CS241 COURSEWORK

I. Design Choices

1. Packet Sniffing

This program uses the pcap_loop() function to sniff packets. pcap_loop() is often preferred over pcap_next() because it lets the user specify a callback function that is executed each time a packet is captured. This lets the user process the packet immediately, instead of waiting until all packets have been captured and then iterating over them one by one. This can make the packet-capturing process more efficient and effective, particularly when handling a large number of packets.,(Write reasons with reference). The whole process of parsing the headers off the packet it is all happening in the analysis.c file. Where, if the verbose flag is on, it contains all the print statements to show the packet header information in the terminal.
The Ethernet Header is the first header that is parsed. We use the ether_header struct form which we check the what ethernet type it is and which will determine how we parse the packet next.

The link layer has length 14 so we adjust the next pointer according to that.If it is an ARP packet then we assign the remaining header to the arpstruct. This struct contains one variable type struct arphdr which is defined in the and 4 more other 8 bit unsigned integers. The first will give us information such as operation and hardware type, while the rest fields store the relevant IPs and MAC addresses

We use iphdr to extract the IP header when the packet is type IP. Most of the struct's fields are used only as a simple output. But the protocol type is important to us because it will determine if we have to parse a UDP or a TCP header next (the other protocol types are just ignored by this program). The pointer for the header of the next layer while be adjust by knowing that the IP length is four times the value stored in the ihl field. If UDP is next, it will just give use source port, destination port and length. The TCP will also have certain flags that will be needed later and if the destination port is 80 it means that an HTTP response/request may be found in the payload at the end of the TCP pointer with and therefore output it.

2. Intrusion Detection

ARP Cache Poisoning: We detecting by checking if the ETHERNET type is ARP . After that in the parsing of the ARP header we check if it is a Response, to add it to the struct list . This list has unique elements and it is passed as a pointer parameter to the analysis file.

SYN Flooding Attack: Our system detects the number of SYN packets that are sent. Such a packet has all the flags set to 0 except the Syn flag in the TCP header. We add the IP of the SYN packet to the list of IPs that we have created. This list is common between threads because every element inserted needs to not be there before. This is because at the end we also need to count the different IP sources that sent SYN packets.

Blacklisted URLs : To avoid any potential attacks from the given malicious domains(www.google.co.uk, www.facebook.com), the sniffer checks if the destination port of the TCP layer is set to port 80 and uses the strstr function to search for the string of one of the blacklisted websites in the HTTP header's data offset byte. If both conditions hold, the counter for the blacklisted website detected is incremented. This helps to ensure that only trusted domains are allowed to pass through.

3. Multithreading

Due to the large amount of packets that are being transferred, it is essential that the program runs with out significant delays. An arp cache poisoning attack may send thousands of packets at very small burst of times. Multithreading in a network application, can improve the lateness, throughput and latency. In this program we chose to use a Thread pool. The alternative of one thread per packet it can work only when we have small number of packets which is not the case. Also Thread Pool means that we create less complex code and it can be more adaptable in the future. We create the threads in the dipatch.c file. We use a custom struct packetqueue to store the packets with the associated headers in the queue to be parsed by one of the threads. It is important that at the enqueue we place both the packet and the header in the heap memory, so that they can be accessed by any thread.

According to a published paper by Chinh (Chinh, Nguyen Duc and Kandasamy, Ettikan and Khei, Lam Yoke, 2007) the largest challenge is to achieve Synchronisation. Errors can occur if shared memory between threads is not being taken care off. For our threads we initialise two mutex locks and use one when needed to access the queue of the thread pool, and the other to insert an element in the shared list of SYN IP addresses.

handle_thread() is the function called that each thread will be executing. Each time a packet is parsed, the counter struct of that packet is added to the one of the thread. When the cntl+c is pressed the signal handler function will call the close threads(). At that point the counter of each thread is added together in the total parameter so that sig_handler() function will print it.

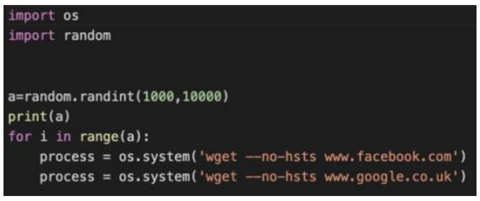
II. Testing

1. Testing if the number of threads is sufficient

It is a fact that each thread created has a runtime overhead that costs a considerable amount of time (Ling, Yibei and Mullen, Tracy and Lin, Xiaola, 2000). We need to find the right balance -which is the lowest amount of threads that achieves the most? We use the lscpu on the terminal and see that the Virtual Machine is single core. Which implies that the number of threads must be minimal, since the use of a single shared cache between threads means that we will come across of more synchronization problems (Koufaty, David and Reddy, Dheeraj and Hahn, Scott, 2010). Therefore we choose 2 threads only, which can also be confirmed by using the clock() of the time.h to count the time it takes for different number of threads to do the same amount of work.

2. Testing if the detection system works correctly

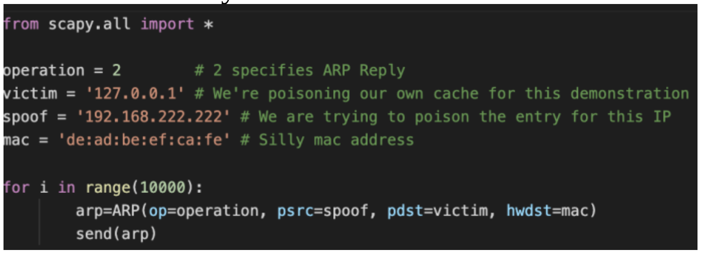
It is essential to test the correctness of the intrusion system. For that I created some simple python files. The *Figure 1* program sends a random amount(which will be printed to know what to expect) of HTTP requests of the blacklisted urls. Whilst the *Figure 2* one sends a large amount of ARP requests. We also test the SYN flood attack by using modification of the command hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost. We changed the number of -c and u flag to see how the program responds in variable amount and time that SYN packets are sent. It can be concluded that the maximum number of packets of this command that the program can handle is 80,000 with u100. The main cause of this result is the memory that is available of the system.



```
import os
import random


a=random.randint(1000,10000)
print(a)
for i in range(a):
    process = os.system('wget --no-hsts www.facebook.com')
    process = os.system('wget --no-hsts www.google.co.uk')
```

Figure 1



```
from scapy.all import *

operation = 2        # 2 specifies ARP Reply
victim = '127.0.0.1' # We're poisoning our own cache for this demonstration
spoof = '192.168.222.222' # We are trying to poison the entry for this IP
mac = 'de:ad:be:ef:ca:fe' # Silly mac address

for i in range(10000):
        arp=ARP(op=operation, psrc=spoof, pdst=victim, hwdst=mac)
        send(arp)
```

Figure 2

Finally, we need to make sure that there are no mistakes in the allocation of memory, i.e. no memory leaks, all allocated memory is free'd , correct allocation of malloc blocks, etc. To do that we used the valgrind tool with --leak-check=full and -v flags at the start of the execution command. This uses as default the Memcheck which according to (Seward, Julian and Nethercote, Nicholas, 2005) it is very reliable with extremely low probability of failure.

Bibliography

Chinh, Nguyen Duc and Kandasamy, Ettikan and Khei, Lam Yoke. (2007). Efficient Development Methodology for Multithreaded Network Application. *2007 5th Student Conference on Research and Development*, (pp. 1-5).

Koufaty, David and Reddy, Dheeraj and Hahn, Scott. (2010). Bias Scheduling in Heterogeneous Multi-Core Architectures. *Proceedings of the 5th European Conference on Computer Systems* (pp. 125–138). Paris, France: Association for Computing Machinery.

Ling, Yibei and Mullen, Tracy and Lin, Xiaola. (2000). Analysis of Optimal Thread Pool Size. *SIGOPS Oper. Syst. Rev.*, 42–55.

Seward, Julian and Nethercote, Nicholas. (2005). Using Valgrind to Detect Undefined Value Errors with Bit-Precision. *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (pp. 17-30). Anaheim, CA: USENIX Association.