CS241 REPORT

Mohammed Rafi

Introduction

In this report We will go over our design methodology and testing methods when making my intrusion detection system for CS241.

Design Choices

analysis.c

As the packet is sent to `analyse()` as a `unsigned char *` we were able to manipulate it and access the different layers of the internet protocol stack[1]. Using the `dump()` method in `sniff.c` as reference we could access the data in the Ethernet, IP and TCP structures by incrementing the length of the pointer by certain lengths. As the Ethernet header was the first layer we did not need to change the length of the pointer, however to access the IP header we had to increment the pointer by 14 as the Ethernet header has a constant length. To access the TCP header it was a different situation as the length of the IP is not constant, however we can get the length of the header by multiplying the data offset of the IP header by 4.The TCP header also has a non constant length so to access the next layer (the data payload) we would then increase the pointer by the TCP data offset multiplied by 4. With all the structures allocated and accessible we could then move onto our intrusion detection system.

To detect unique IP addresses We created a dynamically growing `char` array which would only store unique IPs, to check whether they were unique We created a method called `valueinarray()` which would take an array and a value as values and using `strcmp()` would loop through the array and checked if the value existed in the array. For sake of simplicity I decided to reallocate the array every time a new IP was inserted, however as `realloc()` is a expensive operation in a best case scenario we would of doubled the array capacity every time it reached its limit.

To detect SYN packets being sent we check the TCP headers control bits and checked whether the SYN bit was the only bit which was enabled.

To detect ARP responses we would check whether the Ethernet headers `ether_type` was `ETHERTYPE_ARP` which would imply that the next layer contained an ARP header, we then would access the ARP header and then check whether the `arp_op` was 2 which would signify an ARP response.

To detect blacklisted URLs we accessed the payload of the packet by skipping past the Ethernet, IP and TCP layers and accessing the data payload. We then cast `char *` to the payload to allow us to check it as if it was a string. Using `strtstr()`[2] we could check whether the payload contained the string ”www.google.co.uk” which would signify that the packet is coming from that URL.

sniff.c

`sniff.c` would originally run a `while` loop which would dispatch incoming packets with `pcap_next()`—[3] however this lead to complications when trying to catch signals and output the intrusion report. We decided to use `pcap_loop()`[4] to take incoming packets and would then parse the packets to `packet_handler()` which would parse them into `dispatch()`. I also declared `pcap_handle` globally to allow us to use `pcap_breakloop()`[5] when we catch a signal. To stop any memory leaks from occuring using `pcap_loop()` we closed the connection with `pcap_close()`[6] which freed resources used by our `pcap` handler.

dispatch.c

`dispatch.c` contained my signal catching and threading code however the threading is described later on the report so we will go over the signal catching.

Signal catching

Using `<signal.h>`[7] we could create a function which could catch a signal (in our case a SIGINT) and run code instead of exiting the program normally, our signal catcher would first join the current threads after they are finished analysing their current packets and would close any pcap functions, We would then print out our intrusion detection report which looks like this -

```
1 SYN Packets detected from 12 different IPs (SYN Attack)
4 ARP responses (cache poisoning)
5 URL Blacklist violations
```

After printing the report we would then free any dynamically allocated variables, in our case, the array storing unique IPs and our packet queue. After that we could safely exit the program.

Testing

SYN Flooding Attack

To test if the application would recognise whether a SYN Flooding attack we used the `hping3` package to send packets. The command

```
hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost
```

was used to send 100 packets with the SYN flag enabled with random IPs to our loop back device (lo). Running this command while also running the intrusion detection application leads to the report showing -

```
Intrusion Detection Report:
100 SYN Packets detected from 100 different IPs (SYN Attack)
0 ARP responses (cache poisoning)
0 URL Blacklist violations
```

which is correct, I also tested by sending 1000 random packets and the report outputted `1000 SYN Packets detected from 1000 different IPs (SYN Attack)`. However the program seems to break at around 5000 syn packets due to running out of memory.

ARP Cache Poisoning

To test for an ARP cache poisoning attack we ran the given `arp-poison.py` script but modified the script to loop and send multiple packets, the program correctly identified the packets went sent 1, 10, 100, 1000 and 10000 ARP packets.

Blacklisted URLs

To test for blacklisted URL violations I used the command `wget` to get packets from websites. First I tested whether it would correctly report that we have a packet from a blacklisted URL using `wget www.google.co.uk` which correctly displayed `1 URL Blacklist violations` and then ran the program with a non blacklisted URL using `wget www.bbc.co.uk` which also correctly outputted `0 URL Blacklist violations`.

Threading Method

Choice and Justification

We decided to use the Threadpool model of threading as it made the most sense for this use case compared to one thread per packet model as the number of packets which are sent can vary tremendously so creating 1000 threads for 1000 packets would be silly due to the poor performance as compared to queuing the packets over a fixed amount of threads to analyse.

Implementation

To implement the threadpool We modified the `queue.c` from `multithreaded_serv.zip`[8] to take in `unsigned char *` items instead of int and modified `dequeue()` to return the dequeued item. The `threadpool()` method would then wait until the queue is not empty and then dequeue and analyse the packet. We also had to be wary of race conditions so we included mutex locks around global variables being incremented so they would not be incremented twice.

Bibliography

[1] DARPA INTERNET PROGRAM. *INTERNET PROTOCOL*, 1981 (accessed Dec, 2020).

[2] TutorialsPoint. *C library function* - `strstr()`, 2020 (accessed Dec, 2020).

[3] die.net. `pcap_next(3)` - *Linux man page*, 2020 (accessed Dec, 2020).

[4] tcpdump. *Man page of* `pcap_loop()`, 2020 (accessed Dec, 2020).

[5] die.net. `pcap_breakloop(3)` - *Linux man page*, 2020 (accessed Dec, 2020).

[6] die.net. `pcap_close(3)` - *Linux man page*, 2020 (accessed Dec, 2020).

[7] HIMANSHU ARORA. *Linux Signals, Example C Program to Catch Signals (SIGINT, SIGKILL, SIGSTOP, etc.)*, 2012 (accessed Dec, 2020).

[8] Arpan Mukhopadhyay. `multithreaded_serv.zip`, 2020 (accessed Dec, 2020).