Multithreaded Intrusion Detector in C

Joe Hewett

January 6, 2021

1 Abstract

We explore the design, implementation and testing of a multi-threaded intrusion detection system, that scans a network for packets that contain specific indicators of malicious intent. We will reason about the decisions made and the advantages and disadvantages of the chosen strategies. For more information on the details of the packet parsing itself, see the appendix, where we also discuss future work that needs to be done in pursuit of an optimal solution.

2 Multithreading

Due to the nature of the system we are designing, it is likely to endure heavy load and thus at times, struggle to process the required amount of data. To remedy this, we can split the workload of the system into discrete threads. These threads can process data in parallel which allows us to free up the main thread so that it is able to simply accept packets into the system without needing to do the processing of them itself. This also prevents blockages in the case we encounter a problematic packet that takes a long time to process; since we have $threadCount - 1$ other packets available to process packets, the slow packet will only cause a fraction of the congestion that it otherwise would.

2.1 Threadpool

There are a few threading strategies that we could implement to achieve the desired effect, however in this case opted for a threadpool strategy. This can be though of in perhaps more familiar software engineering terms as a job queue, with "workers" (or sometimes "consumers") that pick jobs from the queue and process them in parallel with other workers. The results are then stored in some centralised location.

2.1.1 Justification of the Threadpool Approach

There are a number of reason why we might opt to use a threadpool strategy, but in order to understand them we should first understand the nature of the problem we are solving and thus the necessary system requirements:

1. We need to deal with a high-throughput that networks handle
2. We need to anticipate bursts in network activity
3. We need to reduce additional packet transmission overhead

From here, it becomes more clear how the threadpool strategy is optimal for our use case;

1. Instantiating threads once when the program starts rather than on a per-job-basis during runtime reduces the overhead caused by the intensive process of creating and destroying threads

2. Having excess threads that exist and are ready to process jobs means we can handle bursty network activity more effectively. If we suddenly encounter a burst in a system that dynamically creates threads when needed, this could exacerbate the problem due to the overhead of creating threads.

3. We also benefit during bursts of network activity from the constrained number of threads since we don't have the thread of too many threads being allocated at once - we always have a fixed count and thus won't run into performance issues caused by the uncontrolled creation of threads during runtime.

2.1.2 Disadvantages of the Threadpool Approach

Having made these points, we do have to consider the disadvantages that come with the threadpool;

1. We cannot be sure of when the job we are submitting will be processed. If there is a backlog in all threads and we do not have the ability to dynamically allocate more threads, we may end up in a situation where a high-priority packet does not get processed and we have no way of determining when it will.

2. Thrashing - if we allocate a number of threads far larger than the processing of the network activity requires, we lock out resources unnecessarily.

However, after evaluating these key disadvantages we can see that the impact of both can be remedied by careful analysis of the throughput of the network we are analysing, so that we can implement the system with the optimal thread count. In this case, we'd reduce the likelihood of both of the issues becoming a reality. This analysis is something we will explore in the future work section.

2.2 Threadpool Optimisations

2.2.1 Queue Conditions

Prior to processing, the packets are placed into a queue. The threads run in an infinite loop that picks jobs from this queue. However, in the case the queue is empty during troughs of network activity, this can result in processing time being used on constantly trying and failing to pull jobs from the queue.

To avoid this, we utilise a *pthread* condition, on which the threads wait. When we queue a packet, we will send a *pthread* condition signal, which prompts the threads to awaken and attempt to pull from the queue. Note we use *pthread_cond_signal* rather than *pthread_cond_broadcast* to do this, to prevent spurious wake up.

3 Testing

3.1 Basic Functionality Testing

To begin, we tested the basic functionality of the program - whether on a surface level, the program does what we expect it to.

1. Run "hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 –rand-source localhost" on the loopback interface and ensure that we see 100 packets with maximum of 100 unique IP addresses

2. Run "hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 localhost" on the loopback interface and ensure we get 100 packets with a maximum of 1 unique IP address

3. Run *pythonarp − poison.py* and ensure we see 1 ARP Poisoning Attack

4. Run the previous 2 commands and ensure we see 100 SYN Attacks, 1 ARP Poison

5. Run *wgetwww.google.co.uk* on the *eth*0 interface and ensure we see 1 Blacklist Violation

3.2 Boundary Testing

Now that we have ensured that the basic functionality works, we need to check the boundaries of the program to ensure it works at extreme values and edge cases

1. Send 100,000 packets on the loopback interface with rand-source off and check that we see 100,000 packets received from 1 IP address

2. Send 100,000 packets on the loopback interface with rand-source on and check that we see 100,000 packets received from less than or equal to 100,000 IP addresses

```
100000 SYN packets detected from 99985 different IPs (could be a SYN attack)
0 ARP responses
0 Blacklist responses
```

3. Run a script to send 100,000 ARP Packets on the loopback interface - ensure that 100,000 are found by the system

3.3 Profiling

We tested the running time of the program to ensure that we were seeing the improvements in speed that we'd expect from a multithreaded approach. An example from the 10 Thread profile:

```
real    0m19.299s
user    0m0.280s
sys     0m0.408s
```

3.4 Static Code Analysis

We ran some static code analysis tools to ensure that the program is working as intended.

1. We first run the program through *gdb* to ensure the flow of the program is as expected. In *gdb* we set breakpoints throughout the code and stepped over each critical section to check the behaviour is as expected. We printed the value of variables through the program to ensrue everything was running according to predictions. We were aided with *stack* and *list* which allowed us a more detailed view at the internal workings of the system.

2. We ran the program through *valgrind* and checked that resources were being correctly allocated and freed. On first running, we noticed that not all of the memory was being managed correctly:

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 3 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
```

However, after some searching we discovered a call to *strdup* which masks a *malloc*. After successfully freeing the returned pointer, we saw the following:

```
==3245==
==3245== HEAP SUMMARY:
==3245==     in use at exit: 0 bytes in 0 blocks
==3245==   total heap usage: 198,170 allocs, 198,170 frees, 2,642,049 bytes allocated
==3245==
==3245== All heap blocks were freed -- no leaks are possible
==3245==
==3245== For counts of detected and suppressed errors, rerun with: -v
==3245== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 17 from 8)
```

4 Appendix

4.1 Future Work

4.1.1 Optimising Thread Count

One of the prerequisites for successfully using a threadpool is an optimised *threadcount*. By this we mean setting a number of threads prior to runtime that is not so low that it creates a backlog due to inadequate processing capacity, but not too high that it results in thrashing. There are numerous metrics we could look into to aid us in this process, such as average throughput over some amount of time, and peaks and troughs of network activity, as well capacity of the machine on which the detection system is running. We will not go into further detail on this, but it is something that needs to be looked into for this system to be maximally efficient.

4.1.2 The Best of Both Worlds

Another enhancement that could be used in substitute to 4.1.1, would be a method of dynamically resizing the number of threads during runtime, depending on the network activity. This is similar to the standard one-thread-per-job model, with the difference being that the thread count is not set on a per-packet basis, but rather with less granularity. For example, when we notice that the system has been under strain for some predetermined number of minutes with some predetermined backlog threshold, we may decide to increase the threadcount by some fixed or calculated amount. If the converse is true, we may like to reduce the threadcount by some fixed or calculated amount. Further, if we have reason to expect an incoming spike in network activity, we may want to change the threadcount beforehand in anticipation.

4.2 Parsing Packets

The main objective of the system was to intercept packets and analyse them for potentially malicious content. Here, we briefly outline how the packets were parsed and what content was extracted.

4.2.1 SYN Attacks

The synchronisation bit is a flag present in TCP packets that is part of the TCP's 3-way handshake used to initiate a connection. TCP flooding involves sending a large number of these packets to open half-open connections on the server which consumes server resources, causing legitimate traffic to be unable to connect. The method for identifying these packets is as follows:

1. Check that the protocol we're dealing with is TCP by checking the IP header
2. Determine a pointer to the start of the TCP header. We can do this by extracting the length of the IP header and then appending this to the memory location at which we know the IP header starts.
3. When we have a pointer to the start of the TCP header, look at the control flags located within the packet. There are 6 - we want to identify the packets where the SYN bit is 1, and the other 5 are 0.
4. If this is the case, add the source address located within the IP header to a dynamically growing array

Dynamically Growing Array - Since we want to identify not only the number of SYN packets we receive, but also the number of unique IP addresses from which we receive the packets, we need to store the source address in dynamically resizable array, rather than blocking off a large section of memory before runtime due to the fact we cannot be sure of the correct size to allocate to the structure. We implement this array with a data structure called *growingarray* which contains not only the array of source addresses, but also an *int* denoting the current capacity, and another indicating the capacity we have used. Upon encountering the case at which the used elements is equivalent to the total capacity of the array, we reallocate memory equal to double the current memory to the array, before adding the element.

4.2.2 ARP Poisoning

ARP poisoning is an attack in which malicious ARP packets are crafted and sent to a default gateway on a network to change the pairings in the IP to MAC address table. This can give rise to "man-in-the-middle" where traffic is routed through the attackers device before it is forwarded on to the actual destination. In order to detect these packets, we do the following:

1. Check the protocol of the packet and ensure it is not TCP
2. Construct an APR header from the default *ether_arp* structure
3. Check if the operation (*ar_op*) is of type $ARPOP\_REPLY$. If so, we have found an ARP response.

4.2.3 Blacklist Violations

Often, malicious programs in a system will communicate with an external server to transfer data or receive instructions. One way of filtering out packets in a network is to store a blacklist of malicious addresses, and check the destination of outbound packets before the exit the network. In this instance, we want to check whether or not the destination is *www.google.co.uk*. In order to do this, we do the following:

1. Check if the packet is a TCP packet, and has a payload
2. If there is a payload, we want to check it for a specific line, containing $Host$: - note we don't just global search the payload for the blacklist entry directly since it may exist out of context within the paylaod, without being the host.
3. If we find a line with $Host$: we then want to string match the blacklist entry, in this case *www.google.co.uk*. If we identify this, then we can confirm this packet contains a blacklist violation.

5 Bibliography

References

[1] Casablanca et al. (2010), "C Dynamically Growing Array", StackOverflow - https://stackoverflow.com/questions/3536153/c-dynamically-growing-array/35362613536261

[2] ICISS (2005), "ARP Poisoning Information Systems Security: First International Conference", Kolkata, India

[3] Arpan Mukhopadhyay (2020), "tcp-serv-threadpool.c", CS241 Module Page - https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/os2020/

[4] Raymod Chen (2020), "Spurious wake-ups in Win32 condition variables", Microsoft Devblogs - https://devblogs.microsoft.com/oldnewthing/20180201-00/?p=97946