Christopher Dixon

U91140733

Computer Networking 4004

28 November 2018

Port Knocking Implementation

I. Introduction

This is a port knocking program that enables a web server on command depending if a client sends the correct sequence of packets to the ports, also known as port knocking. The IP address and the ports this program listens are:

Port One = 2372

Port Two = 8744

IP address = 127.0.0.1

There are three programs used in this implementation: PortClient.c , PortKnock.c , weblite1.c , PortClient.c inities the knock, PortKnock.c processes the knock and enables the web server, and weblite1.c is the web server being used. Once initiated one can access the server by visiting the site 127.0.0.1:8080/, with the desired file name entered right after.

Also this program uses UDP over TCP, the reason for this is that the only thing the server program cares about which port was the packet sent to and the message in the packet. By using UDP there are no connections/streams which means more throughput of UDP packets for the program.

II. Port Knocking Server Side

For this implementation I used two separate sockets, each listening on a port defined at the start of the program. The program sets the sockets to non-blocking and only stops if the correct number of bytes is sent from the client program, typically to deal with multiple clients the use of threads is common but I wanted to avoid threads since I use global variables which could lead to possible race conditions. Likewise the other typical option is set sockets to non-blocking and make use of the select() function to switch to the socket that is receiving a message, which is similar to my implementation except I don't make use of select instead I drop every packet unless a specific message is sent.

Once a valid knock is recorded the program forks off a child process and executes the server from there; this way the program can execute any web server executable without changes to the web server. Once the server is executed the timer starts and counts the amount of seconds until ten seconds has been reached, which the parent process kills the child process then goes back waiting for a valid port knock again. If a valid port knock is recorded while the server is running then the server uptime is increased by ten seconds for every valid knock, the server uptime resets once the child process / web server goes down.

Every attempted knock is recorded in a struct where the server keeps track of the IP address associated with the knock, stores the sequence of the knock, and checks if its a valid knock. If it is still waiting for another knock then do nothing, if wrong sequence is

noticed then clear all previous knock attempts and start over, otherwise open up the web server or increase web server up time.

A. Messages

The sockets only processes packets with a message size of ten bytes, first byte is the intended sequence of the knock since this program uses UDP we can't know what order the packets will arise. The next eight bytes must be a unique series of characters compared to the previous messages, only thing that matters in the message is the first byte the rest can be anything else but the same sequence cannot be reused; this way replay attacks are avoided since every message/password can only be used once.

B. Denial of Service Prevention

Since this program drops every packet that does not meet the specified requirements then most DoS attempts won't do much unless the rate of packets being sent severely outpaces the rate the CPU processes the incoming packets. Also if the attacker knows the exact specifications of the server then they can big it down by forcing the program to dedicate more time to each packet.

III. Port Knocking Client Side

The client side of the program sends UDP packets to the server program sequentially, with each message being randomly changed. After an attempted knock the

program alters a random byte between characters 4 through 8 in the string, changing the character to a random letter. This way the program is probabilistically sending a unique message that the server has never seen before to avoid replay attacks.

The format of the packet's messages, in hex, should look similar to this:

31, 34, 38, 4A, 3E, 36, 4D, 36

, when converted to decimal they represent the string in ascii. The first byte should always either be 30 or 31 since the first character must either be 1 or 0; the rest could be any other character.