header to get a struct for each layer heade (Code 1.5-1.8) we seperate TCP/IP and ARP packets using a switch statement. The second part of this file is to detect what

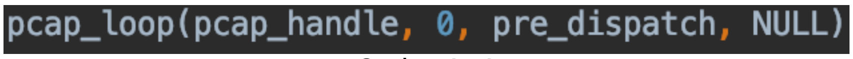
[3] https://linux.die.net/man/3/pcap_loop

we deemed to be potentially malicious files. The first test is the Xmas packet (so called since its flags are all set to true, it is all lit up i.e a Christmas packet) where we just check if the flags FIN, URG and PSH (Code 1.9) are all set to 1. Next is the cache poisoning attack, for this attack the specification asked to report all ARP replies (Code 1.10). Finally the last test is a blacklist violation, for which we check the tcp port and host website (Code1.11).

III. TESTING

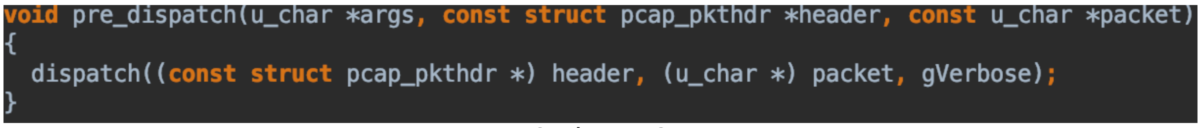
Testing the parsing section was straight forward, justing printing the headers using verbose could unable us to see whether or not the header s made sense. Thanks to the url blacklist of www.bbc.co.uk we could find what IP our parser would give and compare it to the official IP addresses of BBC. When analysing, printing the details of the packet when we detect one shows us when the detection works or not. To debug the multithreaded part of the coursework, I tried debugging it while only using one thread and using prints when I had an error to understand it and fix it. Being able to only send one packet at a time was very helpful (using the -i lo suffix when calling ./idsniff and sending one ARP reply). Finally cleaning up the program after execution was made harder by forbidding us to edit main.c to close easily the program whenever, adding signal handling to dispatch.c means that for the program to close properly it needs to get a packet after CTRL + C.

APENDIX



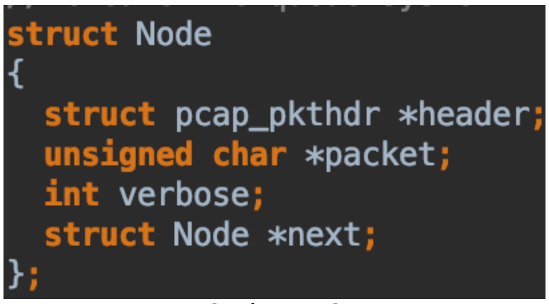
```
pcap_loop(pcap_handle, 0, pre_dispatch, NULL)
```

Code 1.1



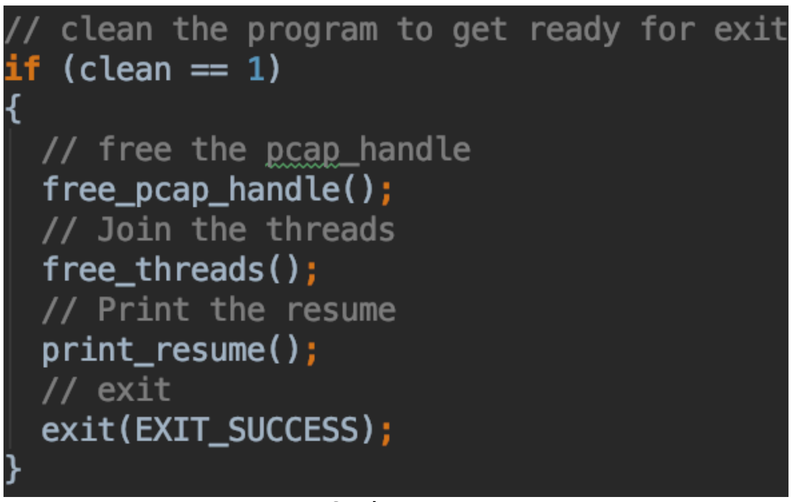
```
void pre_dispatch(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
  dispatch((const struct pcap_pkthdr *) header, (u_char *) packet, gVerbose);
}
```

Code 1.2



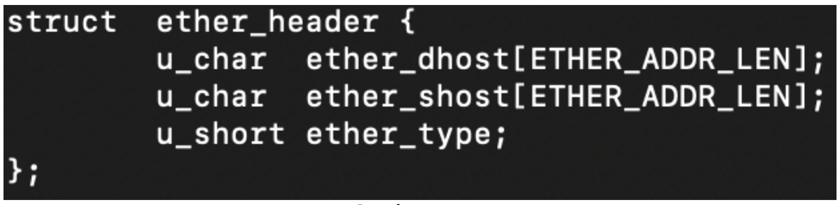
```
struct Node
{
  struct pcap_pkthdr *header;
  unsigned char *packet;
  int verbose;
  struct Node *next;
};
```

Code 1.3



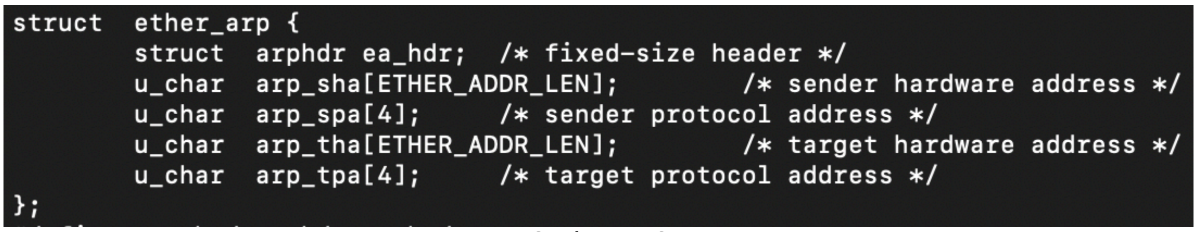
```
// clean the program to get ready for exit
if (clean == 1)
{
  // free the pcap_handle
  free_pcap_handle();
  // Join the threads
  free_threads();
  // Print the resume
  print_resume();
  // exit
  exit(EXIT_SUCCESS);
}
```

Code 1.4



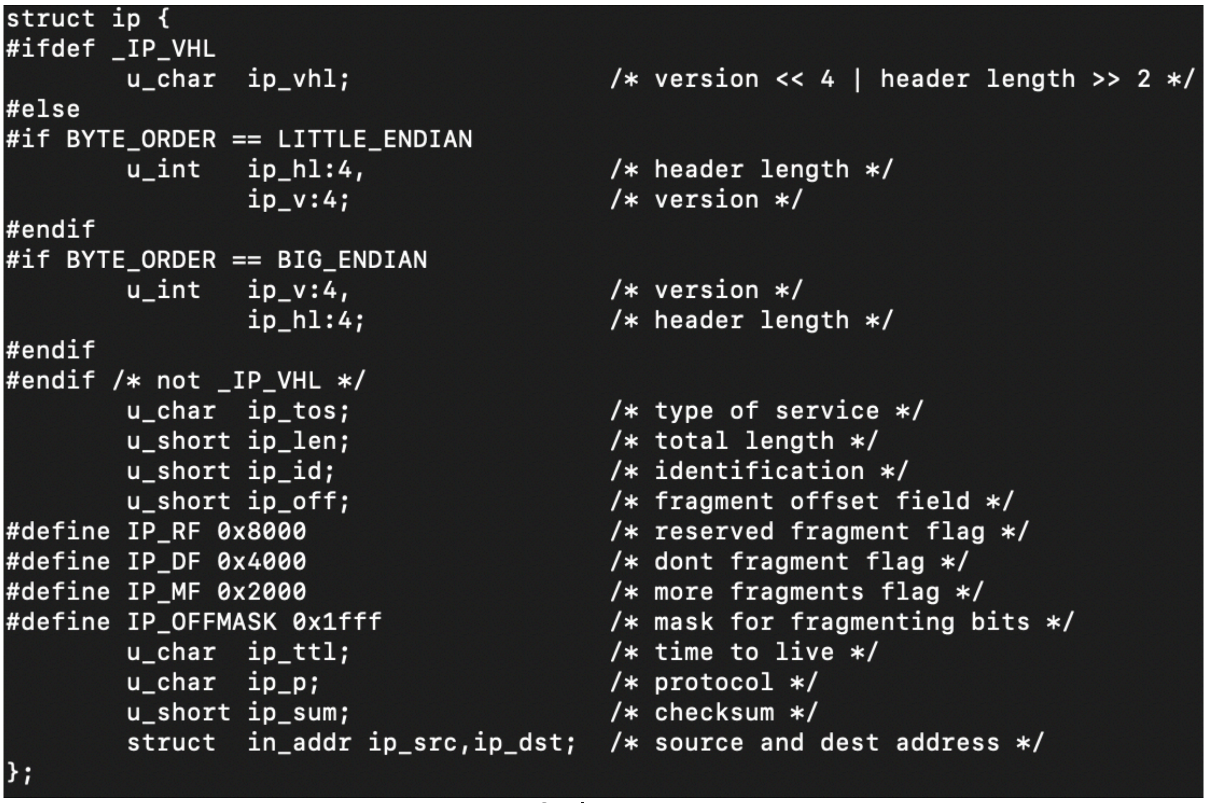
```
struct  ether_header {
        u_char  ether_dhost[ETHER_ADDR_LEN];
        u_char  ether_shost[ETHER_ADDR_LEN];
        u_short ether_type;
};
```

Code 1.5



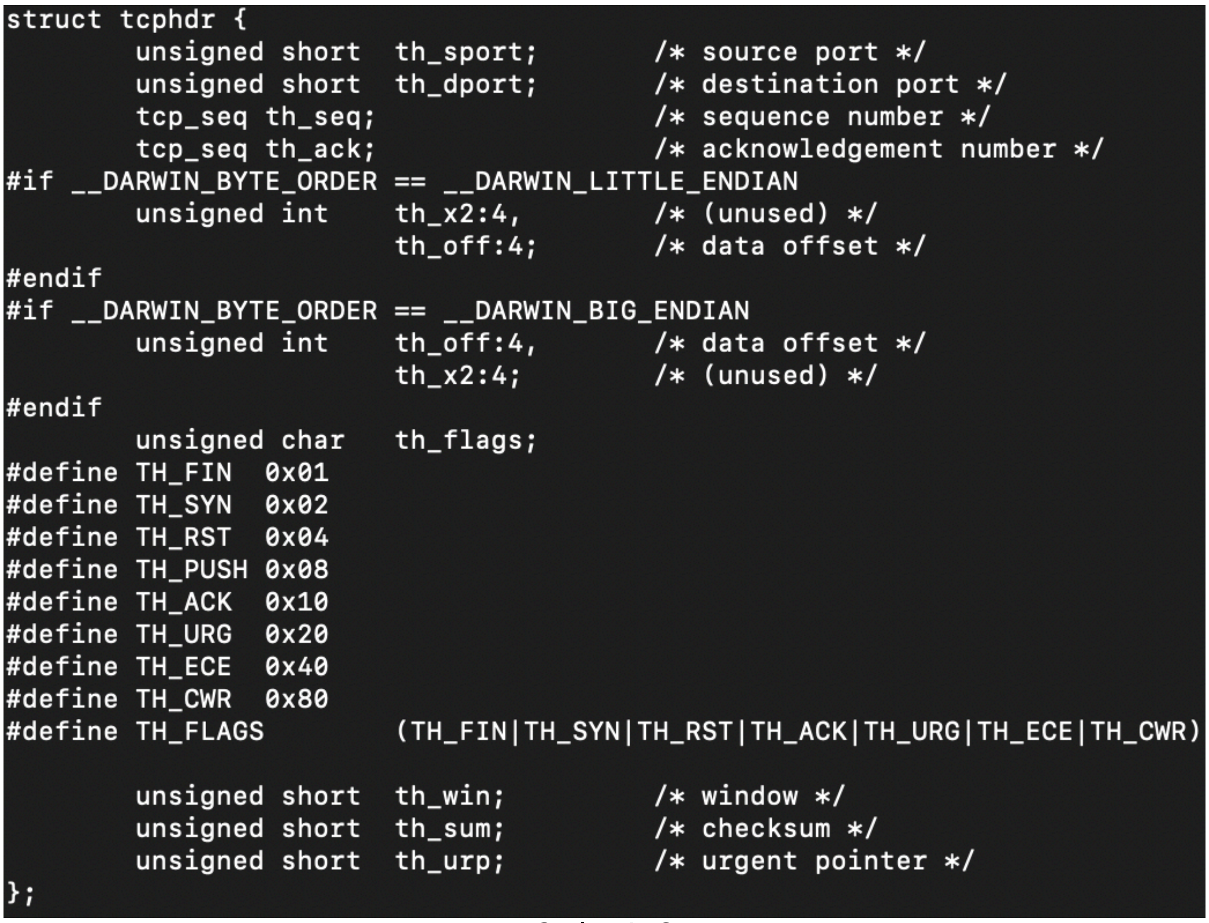
```
struct  ether_arp {
        struct  arphdr ea_hdr;  /* fixed-size header */
        u_char  arp_sha[ETHER_ADDR_LEN];        /* sender hardware address */
        u_char  arp_spa[4];     /* sender protocol address */
        u_char  arp_tha[ETHER_ADDR_LEN];        /* target hardware address */
        u_char  arp_tpa[4];     /* target protocol address */
};
```

Code 1.6



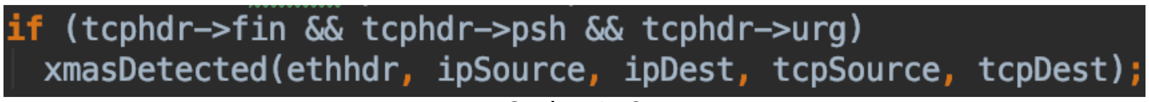
```
struct ip {
#ifdef _IP_VHL
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
#else
#if BYTE_ORDER == LITTLE_ENDIAN
        u_int   ip_hl:4,                /* header length */
                ip_v:4;                 /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
        u_int   ip_v:4,                 /* version */
                ip_hl:4;                /* header length */
#endif
#endif /* not _IP_VHL */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
#define IP_RF 0x8000                    /* reserved fragment flag */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
```

Code 1.7



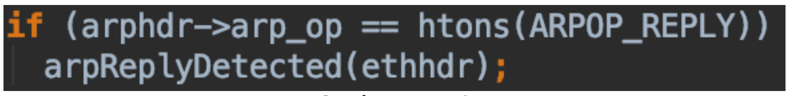
```
struct tcphdr {
        unsigned short  th_sport;       /* source port */
        unsigned short  th_dport;       /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
#if __DARWIN_BYTE_ORDER == __DARWIN_LITTLE_ENDIAN
        unsigned int    th_x2:4,        /* (unused) */
                        th_off:4;       /* data offset */
#endif
#if __DARWIN_BYTE_ORDER == __DARWIN_BIG_ENDIAN
        unsigned int    th_off:4,       /* data offset */
                        th_x2:4;        /* (unused) */
#endif
        unsigned char   th_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
#define TH_ECE  0x40
#define TH_CWR  0x80
#define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)

        unsigned short  th_win;         /* window */
        unsigned short  th_sum;         /* checksum */
        unsigned short  th_urp;         /* urgent pointer */
};
```

Code 1.8



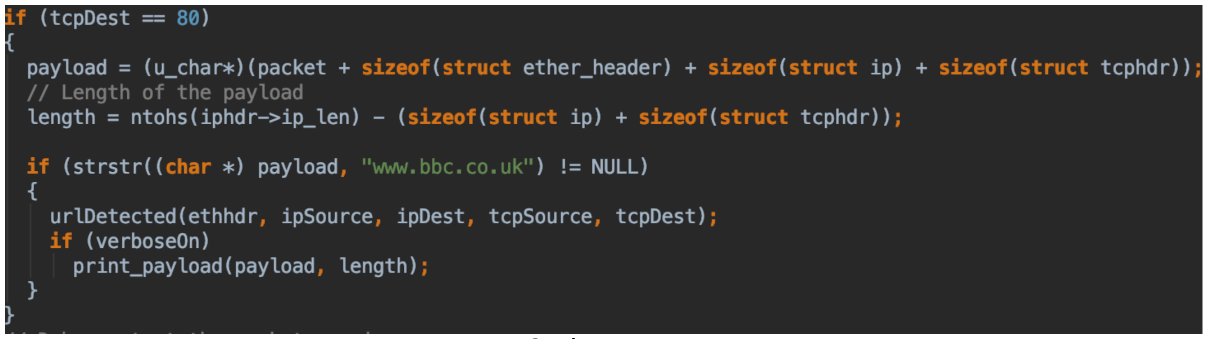
```
if (tcphdr->fin && tcphdr->psh && tcphdr->urg)
  xmasDetected(ethhdr, ipSource, ipDest, tcpSource, tcpDest);
```

Code 1.9



```
if (arphdr->arp_op == htons(ARPOP_REPLY))
  arpReplyDetected(ethhdr);
```

Code 1.10



```
if (tcpDest == 80)
{
  payload = (u_char*)(packet + sizeof(struct ether_header) + sizeof(struct ip) + sizeof(struct tcphdr));
  // Length of the payload
  length = ntohs(iphdr->ip_len) - (sizeof(struct ip) + sizeof(struct tcphdr));

  if (strstr((char *) payload, "www.bbc.co.uk") != NULL)
  {
    urlDetected(ethhdr, ipSource, ipDest, tcpSource, tcpDest);
    if (verboseOn)
      print_payload(payload, length);
  }
}
```

Code 1.11