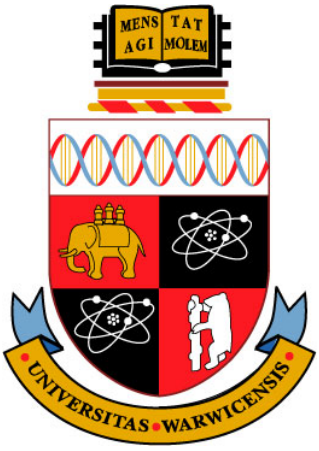

CS241 Report

Packet Sniffer

Josh Turner

Contents

List of Figures

1 Introduction

The aim of this project was to implement a packet sniffer using C to detect three types of malicious packets: SYN Flooding, ARP Cache Poisoning, and Blacklisted URLs. The fast part of this task involved designing data structures and implementing a single-threaded solution. Once this was done, a multi-threaded solution was designed and implemented using the single-threaded solution as a guideline.

2 Implementation

2.1 Overview

The application begins by initialising the thread pool with initPool(). Once this is done the main thread uses pcap_loop to call dispatch() for each captured packet to append them to the queue. Once on the queue the packets are dequeued by one of the threads in the thread pool and then analysed. Once the user terminates the program, the threads each finish processing packets and then join with the main thread where their collected data is aggregated and output to the user.

2.2 Data structures

The following structures were used to pass shared variables and individual thread data between different functions and threads:

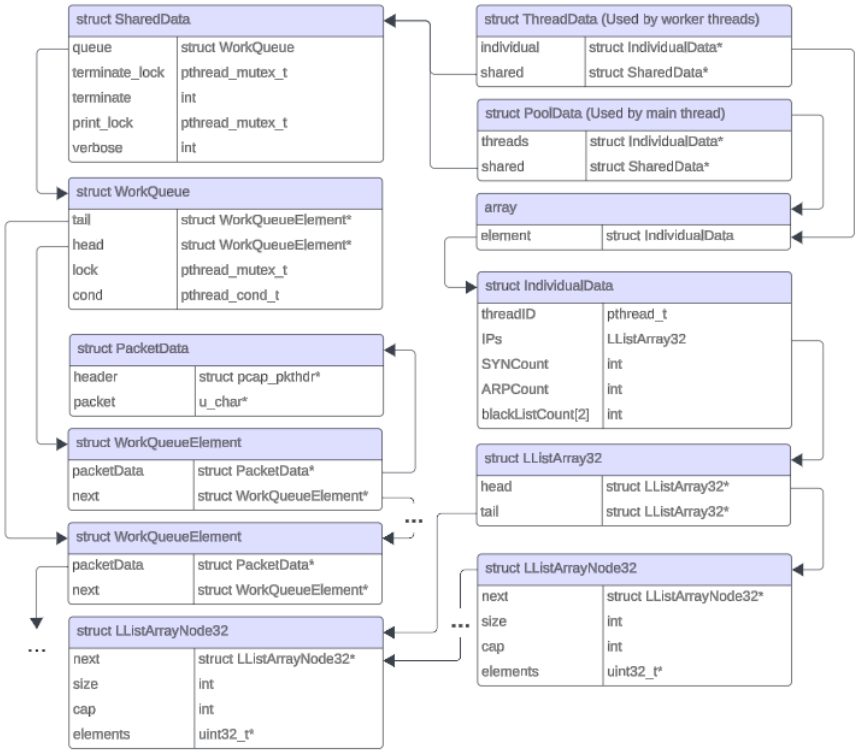

Figure 2.1: Structures

The main thread uses a single instance of PoolData to access the individual data of every thread and the shared resources, whereas each worker thread uses an instance of ThreadData to isolate them.

A set structure was also created, but is not used until all the worker threads have terminated:

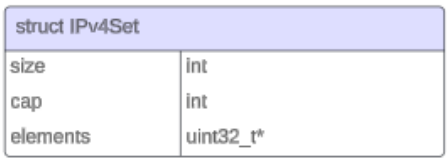

Figure 2.2: set structure

2.3 Analysis

2.3.1 SYN Flooding Attack

These attack were caught by checking if the protocol in the ethernet header was IPv4, the protocol in the IPv4 header was TCP, and only the SYN flag was set in the TCP header.

2.3.2 ARP Cache Poisoning

These attack were were caught by checking if the protocol in the ethernet header was ARP, and the operation in the ARP header was reply.

2.3.3 Blacklisted URLs

These attack were caught by checking if the protocol in the ethernet header was IPv4, the protocol in the IPv4 header was TCP, the destination port in the TCP header was 80, and that the host in the HTTP header was blacklisted.

2.4 Efficiency

2.4.1 Pointers

When analysing packets the application casts the packet pointers to one of several header structures. This is an efficient approach as it allows direct access to the header fields without copying the original data into a new struct.

2.4.2 Linked list array

Each worker thread uses a LListArray32 struct to store IPs from the SYN packets. This was done so that each thread can collect IPs at a faster rate than other structures such as dynamic arrays or linked lists which use a large number of allocations, or copy data when resizing. Instead, the LListArray32 doubles in size by creating a node which contains an array, and then links this to the previous node like a linked list.

2.4.3 pcap_loop

The application uses pcap_loop() to capture packets; this is more efficient than pcap_next() as it calls dispatch() on its own without requiring manual handling of packets.

2.5 Synchronisation

The threads use three mutex locks in total for:

- Accessing/modifying the terminate flag
- Printing, since printViolations() uses printf multiple times
- Modifying the queue

When modifying the queue, the main thread acquires the lock, enqueues an item, then unlocks it. However, the worker threads will instead acquire the lock and check if the queue is empty. While the queue is empty they enter a loop where they terminate if the flag is set and otherwise release the lock to wait for the condition variable. Once the condition variable is signalled and the queue is not empty, or if the queue was never empty, the thread will dequeue a packet and release the lock.
The condition variable is signalled in four cases:

- A packet was enqueued
- A packet was dequeued
- pcap_loop terminated
- A thread terminated

This ensures that packets are always dequeued when available and that threads always terminate when the flag is set.

```c
// Repeatedly removes an element from the queue if one exists,
// or waits until signalled if queue is empty
// Terminates when terminate flag is set and the queue is empty
void* collect(void* arg) {
    struct ThreadData* threadData = (struct ThreadData*)arg;
    struct SharedData* shared = threadData->shared;
    struct PacketData* packetData = NULL;
    struct WorkQueueElement* element = NULL;

    while (1) {
        // Dequeue element from work queue
        pthread_mutex_lock(&shared->queue.lock);

            // Wait till queue contains an element, awake when signalled
            while (shared->queue.head == NULL) {
                // If the program terminates,
                // release locks and signal so another thread can continue
                pthread_mutex_lock(&shared->terminate_lock);
                    if (shared->terminate) {
                        pthread_mutex_unlock(&shared->terminate_lock);
                        pthread_cond_signal(&shared->queue.cond);
                        pthread_mutex_unlock(&shared->queue.lock);
                        pthread_exit((void*)threadData);
                    }
                pthread_mutex_unlock(&shared->terminate_lock);

                pthread_cond_wait(&shared->queue.cond, &shared->queue.lock);
            }
            // Dequeue
            element = shared->queue.head;
            shared->queue.head = shared->queue.head->next;
        // Release lock and signal so another thread can continue
        pthread_cond_signal(&shared->queue.cond);
        pthread_mutex_unlock(&shared->queue.lock);

        // Analyse dequeued packet
        packetData = element->packetData;
        analyse(threadData, packetData->header, packetData->packet);

        // Release memory
        free(element);
        freePacketData(packetData);
    }
}
```

Figure 2.3: collect()

```c
// Adds an element to the work queue, then signal to allow thread to take packet
void enqueue(struct WorkQueue* queue, struct PacketData* packetData) {
    struct WorkQueueElement* element = initWorkQueueElement(packetData);
    pthread_mutex_lock(&queue->lock);
        if (queue->head == NULL) {
            queue->head = element;
        } else {
            queue->tail->next = element;
        }
        queue->tail = element;
    pthread_cond_signal(&queue->cond);
    pthread_mutex_unlock(&queue->lock);
}
```

Figure 2.4: enqueue()

2.6 Signal Handling

When the packet sniffer receives the SIGINT it calls signalHandler():

```
13  // Terminates pcap_loop
14  void signalHandler(int sig) {
15      if (pcap_handle != NULL) {
16          pcap_breakloop(pcap_handle);
17      }
18  }
```

Figure 2.5: signalHandler()

pcap_breakloop() is used to terminate pcap_loop() since it is safe to call in a signal hander[1]. Once the loop is broken, the terminate flag is set causing all threads to terminate once the queue is empty.

2.7 Memory Safety

2.7.1 Memory allocation

In order to ensure that memory was allocated correctly over the course of the program validateAlloc() is called after each malloc() and calloc().

```
6   // Prints error message and exits if memory allocation fails
7   void validateAlloc(void* ptr, char* message) {
8       if (!ptr) {
9           fprintf(stderr, message);
10          exit(EXIT_FAILURE);
11      }
12  }
```

Figure 2.6: validateAlloc()

This ensures that any pointers used by the program are valid.

2.7.2 Memory deallocation

Each time memory is allocated it is also deallocated. There are four places in which memory must be freed by the program.

- In collect(), after analysing each packet, the pointers to the metadata and packet are freed.
- In analyseHTTP() the host is freed after it has been compared with the blacklisted URLs.
- In collect(), the thread's ThreadData struct is freed before terminating
- In sniff(), freeListIntoSet() is called, which inserts all elements into the set and frees the structure.
- In sniff() all memory associated with the thread pool and set are freed via freePoolData() and freeIpv4Set().

```
32  void freeListIntoSet(struct LListArray32* list, struct IPv4Set* set) {
33      struct LListArrayNode32* node = list->head;
34      struct LListArrayNode32* nextNode;
35      int i;
36      while (node != NULL) {
37          for (i=0; i<node->size; i++) {
38              addIPv4(set, node->elements[i]);
39          }
40          nextNode = node->next;
41          free(node->elements);
42          free(node);
43          node = nextNode;
44      }
45  }
```

Figure 2.7: freeListIntoSet()

```
178  // Releases memory for thread pool
179  void freePoolData(struct PoolData* pool) {
180      free(pool->threads);
181      free(pool->shared);
182      free(pool);
183  }
```

Figure 2.8: freePoolData()

```
20  // Releases memory used by sets
21  void freeIpv4Set(struct IPv4Set* set) {
22      free(set->contents);
23      free(set);
24  }
```

Figure 2.9: freeIPv4Set()

3 Testing

To test the application several scripts were used. Each of these tested a separate aspect of the specification, and used Valgrind to check for memory leaks.

3.1 SYN Flooding Attack

The blacklisted URL detection was tested using the following bash script. As expected, there were 3 unique IP addresses, and 18 SYN packets in total.

```
1  #!/bin/bash
2  hping3 -c 1 -d 120 -S -w 64 -p 80 -i u1 -a 1.1.1.1 localhost
3  hping3 -c 5 -d 120 -S -w 64 -p 80 -i u1 -a 1.1.1.2 localhost
4  hping3 -c 10 -d 120 -S -w 64 -p 80 -i u1 -a 1.1.1.3 localhost
5  hping3 -c 2 -d 120 -S -w 64 -p 80 -i u1 -a 1.1.1.1 localhost
```

Figure 3.1: synTest.sh

This script was used to stress test the application. When using Valgrind, it was only able to capture 700,000 - 800,000 packets. However, without Valgrind, the sniffer was able to capture every packet since it runs faster.

```
1  #!/bin/bash
2  hping3 -c 1000000 -d 120 -S -w 64 -p 80 -i u1 --rand-source localhost
```

Figure 3.2: synStressTest.sh

3.2 ARP Cache Poisoning

The ARP cache poisoning was tested via the following script which sent a random number of ARP requests and responses. As expected, the application's output matched the script's output each time.

```python
#!/usr/bin/env python
from scapy.all import *
from random import randint

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
request = 0
response = 0

for i in range(20):
    operation = randint(1,2)
    if operation == 2:
        response += 1
    else:
        request += 1
    arp=ARP(op=operation, psrc=spoof, pdst=victim, hwdst=mac)
    send(arp)
print("responses:", response)
print("requests:", request)
```

Figure 3.3: arp-poison.py

3.3 Blacklisted URLs

A similar script was used for blacklisted URLs which tested each of the blacklisted URLs as well as a third non-blacklisted URL. As expected, the application also passed this test.

```bash
#!/bin/bash
bbc=0
google=0
amazon=0
for i in $(seq 1 100);
do
    request=$((0 + $RANDOM % 3))
    if [ $request -eq 1 ]
    then
        wget --no-hsts www.google.co.uk &>/dev/null
        google=$((1+google))
    elif [ $request -eq 2 ]
    then
        wget --no-hsts www.bbc.co.uk &>/dev/null
        bbc=$((1+bbc))
    else
        wget --no-hsts www.amazon.co.uk &>/dev/null
        amazon=$((1+amazon))
    fi
done
echo Made $google requests to google
echo Made $bbc requests to bbc
echo Made $amazon requests to amazon
```

Figure 3.4: httpTest.sh

4 Reflection

If this project were to be redone, it may be beneficial to implement a method for providing priority access to dispatch() when it wants to enqueue. This would avoid cases in which dispatch() has to wait for multiple threads to dequeue items, and would increase the rate at which packets can be captured.

References

[1] a. pcap_breakloop(3pcap) man page, 2024. Accessed: 25/11/2024. URL: https://www.tcpdump.org/manpages/pcap_breakloop.3pcap.html.