CS241 Intrusion Detection System Report

Packet Detection

When detecting a new SYN packet, the program appends the source IP address of this packet to a Linked List. When the program is terminated, the contents of this Linked List is copied into an Array. This array is then sorted using a quick sort. Duplicate IP addresses can then be counted in linear time. The total number of unique IP addresses in the array is exactly equal to the total number of unique IP addresses that the SYN packets have been sent from.

Multithreading

As this intrusion detection system may have to deal with high traffic volumes, it is important that the program is multithreaded. This allows for efficient detection and handling of packets.

When deciding on a threading model, both the “one thread per request” and “thread pool” models where considered. In the final implementation, the “thread pool” model was chosen for the following reasons:

1. The “thread pool” model allows for an upper limit to be placed on the total number of threads in use. Therefore, there also exists a limit on the total number of resources in use at any given time. This means that after receiving many packets, the program will not overload system resources such as memory.

2. Thread creation and deletion operations use time and resources. As this application may have to handle large amounts of traffic, it would be inefficient to create a new thread to process each new packet. Using a thread pool, threads are only created once (at the start of the program) and are subsequently reused. This reduces the overhead experienced per packet received.

When implementing the “thread pool” strategy outlined above, several details had to be considered to maximise the efficiency of the program. Those were as follows:

- Worker threads repeatedly check the “work queue”, waiting until it is not empty. To avoid a busy waiting period, I used wait(&condition_variable, &mutex_lock). Worker threads are put to sleep until more work is added to the queue. By using this method, CPU cycles and energy are efficiently utilised.
- When updating variables that are shared between threads, it is important to avoid “race conditions”. Mutex locks were used to restrict one thread at a time reading from or writing to shared variables.
- As mutex locks were used, it is important to consider possible deadlocks within the code. To avoid deadlocks, I ensured all threads acquired the locks in the same order.

Signal Handling

When a user closes the program, it is important to perform some tasks.

a) Display the output of the sniffer.
b) Shutdown the threads and free any memory in use.

However, it is important to only use async-signal safe functions - functions that can safely be called from within a signal handler. To achieve this, the pcap_breakloop function was called. This function sets a flag that is then used to exit out of the pcap_loop. After this has run, the program then outputs the appropriate information, frees the memory currently in use and closes the threads in the thread pool.

Testing

During testing, the program was evaluated in the following ways:

1. The program was compiled without any errors or warnings.
2. The program did not crash or generate any segfaults during runtime.

The program was run multiple times, each time using a different combination of network interface and packet sending tool (hping, wget or arp-poison.py). Additionally, different amounts of packets were sent to the program. Across all these tests, the program never crashed because of a segfault.

3. Valgrind, a memory error detection tool, did not detect any errors in the program.

When running the program will Valgrind, many packets were sent to the program. As a result, thousands of calls to malloc were issued. Valgrind detected no errors.

4. Helgrind, a tool for detecting errors in multi-threaded programs, did not find any deadlocks, race conditions, or other synchronisation errors.

During testing, the total number of threads within the threadpool was varied to ensure that the program was correctly multithreaded. Helgrind detected no synchronisation issues when running the program with any number of threads.

5. The program was tested for its overall correctness.

By running the program with the interface set to lo, hping and arp-poison.py were used to send packets to the program. By sending a fixed number of packets, we could then compare the output of the program after ctrl-c to the number of packets sent to the program. The output always showed the correct number of packets, which indicated that the program was correctly detecting the packets.

The program was also tested on the eth0 interface using the wget command. By requesting blacklisted urls, it was evident that the program was correctly intercepting and analysing this traffic.

6. The program was also tested for its ability to handle large volumes of traffic without any issues.

Using the hping command and the --faster flag, 1 million packets were sent to the interface where the program was monitoring traffic. The program correctly detected all of the SYN packets sent.