CS241 - Packet Sniffer

Connor Barr - U1530685

22/11/2016

Design & Implementation

Introduction

The aim of this project was to produce a program that could sniff packets, analyse them and detect potentially malicious packets. The process for this involved three major steps. First off, to be able to detect if a packet was malicious the data held within the packet would have to be parsed. The next step would be to determine if the packets were malicious and return a report if so. Finally, the last step would be to introduce threading to the programming to aid with heavy loads of packets. In this case the threading method I chose to use was a thread pool combined with a work queue implemented by a linked list structure.

Part One

In each packet there are several headers, the type of which is defined by the protocol used by the packet. In this case each packet first contained an ethernet header, of length 14 bytes, that contained a variable to specify the protocol of the packet (ethernet type). If the packet was of type 0x0800 (IPv4) then the packet used TCP/IP, and if the packet was of type 0x0806 (ARP) then the packet contained an ARP header. To investigate each header a pointer was created at the beginning of the appropriate header and this was used to initialize a structure for each. Each structure contained key data stored within the header. In the case of the IP header (first following Ethernet in IPv4) it contained a source and destination address. In the case of the TCP header it contained the three flags involved in an 'Xmas Scan' packet (All 3 set to 1 indicates an 'Xmas Scan'). Finally, an ARP packet contained an operation specifier that indicated if the packet was an ARP response (ARP Operation type is two in an ARP response).

Part Two

After completing part one the program now had a method to detect whether a packet was malicious or not. Detecting whether a packet was malicious alone was useless, to implement an intrusion report system the program makes use of signal catching. In the case that the user inputs Ctrl + C while the program is running the signal sent to the kernel is caught and outputs a report of all the intrusions detected. In order to count the amount of intrusions detected and each type the program makes use of global variables within the analysis file. Each time one is detected the appropriate variable is incremented by one and each is printed in the intrusion report when requested. To detect if a packet was an HTTP request to a blacklisted URL the packet would first have to come from a host port of 80. If a packet was detected to have a source port of 80 then the payload of the packet was scanned in the case it contained the string "Host: -blacklisted url-" using a string contains function, if this was the case then the packet was an http request to a blacklisted url.

Part Three

The final part of the program was to introduce threading to the packet analysis. The method I chose to use in this case was a thread pool method instead of the Apache standard method. The thread pool method involved creating a persistent set of threads outside of the dispatch method, a long with a linked list structure to hold the global queue of work for the threads. When the dispatch method is called in the sniff file it takes the packet data, copies it into a packet structure and adds it to the queue (head if empty, tail otherwise). The first implementation of the thread pool

caused issues with the data contained within the headers, to prevent this issue the packet header and packet were copied using 'memcpy()' before being added to the queue. Upon the first call of the dispatch method the threads are created and the queue is initialized before any packets are added to the queue. The thread function then runs in a while-true loop to make them persistent, in which the threads continually check the head of the queue for any potential packets to be analysed before analysing them.

Testing

In order to test if the program was functioning correctly it makes use of a 'verbose' flag. When the flag is set using '../build/idsniff -v' the program prints out all data parsed from the different headers. The verbose flag aided in debugging the program to ensure the correct data was being analysed. To test if the program was correctly detected the different potential malicious packets there were three different primary tests. Firstly, for the 'Xmas Scan' packets, when the program was running, the 'nmap -sX localhost' command could be used to send requests from and to the local machine containing several 'Xmas Scan' packets. Secondly, for the ARP cache poisoning the provided python script sent an ARP response to the local machine, this could be used to ensure that the program was correctly detecting the ARP response packets. And finally, a 'wget' command was used to send an http request to 'www.bbc.co.uk/news' to test if the program could correctly detect any blacklisted url requests. All three of these tests returned positive results. To test the final part of the program, threading, when a thread removed a packet from the queue it would print its thread identifier, to ensure that all and only the threads from the pool were being used.

Conclusion

The final version of the program was one that made use of multi-threading to analyse incoming packets and detect if they were 'Xmas Scan', ARP cache poisoning or blacklisted URL requests and print a report accordingly. Testing the program and tweaking it to correctly parse data was laborious but rewarding, from it I learnt a lot about packet structure, data parsing and the C programming language.

References

1. Santa Clara University - Computer Engineering (no date) Available at: http://www.cse.scu.edu/ (Accessed: 9-19 November 2016).

2. University of Warwick - Department of Computer Science - CS241 - Lab 2 (no date) Available at: http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/labs/lab2/ (Accessed: 9-19 November 2016).

3. University of Warwick - Department of Computer Science - CS241 - Network Primer (no date) Available at: http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/coursework16-17/network-primer (Accessed 19 November 2016).

4. The Geek Stuff - Linux Signals (No Date) Available at: http://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/ (Accessed: 17-19 November 2016).