Overview

This program uses a producer-consumer threading model with a thread pool to allow packet analysis even under high traffic. Packet capture is performed by the *sniff()* function via *pcap_loop()* whilst analysis is done by separate worker threads.

The *dispatch()* function acts as the producer in this model. Each captured packet is safely copied and enqueued into a fixed-size shared queue implemented in *queue.c*. Worker threads continuously dequeue packets (assuming the queue is not empty) and perform analysis via calls to the *analyse( )* function in *analysis.c*.

The *analyse()* function performs multiple detection checks on each packet, including identification of SYN floods, ARP cache poisoning attempts, and access to blacklisted URLs. To ensure thread safety, all shared counters and dynamically allocated data structures are protected using mutexes, so that two threads don't update the data at the same time

Shutdown is handled gracefully: when a SIGINT (Ctrl+C) is received, packet capture is stopped using *pcap_breakloop()*, the queue is emptied fully by worker threads, and resources are cleaned up safely. This ensures that no captured packets are lost and all analysis statistics are accurate.

Efficiency

Various efficiency improvements were made such as splitting packet capture from analysis using a multi-threaded producer-consumer design. Packet capture is lightweight by design as it enqueues packets immediately and there is a lack of expensive processing in the callback function dispatch, reducing the likelihood of packet drops under high traffic rates.

Multiple worker threads perform analysis in parallel, allowing the workload to scale with available CPU cores. Mutex locking is used to minimise contention on shared intrusion detection statistics, particularly for high-frequency SYN flood detection, ensuring that intensive attacks do not reduce overall system performance. Blocking condition variables further improve efficiency by preventing busy-waiting and unnecessary CPU usage. This is further touched upon in "Synchronisation".

Synchronisation

Synchronisation is achieved and maintained via the use of mutex (*pthread_mutex*_t) and condition variables (*pthread_cond_t*) to protect access to shared resources.

The packet queue is protected by a mutex, with the two condition variables (*not_empty* and *not_full*) used to coordinate producers and consumers. This ensures that when the queue is full the capture threads are blocked and the workers threads block when the queue is empty. This avoids CPU overhead and threads repeatedly looping to check these conditions.

In *analysis.c*, all shared counters and resources used for intrusion detection (such as SYN packet counters, unique SYN source IPs and ARP response counts) are protected by mutexes to ensure thread safety.

A dedicated mutex is used for SYN-related activity because SYN flood attacks generate a very high packet rate. Using a separate lock prevented frequent updates from blocking access to the other unrelated statistics, improving overall concurrency and reducing lock contention during attacks.

Signal handling

The program includes a SIGINT handler to allow a safe and controlled shutdown when the user presses Ctrl+C

When the signal is received:

Packet capture is stopped via *pcap_breakloop()*. This prevents any further packets from being enqueued.

Workers threads continue processing any packets present before exiting. This avoids any captured packets being lost during shutdown. Once analysis is complete, a final intrusion detection report is printed using *analysis_cleanup()*.

Dynamically allocated memory used to store SYN source IP addresses and other shared resources is freed before program termination. The signal handler itself avoids performing any unsafe complex operations and instead triggers the normal shutdown path, so it is “async-signal-safe”. (see reference). This ensure that resources are cleaned up correctly and memory leaks are avoided on shutdown.

Memory safety

Memory safety is maintained by carefully managing when memory is allocated and freed and also by clearly defining which part of the program is responsible for freeing memory. Each captured packet is copied into newly allocated memory before being placed into the shared queue, ensuring that worker threads do not access libpcap’s internal buffers after they are reused. Worker threads free packet memory immediately after analysis, preventing memory leaks during long-running captures. Shared data

structures are only accessed while holding the appropriate mutexes, preventing data races and undefined behaviour.

Additionally, the signal handler avoids performing memory allocation or deallocation directly, preventing memory corruption during shutdown.

Additional memory safety is provided through reusable packet validation helper functions that verify packet length before accessing protocol headers or payload data. These checks prevent out-of-bounds memory access when handling truncated or malformed packets and ensure consistent parsing across all analysis functions.

Testing solution

SYN flooding:

SYN flood detection was tested on the loopback interface with the command "*hping3 -c 10000 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost*"

The command flooded the server on port 80 with 10000 SYN packets from randomised source addresses with each packet containing 120 bytes of data.

The result is shown below where the program correctly identified all 10,000 packets and recognises each IP source was different. It was noted that on high packet attempts, the random IP sources could generate the same IP due to the volume.

```
/root/workspace/skeleton/build/idsniff invoked. Settings:
        Interface: lo
        Verbose: 1
SUCCESS! Opened lo for capture
^C
Stopping capture...

Intrusion Detection Report:
10000 SYN packets detected from 10000 different IPs (syn attack)
0 ARP responses (cache poisoning)
0 URL Blacklist violations (0 google and 0 facebook)
```

ARP poisoning:

ARP poisoning detection was tested on the loopback interface with the python file provided *arp-poison.py*. The program originally detected two responses as it was flagging all ARP packets, not only responses.

Upon fixing these mistakes and running the python script, the program correctly detects only one ARP response.

Another ARP poisoning tool was created in test which sends 10 ARP replies per second over five seconds. The program detects all these and reports it afterwards (note verbose is turned on).

```
operation = 2        # 2 specifies ARP Reply
victim = '127.0.0.1' # We're poisoning our own cache for this demonstration
spoof = '192.168.1.1' # We are trying to poison the entry for this IP
mac = 'de:ad:be:ef:ca:fe' # Silly mac address

arp_reply = Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(
    op=2,
    psrc=spoof,
    pdst=victim,
    hwsrc=mac
)


print("Flooding ARP replies, can stop with ctrl-c")

for i in range(50):
    sendp(arp_reply, iface="lo", verbose=False)
    time.sleep(0.1)  # 10 ARP replies per second
print("finished")
```

```
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
ARP response detected: 192.168.1.1 -> 127.0.0.1
^C
Stopping capture...

Intrusion Detection Report:
0 SYN packets detected from 0 different IPs (syn attack)
50 ARP responses (cache poisoning)
0 URL Blacklist violations (0 google and 0 facebook)
```

Blacklisted Urls:

Blacklisted URL detection was tested on the eth0 interface using HTTP traffic generated with "*wget --no-hsts*". Web requests were made over TCP port 80 to predefined blacklisted domains Google and Facebook. The system successfully extracted HTTP Host headers from packet payloads and reported violations with the correct source and destination IP addresses.

This test validated correct payload parsing, protocol filtering, and safe handling of application-layer data within the multi-threaded analysis framework.

Signal handling:

Signal handling tests verified that Ctrl-C consistently produced an accurate and clear full intrusion detection report, was async-signal-safe and cleaned up resources during program termination (such as joining threads and freeing variables.)

Limitations

The program successfully detects the specified common attack patterns but there are known limitations which are out of the coursework specification. SYN flood detection is based on counting TCP packets with the SYN flag set and ACK flag unset. As such, the system cannot distinguish between legitimate workloads or malicious floods. To fix this, full connection tracking would need to be used. ARP poisoning simply flags all ARP response packets and ignores more sophisticated analysis. Blacklisted URL detection only applies to unencrypted HTTP traffic on port 80 and cannot inspect HTTPS traffic. In the future, the program could be extended to properly deal with these attacks

References:

*https://www.tcpdump.org/manpages/pcap_loop.3pcap.*html - used to understand how pcap_loop works and what parameters it should be fed

https://en.wikipedia.org/wiki/Spurious_wakeup - used very briefly and led to WHILE statements being used for queue full/empty condition checking rather than IF statements

https://man7.org/linux/man-pages/man7/signal-safety.7.html - used for async-signal-safe when closing