u2053845

CS241 Report

DESIGN AND IMPLEMENTATION

Initially, I decided I would decode each packet header, allowing me to parse through only the relevant headers needed to complete the tasks needed. In order to access the data/payload using the tcphdr and iphdr, I had to use defined structs from the netinet package files to break the packets coming into the system into their respective headers for multiple layers. The ethernet layer was the most outer layer, which needed to be parsed first, then the Ip layer and then tcp. This is why my if statements to parse through are embedded in this way, to ensure all layers can be parsed through. This is important as the system uses the previous header to identify what protocol is next. The way pointer arithmetic works was used to operate packets into individual structs. I used the line '*if (ntohs(headerEther->ether_type) == ETH_P_IP)'* to find out if the packet that the ethernet header declared was of type IP. If so, then the IP header was populated. Similarly, if the protocol in the IP header is TCP, the the TCP header is populated.

Once the packets are broken down and the headers are populated, the system can then identify malicious activities. For example, for blacklisted URLs, the address of the request asking for the webpage information is stored in the HTTP header. I used the method of strstr to identify the exact strings 'google.co.uk' and 'bbc.com'. If this returned not null, then a blacklisted website was entered, so increment total black list count.

To identify cache poisoning, the protocol used was ETHERTYPE_ARP, which is defined in if_ether.h, and the arp count was incremented.

In order to detect a SYN flooding attack, I had to find all SYN packets sniffed. I had to find all packets with either the SYN flag or the ACK flag set, and incrementing a counter if this was the case. To find all unique ip addresses I decided to implement a dynamically growing array. I used structures to do this, as they allowed me to combine a group of related variables to be placed in a contiguous block of memory. I iterated through the array to look for unique ip addresses in the array, and using Boolean algebra, if a packet was encountered before, then I would use the a insertArray function to add it to the array.

I used mutex locks for proper thread synchronisation, so that when assigning values to global variables, there is no overwriting of values. Once a thread locks a variables, no other thread can change its value until the variable has been unlocked.

A SIGINT signal could be sent upon entering control C to stop the system and return the intrusion detection report. These variables had to be made global, so they can be used in the displaying of the report. All threads and pcap_loop is stopped.

I designed a multithreaded approach to deal with high load packets not being able to be dealt with in a single threaded system. Upon entering the system, packets are added to a queue, and wait to be processed until a thread requests a specific packet. Implementing a queue structure from scratch means that a new thread does not need to be created for every packet, and processing can happen simultaneously. The client can accept requests, creating a connection socket, but in the meantime passes these requests to threads which can handle new requests while previous ones are still being processed. Pcap_loop() is used to consistently fetch packets, and calls the callback function, where packets are sent to dispatch to be processed. Here, initialising of a packet element struct occurs and is added to a queue via a mutex lock. The mutex locks ensure synchronisation of threads. The number of threads is set to 2, as the virtual machine uses 1 core, so 2 threads are ideal. This was set using #define NUMTHREADS 2.

TESTING

The main way of testing was using the verbose mode provided. Initially, setting verbose to 1 when testing for SYN flooding attack meant that I could observe the exact structure of the packet, and verify manually whether each packet was correctly printing out SYN packet detected. Setting verbose to 0 also outputs 100 SYN packets from 100 ips, which was a trivial test. The same situation applied to testing for cache poisoning and blacklisted Urls, by setting verbose to 0 throughout the coding process until the desired output was achieved.

There are however bugs with the IP address of SYN packets upon the blacklist test, as extra Ips are being allocated to the count, which could potentially be resolved by looping through the pointer of the dynamic array.

Another way of testing is using Valgrind, which displays all memory leaks and how memory is used on the computer. A few memory leaks are present however, as freeing up of heap memory is not present, due to complications with the IP addresses. This would be an issue that can be worked in future iterations.

The system should be able to handle high loads via the use of multi-threading. For example, the cache poisoning python script can be modified to send a high number of ARP packets, using the following …

```
for i in range (int(sys.argv[1])):
        send(arp)
```

A similar approach can be used for SYN flooding and Blacklisting URLs; more specifically running a bash script running wget i number of times for either google.co.uk or bbc.com. i is a command line argument here.

A multithreading approach can also be tested using speed, if the method captures packets quicker than singular threaded models, it is more efficient. For instance, we could compare which model processes and returns 10000 arp packets first.

In conclusion, the development of threading is a elaborate process, and therefore, there several improvements could be made. There is a potential to adapt to cover additional types of attacks to add further functionality. The system efficiently adapts the theory of parsing through packets practical in a efficient manner.

REFERENCES

https://datatracker.ietf.org/doc/html/rfc793#page-15

https://datatracker.ietf.org/doc/html/rfc791#page-11

CS241 – Lab 3 Code

https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/cn2021/cn_lec2_courseworktopics.pdf

https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

PThreads Primer - A Guide to Multithreaded Programming , Bil Lewis , Daniel J. Berg