CS241 Coursework : Basic Intrusion Detection System - Report

Joshua Lawson u2058262

1. **Capturing Packets**
   We captured packets using pcap_loop() as it is more efficient than pcap_next(). pcap_loop() used a callback function (process()) to determine how to process the packet, and it ran indefinitely since we supplied it with a negative number[1]
2. **Analysis**
   Analysis began with parsing the headers: the ethernet,IP header and TCP headers, with the payload after the TCP header. To parse the ethernet header, since it is statically 14 bytes long, all we did was increment the packet pointer by 14 bytes. To parse the IP and TCP headers we got the size of the headers found in ip->ip_hl and tcphdr->doff. Since they were stored as 4 byte words, we had to multiply those values by 4 to get the real header lengths.
   2.1 SYN Attack
   To check if a packet is a SYN packet, all we did was look at the control bit fields in the TCP header, named SYN, ACK, etc. If SYN was set to 1 and all the other control bits were set to 0, then that meant the packet must be a SYN packet.
   However, we also had to find the IP address of each SYN packet. This meant that we had to store the IP address of each SYN packet in an array (found in the IP header of the packet) and when the program terminated, would work out how many of those SYN packets were from unique IP addresses. The IP addresses had to be processed with inet_ntoa() so that the IP could be read easily as a 4 byte string. We implemented the array as a dynamically growing array for robustness.
   2.2 Dynamic Array
   The dynamic array was implemented as a char**. The dynamic array initialized with an initial size (100) and has an error check so that a dynamic array cannot be initialized twice. It then malloced the array of strings and set the (current total) size to 100 and the elements used to 0. Whenever we wanted to insert an element to the array, it would first check if the total size equaled the elements used. If not, just add the element to the array. If it did, the array was full, so we increased the total size of the array (squared so 100000) and then realloced that amount of memory. However, the return value of realloc must be checked, as if realloc fails then the original pointer to the data could be lost, so we first assigned the return value of realloc to a temporary pointer, and if realloc succeeded then assign it to the array pointer.
   2.3 ARP Cache Poisoning
   To find if a packet was an ARP response, we used the ether_arp structure and checked the value of the arp_op data field, and if that contained 2 then the packet was an ARP response.[2]

2.4 URL Blacklisting

The program first worked out if the destination port of the packet was 80 (a HTTP related packet. Then the program had to find the URL of the HTTP request packet, stored in the payload of the packet which is found after the TCP header.[3]

Then once the URL was found, compare the URL with any of the blacklisted URLs, and if a match was found displayed to the console a violation was found and the source and destination IP addresses of the offending packet in real time.

3. **Signal Handling**

The program only displayed this information (other than URL blacklist violations) when the program terminated. Since the program ran by capturing packets indefinitely, we had to catch the termination signal (Ctrl^C), process, output the required information, then end. This was used with a signal handler, INTHandler(). catching SIGINT in sniff.c[4]. Since printf() is unsafe in signal handlers, we used function calls then exited. The program had to parse the dynamic array to find the number of unique IP addresses. The program used a nested for loop that checked each IP against every other IP in the array, running in O(n^2), which found the number of unique IP addresses.

4. **Multithreading**

We went with the threadpool approach since we believed the added complexity would not be too bad and had already implemented a multithreaded threadpool server in lab 3. This meant we could re-implement the queue in lab 3 but change the data from INT to a structure that contained all the packet information. The implementation was essentially taken from the lab 3 solution with necessary adjustments made. First sniff.c created 3 threads which would wait while the queue was empty. Then, the packet is sent to an intermediary dispatch() function which enqueued the packet and told the threads work is ready. We used mutex locks to avoid race conditions since the threads ran in parallel so only one thread can dequeue at a time.

We also had to update analyze() so that none of the global variables such as the dynamic array would have values overwritten. This was achieved with mutex locks.

We also had to send the SIGINT signal to all the threads so they would close before the program begun processing and outputting information.

5. **Testing**

We mainly tested using printf() statements to check the values of certain variables. In darray.c we declared a printArray() function that would print all the IP addresses in the array, which was useful when testing if checkIfSYN() was correct. We also used valgrind to check if there were any memory leaks.

It was hard debugging the multithreaded version, so we added a packet number variable to each packet sent to analyze(). This was to check that every packet sent by pcap_loop() was actually received (and processed) by analyze().

6. **References**

[1]https://www.tcpdump.org/pcap.html

[2]http://www.propox.com/download/edunet_doc/all/html/structether__arp.html

[3]https://stackoverflow.com/questions/2703238/how-to-hijack-all-local-http-request-and-extract-the-url-using-c

[4]https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/