Intrusion Detection System Coursework Report

This report will examine the design choices, implementation and success of the supplied code in terms analysing packets and the utilisation of multi-threading. This will explore the capturing of packets and the multi-threaded analysis of the packets and their headers. The code will be evaluated according to tests that focused on correctness, efficiency, and memory safety and finally any drawbacks of the code, testing and evaluation process will be highlighted. This report assumes that the reader is comfortable with the “tcp_serv_threadpool.c”(6) source file. Since the implementation of multi-threading is very similar to it, minor details of that implementation will be omitted and only the major parts and the differences will be mentioned.

The entire process of capturing packets was done via the functions named “pcap_loop”(1.2), “dispatch”(1.3) and “analysis”(1.4). “pcap_loop” assigned “dispatch” as the callback function for when a packet was captured. “dispatch” added packets to a packet queue that was used in multi-threading. “analysis” was given the packet and corresponding metadata, with which it stripped back the link, network and transport header and according to them it updated global trackers(declared in 1.1) for SYN flooding, ARP poisoning and blacklisted URL requests. Note that when pcap_loop was invoked it completely occupied the thread that called it, in this case that was the main thread.

The multi-threading was implemented using the threadpool model with help from (6). All that was created around the threadpool model was the queue mutex lock, the queue condition variable and mutex locks for SYN flooding, ARP poisoning and blacklisted URLs. This design choice was so that threads didn’t have to wait for a tracker they didn’t need. For example, if a thread was processing an ARP packet, it wouldn’t wait for a thread that was processing a TCP packet with a SYN flag. Threads took work from the packet queue and were put to sleep if there was no work. That was done with the use of “pthread_cond_wait” in “threadAnalyse” which also doubled as a wrapper around “analyse” so that it could have been used by threads. Moreover an extra thread was created to monitor the “loop” variable(defined in 1.1 and performing the same role as in (6)). It was used to wake up sleeping threads and invoke “pcap_breakloop” to stop the main thread capturing packets, when the “SIGINT” signal was passed. Before the threads are joined back into main they have to finish processing all packets in the packet queue.

Other implementation details worth mentioning are the data structures, the SIGINT signal handler and manipulating “char” and “unsigned char” pointers. The SIGINT handler has the function “numToStrSafe”(developed using (4)) to convert the analysis trackers signal safely and print out a report using “write”. The function “inet_ntoa”(The parameter was figured out with help from (3)) is used to create an IP string from an integer. The output has to be copied into a dynamically allocated pointer because “inet_ntoa” returns a pointer to an internal buffer that was overwritten but further uses of it. Finally, the linked list and and queue dynamically allocated the items inside the nodes in order for them to be accessible, regardless if the passed data still existed.

The testing was done predominantly by having checked the final report, produced by the detection system, was aligned with expected result and having used Valgrind. Conducted tests are available in (2) and (3) and they show that the solution produced correct results for detecting the required intrusion vectors and that the code is memory safe. A note about the results any use of “x”, means either the value is variable or it isn’t important. In all cases the outputted reports matched the expected values and that implies correctly implemented synchronisation since no race conditions have falsified the totals. A notable point is directed requests to URLs massively increased the number of memory allocations but it isn’t completely clear why, maybe it is a result of using “wget”. A shortcoming of this testing has been not testing the efficiency gained as a result of multi-threading. That is because the testing required invoking commands and the SIGINT signal which heavily interrupted using a timer approach. However, it can be estimated that the potential speed up

was quite large, as once everything was initiated, the entirety of the systems computation is concurrent. We use 10 threads, therefore the potential speed up, according to Andahl's law and assuming complete use of said threads, is up to 10 times. This can be altered and it really depends on your machine's logical cores and the traffic on the used interface. During a test a problem was noticed with the use of the verbose flag. If the program uses the "eth0" interface and the verbose flag the final report get lets lost amongst the wall of text of printing the packets.

In conclusion, the network intrusion detection system and compatible multi-threading were implemented very successfully. All requirements of the specification have been fulfilled and the multi-threading offered an increase in efficiency even if it is not properly measured. A possible improvement is a better way to handle unique IPs, because a linked list can be quite slow once a very high number of unique IPs is stored.

References

1.The code – provided separately

- 1.1 – main.c
- 1.2 – sniff.c & sniff.h
- 1.3 – dispatch.c & dispatch.h
- 1.4 – analysis.c & analysis.h
- 1.5 – linkedlist.c & linkedlist.h
- 1.6 – queue.c & queue.h

2. Correctness Test Table

Test Name	Test ID	Interface Used	Expected tracker Values	Total Tracker Values
Loopback w/ no commands	1	"lo"	{0,0,0,0}	{0,0,0,0}
Loopback w/ 1 ARP commands	2	"lo"	{0,0,1,0}	{0,0,1,0}
Loopback w/ 10 ARP commands	3	"lo"	{0,0,10,0}	{0,0,10,0}
Loopback w/ 1 hping packet	4	"lo"	{1,1,0,0}	{1,1,0,0}
Loopback w/ 10 hping packets	5	"lo"	{10,10,0,0}	{10,10,0,0}
Loopback w/ 10000 hping packets	6	"lo"	{10000,10000,0,0}	{10000,10000,0,0}
Loopback w/ 10000 hping packets and 10 ARP commands	7	"lo"	{10000,10000,10,0}	{10000,10000,10,0}
Ethernet for approximately 5 seconds	8	"eth0"	{0,0,0,0}	{0,0,0,0}
Ethernet w/ wget to google	9	"eth0"	{x,x,x,1}	{x,x,x,1}
Ethernet w/ wget to bbc	10	"eth0"	{x,x,x,1}	{x,x,x,1}
Ethernet w/ wget to google 10 times	11	"eth0"	{x,x,x,10}	{x,x,x,10}
Ethernet w/ wget to bbc 10 times	12	"eth0"	{x,x,x,10}	{x,x,x,10}
Ethernet w/ wget to google and bbc	13	"eth0"	{x,x,x,2}	{x,x,x,2}
Ethernet w/ wget to google and bbc 10 times each	14	"eth0"	{x,x,x,20}	{x,x,x,20}

3. Memory Safety Test Table

Test Name	Test ID	Number of Allocations	Number of Frees
Loopback w/ no commands	1	21	21
Loopback w/ 1 ARP commands	2	25	25
Loopback w/ 10 hping packets	5	107	107
Loopback w/ 10000 hping packets	6	83,159	83,159
Ethernet for approximately 5 seconds	8	50	50
Ethernet w/ wget to google	9	1,054	1,054
Ethernet w/ wget to bbc	10	11,734	11,734
Ethernet w/ wget to google and bbc	13	12,094	12,094

4. someuser. (2014, April 16). Comment on "Error in inet_ntoa function." *Stack Overflow*. https://stackoverflow.com/questions/23105644/error-in-inet-ntoa-function

5. Santilli, Cirro. (2013, May 10). Answer to "Print int from signal handler using write or async-safe function." *Stack Overflow*. https://stackoverflow.com/questions/14573000/print-int-from-signal-handler-using-write-or-async-safe-function

6. “tcp_serv_threadpool” source file provided by the module.