Introduction

This report outlines the design, implementation and testing processes for creating a packet sniffer to detect intrusion attempts, particularly Address Resolution Protocol (ARP) Cache Poisoning, Xmas Tree Scans and Blacklist Violation attempts, as described by the coursework. Interception of the packages is implemented in sniff.c. The solution uses a set number of threads to carry out the task in an efficient manner. Thread management and assignment of packets to threads is implemented in dispatch.c. Actual processing of the packet headers and contained data is implemented in analysis.c.

Design

In order to obtain the information required for intrusion detection from the received packets, they have to be parsed, separating the TCP/IP model layer headers (network, internet and transport) and the payloads. Therefore, the Ethernet header information is used first in order to detect the type of packet contained as a payload (either ARP or IP). To begin with, an ARP Cache Poisoning attack is a situation in which an attacker sends an ARP response containing a fake MAC address to a victim, pretending to be a trusted device. Therefore, in the case of ARP packets, the Ethernet header has to be removed in order to use the ARP header information to determine whether or not the packet is a response, in which case it must be considered as an ARP Cache Poisoning threat.

Otherwise, the packet is stripped of its Ethernet and IP headers to obtain the TCP packet. An Xmas Tree Scan is an attack in which a TCP packet with the FIN, PSH and URG flags set is sent to a victim in order to capture information about the receiver’s system. Hence, the header of this packet must be observed in order to determine if it should be considered an Xmas Tree Scan. Furthermore, the TCP header of any outgoing HTTP packet must be stripped in order to obtain the final payload of the packet. The destination of this packet must be checked against blacklisted domains and traffic to such domains (in this case '*www.bbc.co.uk*') must be registered.

For efficient processing of packets, a limited-size thread pool is created and each thread is given the task of carrying out the analysis of packet contents.

Implementation

The `analyse()` method in `analysis.c` makes use of '`netinet/if_ether.h`', '`netinet/ip.h`' and '`netinet/tcp.h`' in order to access header data for each TCP/IP layer. The process of parsing the header information and stripping the header is very similar for each layer and packet type. It is done by increasing the location of the pointer pointing to the start of the currently analysed packet by the number of bytes occupied by the header, such that it points to the beginning of the payload (at the current layer). Additionally, all header fields are converted to the host byte order by using `ntohs()` to prevent any misinterpretation of data.

The initial packet data is accessed using a pointer to a character buffer, which allows it to be converted and stored as an `ether_header` structure in order to parse the header information. The information required from this header is the type of the contained packet, found in the `ether_type` field. The value of this field can have many values, of which only `ETHERTYPE_ARP` and `ETHERTYPE_IP` are useful, characterising packets as ARP and IP packets, respectively. Next, the packet is stripped of its header in both cases, as explained above. As was done with the Ethernet (network layer) packet, the internet packet is converted and stored as an `ether_arp` or an `ip` structure. The ARP packet then is checked for its operation type, found in the `ar_op` field of its `ea_hdr` field (so at `ea_hdr.arp_op`). An operation of type ‘2’ identifies an ARP response, which increases the `arp_count` threat number for later intrusion reporting. Otherwise, the `ip_hl` field of the IP packet is extracted to identify the IP header length. This value represents the header size in terms of 32-bit words, so it is multiplied by 4 to obtain the number of bytes. This size is used to strip the IP header, as explained above, and obtain the TCP packet. This data is converted and stored as a `tcphdr` structure in order to obtain the header size as 32-bit words, which is then converted to bytes as before by multiplying this value by 4. While still observing the TCP packet, the `fin`, `psh` and `urg` flags from the header are checked. If they are all set, the current packet is flagged as an Xmas Tree Scan threat by incrementing the `xmas_count`. Furthermore, if the packet socket destination (stored in the `dest`) is ‘80’ (HTTP port), the TCP header is stripped and the packet is searched for data identifying the destination host as ‘*www.bbc.co.uk*’. This procedure is done via the `strstr()` which returns a non-NULL pointer if the search value (`Host: www.bbc.co.uk`) is in the final packet payload. If this is the case, the destination host is the dangerous domain, and the `blacklist_count` counter is incremented to account for the blacklist violation.

For efficient packet analysis, packets are handed to parallel threads in `dispatch.c`. Moreover, to prevent an excessive number of threads from running, a set number of threads (`MAX_THREAD_NUM`) are created. This is done by calling `thread_create()`in `sniff.c`. The global `packet_queue` structure (defined in `dispatch.h`) is also initialised by this method. This structure is used to pass packets to threads: `dispatch()` enqueues `packet_queue_element` structures (also defined in `dispatch.h`) containing packet data, then signals the threads to process this queue element by calling `pthread_cond_signal(&added)`. Next, a thread (which either just finished processing a packet or was waiting for the signal) dequeues the element and calls `analyse()` on the extracted data. After the analysis, the thread frees any redundant memory, and then checks for any elements in the queue or starts waiting again by calling `pthread_cond_wait()`.

In order to provide thread safety, the queue and threat counter shared variables are locked using `pthread_mutex_t` locks. In order to prevent permanently locking these, `pthread_mutex_trylock()` were used instead of the more dangerous `pthread_mutex_lock()`. This method will return 0 if the lock was free and the operation was successful. Otherwise, it will return a non-zero value. Thus any thread will wait until the locking is successful through the `while(pthread_mutex_trylock(&muxlock))` operation. Further, because the shared variable values are accessed and modified frequently, they are declared as `volatile`. As a final precaution, to prevent any data to be overwritten by `sniff()`, the paket data buffer is copied using `memcpy()` to a new memory location and a new `pcap_pkthdr` is declared for each intercepted packet.

In order to terminate the packet analysis process, `sig_handler` cancels all threads and frees any memory allocated for queue elements and the queue if the `SIGINT` signal is received. This signal handler is initialized in `sniff.c`. All of the threads are initialised as cancellable and are set to not be cancellable once they start processing information to ensure that the queue elements that are removed from the queue are freed by the threads processing them. If a thread is not cancellable when the cancel command is issued, this command is queued and processed by the thread once it is returned to a cancellable state. This ensures proper clean-up of threads and memory upon termination.

Testing

Initially, to check if the packet headers were correctly removed, files were requested through HTTP, the requests as well as files were stripped of their headers, and their payloads were outputted to the terminal. The easiest packets to analyse were the HTTP requests, which had all of the expected fields intact. The conclusion was that headers were being correctly removed.

In order to confirm correctness of the solution, several situations were tested against. For correct detection of ARP responses, the provided `arp-poison.py` script was run multiple times while the solution was running and listening on the loopback interface. The report displayed upon termination confirmed that all received ARP responses were registered.

Next, to verify the detection of Xmas Tree Scans, `nmap -sX localhost` was run on the virtual machine while the solution was running and listening on the loopback interface. The correct number (1003) of suspicious packets was registered for each time the command was run.

Then, to ensure the correct detection of blacklist violations, webpages were requested using the `wget` command on the virtual machine while the solution was running and listening on the first ethernet interface (`eth0`). All HTTP requests to '*www.bbc.co.uk*' were registered (while any other requests were not) and the report displayed the correct number of suspicious requests.

In conclusion, all threats were being correctly registered and reported, demonstrating proper packet parsing and intrusion detection.

Finally, correct functioning of the threads was tested by subjecting the virtual machine to high loads of traffic for extended periods of time while the solution was running. No deadlocks were encountered, all packets were processed and any threats were properly registered. The conclusion drawn from these tests was that the threads were functioning properly in a concurrent manner.

As a result of the outlined process, an efficient, correct solution for the specification outlined by the coursework was created.