IMPLEMENTING A PORT KNOCKING SYSTEM IN C

An Honors Thesis submitted in partial fulfillment
of the requirements for Honors Studies in
Physics

By

Matt Doyle

2004
Physics
J. William Fulbright College of Arts and Sciences
The University of Arkansas

Acknowledgements

I would like to thank Dr. Craig Thompson of the Department of Computer Science and Computer Engineering, Dr. Gay Stewart of the Physics Department, and Dr. John Stewart of the Physics Department. They have provided me with endless support over the past years, without which my degree and this project would not have been possible.

Table of Contents

Abstract

Modern computer security typically takes a multi-layered approach in which the desired security level for a computer system is reached by combining a number of different security methods. While these methods, taken individually, are often not enough to provide thorough defenses, when implemented together, they can form a significant deterrent against malicious users.

This thesis explores the concept of port knocking, a relatively unknown method for augmenting the security of computer systems. It begins with a very basic primer on host-to-host communication over a computer network, followed by an introduction to the concept of port knocking and a description of the goals of this project. This thesis then describes in detail the primary goal of the project: the design and implementation of a port knocking system in C. Each component of the project is described in detail, with source code examples included for clarity. The thesis concludes with a critical look at the accomplishments and shortcomings of the project in its current state, as well as a thorough discussion of suggested future work.

1.0: Introduction

In times of revolution, people often find themselves confronted with dramatic changes, bold new ideas, and new problems for which they must find solutions. Mankind is currently in the midst of one of the most significant and fascinating revolutions yet: the proliferation of computer technology. There are likely few who would fail to recognize the pervasiveness of computers in our present-day society. Indeed, computer technology has become almost a necessity for maintaining our current way of life. It is this very dependency that holds the greatest threat of computer technology: the sudden lack of it.

Recent times have seen the advent of a number of devastating computer viruses, worms, and exploits. With attacks such as these on the rise, computer security is becoming an area of research vitally important to protecting our way of life. Typically, system security relies on a multi-layered approach, utilizing a number of different security methods simultaneously in order to defend against malicious activity. The decision of how secure to make a system is often a balancing act, for the cost of increased security is often decreased usability. A prime example of this is the firewall [2]**.**

The purpose of a firewall is analogous to that of a bouncer in a club. The firewall, acting on a given set of rules, decides what network traffic is allowed into a system, and what traffic is denied. The most straightforward way to keep out all malicious traffic is easily implemented: just reject all incoming network data. For the club owner, a policy such as this would certainly prevent fights from breaking out inside, but it would also result in a lot of angry patrons outside and not many drinks being sold at the bar. Likewise, firewall rules such as these will make a system highly secure, but completely inaccessible to anyone who is not physically nearby. The most beneficial outcome for the system administrator would be to implement a system that would provide a high level of security, but manage to stay flexible enough to allow desired traffic through when needed. This is where the concept of port knocking becomes relevant.

1.1: Basics of Network Communication

In order to accurately describe the concept of port knocking, some groundwork must first be laid. Communication between two networked computers requires, at the very least, two very basic pieces of information: an IP address and a port number. An IP address is a value similar to a postal address. It tells where a specific computer can be found on the internet. With an IP address, we can be sure that information will reach the machine for which it is destined. Once there, the remote machine must determine where the data must go, that is, which application it is intended for. It does this with the second value, the port number. Going back to the postal analogy, port numbers can be compared to the “care of” field in a postal address. The port number indicates which application should process the data that has been sent. Many common applications have a standard port number with which they communicate. Webservers typically use port 80 (HTTP). Secure Shell sessions use port 22 (SSH). When programs such as these are running on a remote machine, these ports that they use are “open,” meaning that they are ready to begin accepting data. Ports which do not have a program currently associated with them are “closed.”

Firewall software regulates inbound and outbound network traffic based on these port numbers. Network data, upon arrival at its destination machine, must first pass through the firewall (remember the bouncer analogy) before it is sent on to its destination port. In this manner, even if a Secure Shell server is active and awaiting data on port 22, if the firewall is configured to deny all traffic on port 22, no communication will be possible with this machine via Secure Shell and port 22 will appear to be closed.

At first, closing off ports may seem undesirable, but every program listening on an open port can potentially pose a security threat to the system. Most of the time, the vulnerabilities that hackers exploit are due to programs running on open ports [8]. By sending these programs "bad data," hackers can cause a range of problems to occur on these systems, ranging from program crashes to arbitrary code execution [8]. Hence, the open ports on a system are only as secure as the programs using those ports. Port knocking provides a simple yet effective solution to this problem. It enables the computer to pass traffic to these ports when needed and deny it the rest of the time, all the while keeping the entire process secure against those who might be watching.

1.2: Port Knocking

During the Prohibition Era of the 20th century, it was not uncommon for alcohol to be provided within the confines of highly secretive speakeasies. Owners of such establishments would often enforce a strict set of rules at the door. On one hand, they would want to let in patrons who would provide them with profits; on the other hand, they also wanted to keep out law enforcement officers who sought to shut down their operations. In some instances, policy such as this was implemented by means of a "secret knock." If a patron knocked on the door in a certain way, they would be let in, for it was assumed that if they knew the secret knock, they were someone who could be trusted. Port knocking works in much the same way, only the "secret knock" is actually a series of connection attempts to closed ports, and the door is actually a specific port that is to be opened.

At this point, the concept of port knocking might seem like a contradiction in terms. The goal is to design a system that allows authentication with a remote machine by way of communication across a series of ports, but these ports are closed, preventing all communication. This is exactly the problem for which port knocking provides such an interesting solution. Many systems that use a firewall include options for logging connection attempts to closed ports [3]. By enabling these options, it becomes possible to communicate with the server by way of log entries. It is not the log entry that is interesting, but the information it contains. From these entries, the server can retrieve the IP address of the machine initiating the connection attempt, and the port it is trying to connect to. If a sequence of port numbers can be constructed to encode some desired information, it will be possible to send that information to a machine which initially may not even appear to exist on the network. Furthermore, if a server can be configured in such a way that it will interpret this information as a command to open a specific port, then a basic port knocking system will be in place.

2.0: Goals

The primary goal for this research project has been to design a port knocking system written in C. The most readily available example of the potential of port knocking is Martin Krzywinski's `knockclient` and `knockdaemon`. Krzywinski's software is presented as a proof-of-concept design, written in Perl [1]. While Perl is clearly superior to C in terms of its text parsing capabilities and concise code, the goal is to design a compiled solution, as opposed to interpreted, that can eventually become an efficient client/server port knocking system.

The secondary goal for this project is to implement port knocking in such a way as to be secure in the event of malicious users monitoring network traffic. In other words, the data that is being communicated to the server should be encrypted within the sequence of port numbers itself. With such encryption in place, even if someone manages to observe the knock sequence and recreate it, they will still only be able to open the chosen port (a value which will be unknown to them) and it will only be open to traffic originating from a chosen IP address (also unknown to them).

3.0: Implementation

The design of the client and server components of this project was done in a fairly platform-specific way. The client code, `knockc.c`, was primarily developed and tested on a machine running Windows 2000 Professional under the Cygwin Linux emulation environment. This code was compiled and tested under both Cygwin Linux and FreeBSD 4.8. The server code, `knockd.c`, was developed on a FreeBSD 4.8 machine running the firewall package `ipfilter` and was designed with this platform in mind. Although at this time `knockd` has only been tested on FreeBSD 4.8, it was designed in such as way as to ease future work in making the code platform-independent, a topic addressed in **Future Work**.

3.1: The Client, `knockc`

The first key component needed to design an effective port knocking system is the client program, the part that will do the knocking. The client designed for this project, `knockc`, begins by prompting the user for a source host, a destination host, a port number, an offset, an action value, and an encryption password. Host names can be entered either in the form of a domain name (i.e. domain.name.com) or an IP address (i.e. 130.184.92.77), as they will ultimately be resolved to an IP address. The client, upon resolving the source hostname, is left with an IP address which is composed of 4 1-byte values, each with a possible range of 0 – 255. The port number, on the other hand, has a possible range of 0 – 65,535, which makes it a 2-byte value. By breaking this value up into its upper 8 bits and lower 8 bits, it can be represented by two 1-byte values. Finally, the action value tells the server whether the requested port should be opened or closed.

The data now consists of 7 bytes, which can then be mapped into characters, as there are 256 total characters represented in the extended ASCII character table (see Appendix C). This resulting 7-byte character string is then input into an OpenSSL library function `BF_ecb_encrypt( )`, which implements the Blowfish encryption algorithm [7]. `BF_ecb_encrypt( )` then takes the 7-byte character string, which has been padded with null characters to 8 bytes (as the Blowfish algorithm must operate on 8-byte chunks of data), and produces an encrypted 8-byte character string representing the input information. The client then takes this encrypted string, and by reversing the conversion process used above, stores each of the 8 characters of this string as an integer value with range 0 – 255. Finally, the offset is added to each of these values, making the ports fall on a specific range of 256 ports to which the server will be paying special attention. Once this sequence of port numbers has been constructed, the client performs the final task of making connection attempts to each of these ports on the remote host.

3.2: The Server, knockd

The second key component of a viable port knocking system is the server, the program which receives the knocks from the client and interprets them as commands to open ports in the firewall, execute system calls, or take other actions. The goal of the server, `knockd`, is for it to successfully perform four primary tasks: log monitoring, retrieval and storage of port knocks, decryption of completed knock sequences, and firewall modification.

The first task, log file monitoring, ties back to the firewall option mentioned earlier. For a port knocking solution to be effective, the system must be set up to log all connection attempts to closed ports, specifically, the range of 256 ports which have been designated to receive knocks. Log entries such as this on the FreeBSD 4.8 operating system will appear as:

```
Apr 13 03:21:08 planb /kernel: Connection attempt to TCP 192.168.1.1:90
from 192.168.1.2:3120
Apr 13 03:21:09 planb /kernel: Connection attempt to TCP 192.168.1.1:87
from 192.168.1.2:3123
```

In the sample entries above, it is indicated that the client located at IP address 192.168.1.2 attempted to connect to ports 90 and 87 on the server located at IP address 192.168.1.1.

By creating a regular expression to match against these types of log entries ..

```
^.+Connection attempt to TCP [0-9.]+:([0-9]+) from ([0-9.]+):[0-9]+
```

... the server will be able to separate these entries from the log file. But first, there must be some selectivity about which of these entries will be processed. Older log entries cannot be reevaluated every time the log file is checked. Only those entries which have been recently appended to the log file should be processed.

`knockd` begins by calculating the initial MD5 hash of the log file with the function `MD5File( )`. MD5 is an iterative one-way hash function that takes an arbitrary length string and calculates a 128-bit hash value for that data [6]. In other words, MD5 generates a unique digital “fingerprint” based on the data currently stored in the log file. `knockd` then sets the log file pointer to the end of the file, which will tell it where to begin reading once log entries have been appended to the log file. By recalculating the MD5 hash at regular intervals, it is possible to detect when entries have been appended to the log file, and promptly search this new data for entries matching the regular expression.

The use of regular expressions allows the storage of portions of a matched string. This is done by way of the parenthesis operators. In the regular expression above, the substrings that are being stored are the IP address from which the connection attempt originated, `([0-9.]+)`, and the port which that machine was attempting to connect to, `([0-9]+)`. Now that these pieces of data can be retrieved on the server side, there must be a way to store them until they are ready for processing.

The method used for storing this incoming data was a linked list of simple structures. These structures, designated `struct hostdata`, consist of the following fields:

```
struct hostdata{
          char originIP[16];
          unsigned char blf_in[MAXNUMOFKNOCKS];
          unsigned char blf_out[MAXNUMOFKNOCKS];
          unsigned short int knocks_enc[MAXNUMOFKNOCKS];
          unsigned short int knocks_dec[MAXNUMOFKNOCKS];
          int knocksreceived;
          struct hostdata *next;
          time_t timestamp;
 };
```

The IP address which a particular set of knocks originated from is stored in the field `originIP[ ]`. The encoded knock sequence is stored in the array `knocks_enc[ ]`, and is indexed by the integer value `knocksreceived`. This value is used to add the port numbers to the array as they are received, and is used to detect when a knock sequence has been completed.

Once a complete knock sequence has been received, it is ready to be decrypted. This is done by first subtracting the offset from each port value in order to produce the original sequence of values generated by the client. Next, each integer port value is converted into its corresponding ASCII character. The 8-byte string formed by these characters, `blf_in[ ]`, is then decrypted with the same function used for encrypting, `BF_ecb_encrypt( )`, and the same password used for encrypting on the client side (the user is prompted for this password when `knockd` is started). The decrypted string is then stored in the buffer `blf_out[ ]`. These decrypted characters are then converted back to integer values and reassembled into the source IP address, requested port number, and action value.

At this point, the port knocking server now has all the information it needs in order to modify the firewall rules. This is done with the function `modify_ipfilter_firewall( )`, which utilizes the fact that the program being used to create the firewall, `ipf`, can take as input multiple rule files when it creates a firewall. This allows the existing firewall rule file to remain untouched, and an auxiliary rule file, `ipf.rules.knockd`, to store the firewall rules that are generated by `knockd`. If the client system is requesting that the server open a port for access, the server scans the auxiliary log file to see if such a rule already exists. The server then adds this rule if it doesn't exist, and ignores the request if it does. If the client is requesting to close a port, the server looks for the rule opening this port, and deletes it. The server ignores the request if such a rule is not found. Once this auxiliary file has been successfully modified, the server then executes a system call:

```
ipf -Fa -f ./ipf.rules.knockd -f /etc/ipf.rules
```

This command rebuilds the firewall based on the existing firewall rules and the auxiliary firewall rule file. Once the server has completed this task, it frees the memory allocated to storing the data from the client, and then continues monitoring the log file for changes.

4.0: Analysis

The programs `knockc` and `knockd`, in their current state, provide a simple yet effective port knocking solution. `knockc` allows the user to construct an 8-byte port knock sequence once the requisite information has been supplied. This enables an authenticated user to modify the firewall of a remote machine in order to access services on that machine that were hidden to them. `knockd` enables a system administrator to hide running services, only allowing selective access to those services by way of the client, `knockc`. Overall, this port knocking project holds great potential for becoming another layer of security to defend critical systems.

Some argue that port knocking takes a “security through obscurity” approach to protect computer systems [1, 4, 5]. Security through obscurity prevents access by hiding something in a way that makes discovery unlikely, but not impossible. An example of this would be storing money in a shoebox, and then burying the shoebox on a huge plot of land. Opponents of security through obscurity argue that the security of the money only depends on the thief's willingness to find it, which is true. The more desirable option for implementing security would be to place the money in a safe, place the safe out in the open for anyone to access, and making the details of its inner workings known to all. If the safe was so secure that thieves, armed with this knowledge, could not crack it, then it could be considered truly secure.

The main concern brought up by critics of port knocking is that of replay attacks [4, 5]. That is, if a malicious user eavesdrops on a knock sequence, it is trivial for them to reproduce the sequence, thereby reproducing the actions invoked on the server. `knockd` addresses this problem in two ways.

First, the information which is transmitted to the server is encrypted within the sequence of knocks itself. This information is encrypted and decrypted with a password known only to the client and server. Malicious users are therefore unable to construct their own knock sequences for the server, because without the proper password, their encrypted data will only come out as garbage when decrypted by the server.

Second, `knockd` is designed to only allow access to the machine whose IP address is encrypted in the knock sequence. This is an important feature, as the knock sequence generated by the client is easily reproducible. However, `knockd` is implemented in a way that defeats such "replay attacks." Even if a malicious user were to replay the knock sequence, they would only be causing the server to react in the way intended by the original user. That is, the server would only be opening a port to traffic from an IP address chosen by the original user. Since these values are encrypted within the knock sequence, the malicious user has no way of knowing the nature of the access being granted, and their replayed knock will appear to not have any effect at all.

Another issue that a reliable port knocking solution must address is Denial-of-Service (DoS) attacks. With `knockd`, the most likely target of a DoS attack would be the system memory. When `knockd` receives a port knock from a new host, it allocates a structure of type `struct hostdata` in order to store information pertinent to that specific host. There exists the possibility that a malicious user could flood the server with fabricated port knocks, so that each new knock appears to originate from a new host. The eventual result of such an attack would be the exhaustion of system memory, and most likely system failure.

`knockd` is implemented in such a way as to largely mitigate the effects of this type of attack. When a structure is allocated for a new host, it is given a timestamp using the variable `time_t timestamp`. This timestamp is then periodically checked (via the function `check_timestamps( )`) to see if a predetermined amount of time has elapsed without that host completing a full knock sequence. If this structure has expired, the memory allocated to it is freed and the program continues as normal. It is likely still possible that with a large enough timeout value and a powerful enough DoS attack, the system memory could still be exhausted. However, the current method seems sufficient for the scope of this project. More sophisticated methods for warding off such attacks will be investigated in future versions of this software.

One final issue that should be addressed concerning any port knocking implementation is that of log file pollution. Since a single knock sequence causes 8 connection attempt entries to be appended to the system logs, regular use of port knocking can defeat the purpose of log files by bloating them with useless information. There are a few possible solutions to this problem which will be discussed in the following section.

5.0: Future Work

The code developed for this project in the files `knockc.c` and `knockd.c`, while providing a working port knocking solution, leaves much room for future expansion and enhancement. The first and likely most obvious topic for future work is that of platform independence. The client program has been successfully compiled and tested on both Cygwin Linux and FreeBSD 4.8. The majority of future work done on `knockc` will most likely involve testing on a range of UNIX/Linux platforms, and modifying the code so that it works correctly on all these platforms. In addition, `knockc` will be modified to accept all necessary values as command-line arguments, in addition to its existing interactive user-input mode.

Other work to consider would be designing a version of this program for the WindowsNT/2000/XP operating system. This would probably require a significant amount of modification to the network code, as well as the design of a graphical user interface (GUI). Finally, the most interesting possibility for this project would be the design of a web-based port knocking client, most likely written in PHP. Such a client would be more flexible than a command-line or GUI based client, as it would be operating system independent and simpler to access and use.

Platform independence for `knockd` will require much more work than that of the client, as the server uses more specialized libraries and, in places, has a very system-specific design. First among considerations is the use of the libraries `std.h` and `daemon.h`. These are third-party function libraries that come as part of the package `libslack`. The porting of `knockd` to other platforms would require either those platforms to support `libslack`, or the writing of custom code that would give `knockd` functionality equivalent to that offered by `libslack`.

In addition, `knockd` needs to be modified to support a number of different firewalls. Fortunately, the design of `knockd` was done with this future expansion in mind. Currently, `knockd` is only capable of modifying firewalls implemented with the `ipfilter` software package. It does this via the function `modify_ipfilter_firewall( )`. It would not be difficult to implement functions that could modify firewalls based on other software packages, and allow the user to specify which function `knockd` should use.

One particularly interesting concept which arose out of the design of this project is that of granting access to a remote third party. Currently, `knockc` and `knockd` do not distinguish between the client IP address extracted from the log entries and the IP address decrypted from the port knock sequence. `knockc` is designed in such a way that it allows the user to specify which IP address `knockd` will be granting access to. Thus, an authenticated user can grant access to whomever they choose. There are a number of reasons for and against this capability, and so `knockd` will eventually allow the administrator to specify whether or not this behavior will be allowed.

Another interesting possibility for future work with port knocking is the expansion of what the server can do when evaluating a knock sequence. Currently, the port knocking project is built around the idea that when a valid knock sequence is received, a port is either opened or closed in the firewall. This functionality could be expanded to associate a predefined knock sequence with an arbitrarily defined system action. One possible way to do this would be to use values in the source IP address which are beyond the value 255. We could then implement a user-written rule file which the server would check when evaluating a knock sequence. If it detects values that are invalid in the source IP address, it could be designed to check this rule file for a matching sequence, and then execute the command that the user has associated with that knock sequence.

Solving the problem of log file pollution is another area in which `knockd` has much room for growth. Possible solutions to this would most likely require a modification of the firewall, and not the port knock server itself. The most appealing solution would be to modify the firewall logging in such a way that it would log connection attempts that fall within the chosen 256 port range to a separate log file, and then modifying the port knock server to only monitor that special log file. Other possibilities might involve the removal of connection attempt entries from the log file once they have been processed by `knockd`.

Finally, authentication is an area that needs further development. A more robust password management system would be a significant step towards the use of `knockd` in a multi-user environment. Currently, `knockc` and `knockd` both rely on a shared password used for encryption and decryption. While this works, it does not afford much flexibility. With a multi-user, multi-password system in place, a system administrator could define which ports a given password is allowed to open. A system such as this is certainly feasible, but will likely require substantial development.

6.0: Conclusion

This project offers a successful demonstration of the potential security benefits to be gained by implementing a fully-functional port knocking system. While `knockc` and `knockd` offer a working solution, they are far from their full potential and still need much enhancement before they can be deployed as a trustworthy security enhancement.

Currently, preparations are under way to release all code written for this project as open source software under the GNU General Public License (GPL). The current plan is to host this code on the website www.sourceforge.net, a popular repository for open source software. The project is registered under the name “Port Knocking,” and it is hoped that the availability of this code will attract other programmers to contribute their ideas and talent.

The work done for this project will hopefully lay a solid foundation for future work towards developing a robust, viable port knocking implementation. The ultimate goal for this project is for it to develop into a viable port knocking solution that administrators can rely on to enhance the security of their systems. While it is still only in the early stages of that development, the potential for such an accomplishment certainly exists in this project, and will remain a driving force in my future work in this subject.

Appendix A: Source Code for **`knockc.c`**

```
//knockc v0.1 by Matt Doyle


// #INCLUDES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <pwd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <openssl/blowfish.h>


// #DEFINES
#define MAXHOSTNAMESIZE 51  // defines maximum hostname string length
#define MAXNUMOFKNOCKS 8  // defines max possible # of knocks
#define MAXPASSWDSIZE 128  // defines maximum password length


// GLOBALS
char password[MAXPASSWDSIZE];


// START MAIN
int main(void){


        // VARIABLES
        int sockFileDesc;  // socket file descriptor
        int count;  // loop counter
        unsigned short int port;  // remote port to open
        unsigned short int upper = 0;  // upper byte of port number
        unsigned short int lower = 0;  // lower byte of port number
        unsigned short int action = 0;  // determines open or close
        unsigned short int knocks_enc[MAXNUMOFKNOCKS];  // encrypted
port knock sequence
        unsigned short int offset = 0;  // port range offset
        char remotehost[MAXHOSTNAMESIZE];  // user-input remote hostname
        char localhost[MAXHOSTNAMESIZE];  // user-input local hostname
        char portstring[6];  // user-input knock port
        char offsetstring[6];  // user-input offset
        char clientIP[16];  // resolved IP address of client
        char serverIP[16];  // resolved IP address of server
        unsigned char blf_in[MAXNUMOFKNOCKS];  // string to be encrypted
        unsigned char blf_out[MAXNUMOFKNOCKS];  // encrypted string
        struct sockaddr_in serverAddress;  // used to define socket
        struct sockaddr_in clientAddress;  // used to define socket
        struct hostent *servername;  // used for server name lookup
        struct hostent *clientname;  // used for server name lookup
        BF_KEY bfkey;
```

```
// CLEAR OUT BUFFERS
memset(remotehost, '\0', sizeof(remotehost));
memset(localhost, '\0', sizeof(localhost));
memset(portstring, '\0', sizeof(portstring));
memset(offsetstring, '\0', sizeof(offsetstring));
memset(blf_in, '0', sizeof(blf_in));
memset(blf_out, '\0', sizeof(blf_out));
memset(clientIP, '\0', sizeof(clientIP));
memset(serverIP, '\0', sizeof(serverIP));
memset(knocks_enc, 0, sizeof(knocks_enc));


// PROMPT USER FOR LOCAL HOSTNAME OR IP
printf("\nEnter local hostname or IP: ");
fgets(localhost, MAXHOSTNAMESIZE, stdin);


// PROMPT USER FOR REMOTE HOSTNAME OR IP
printf("Enter remote hostname or IP: ");
fgets(remotehost, MAXHOSTNAMESIZE, stdin);


// PROMPT USER FOR PORT
printf("Enter remote port to open/close: ");
fgets(portstring, 6, stdin);


// PROMPT USER FOR OFFSET
printf("Enter port range offset: ");
fgets(offsetstring, 6, stdin);


// PROMPT USER FOR PASSWORD
strcpy(password, getpass("Enter encryption password: "));


// PROMPT USER FOR OPEN/CLOSE
printf("Enter (1) to open port, (0) to close: ");
action = getc(stdin);
if((unsigned char) action == '1')
        action = 1;
else action = 0;


// CHECKS FOR NULL STRING INPUT
if((strlen(portstring) == 0) || (portstring[0] == '\n')){
        printf("ERROR: No ports specified\n\n");
        exit(0);
}


// CHECKS FOR ZERO OFFSET
if((strlen(offsetstring) == 0) || (offsetstring[0] == '\n'))
        offset = 0;


// GET RID OF NEWLINES
count = 0;
while(count <= MAXHOSTNAMESIZE){
        if(remotehost[count] == '\n')
                remotehost[count] = '\0';
                if(localhost[count] == '\n')
                        localhost[count] = '\0';
                count++;
        }
        count = 0;
        while(count < 6){
                if(portstring[count] == '\n')
                        portstring[count] = '\0';
                if(offsetstring[count] == '\n')
                        offsetstring[count] = '\0';
                count++;
        }


        // GET THE PORT NUMBER
        port = (unsigned short int) strtol(portstring, NULL, 10);


        // GET THE OFFSET
        if((offset = (unsigned short int) strtol(offsetstring, NULL,
10)) > 65279){
                printf("ERROR: Port offset > 65279\n");
                exit(0);
        }


        // CLIENT AND SERVER NAME LOOKUP
        clientname = gethostbyname(localhost);
        if(clientname == NULL){
                herror("\nERROR");
                exit(0);
        }
        memcpy((char *)&clientAddress.sin_addr.s_addr, clientname-
>h_addr_list[0], clientname->h_length);
        memcpy(clientIP, inet_ntoa(clientAddress.sin_addr), 15);
        printf("\nSource: %s (%s)\n", clientname->h_name, clientIP);

        servername = gethostbyname(remotehost);
        if(servername == NULL){
                herror("\nERROR");
                exit(0);
        }
        memcpy((char *)&serverAddress.sin_addr.s_addr, servername-
>h_addr_list[0], servername->h_length);
        memcpy(serverIP, inet_ntoa(serverAddress.sin_addr), 15);
        printf("Destination: %s (%s)", servername->h_name, serverIP);


        // ENCRYPT THE KNOCK SEQUENCE
        blf_in[0] = (unsigned char) strtol(strtok(clientIP, "."), NULL,
10);
        blf_in[1] = (unsigned char) strtol(strtok('\0', "."), NULL, 10);
        blf_in[2] = (unsigned char) strtol(strtok('\0', "."), NULL, 10);
        blf_in[3] = (unsigned char) strtol(strtok('\0', "."), NULL, 10);
        upper = port;
        upper = upper >> 8;
        lower = port;
        lower = ((lower << 8) >> 8);
        blf_in[4] = (unsigned char) upper;
        blf_in[5] = (unsigned char) lower;
        blf_in[6] = (unsigned char) action;
        BF_set_key(&bfkey, sizeof(password), password);
        BF_ecb_encrypt(blf_in, blf_out, &bfkey, BF_ENCRYPT);

        knocks_enc[0] = ((unsigned short int) blf_out[0]) + offset;
        knocks_enc[1] = ((unsigned short int) blf_out[1]) + offset;
        knocks_enc[2] = ((unsigned short int) blf_out[2]) + offset;
        knocks_enc[3] = ((unsigned short int) blf_out[3]) + offset;
        knocks_enc[4] = ((unsigned short int) blf_out[4]) + offset;
        knocks_enc[5] = ((unsigned short int) blf_out[5]) + offset;
        knocks_enc[6] = ((unsigned short int) blf_out[6]) + offset;
        knocks_enc[7] = ((unsigned short int) blf_out[7]) + offset;


        // START KNOCKING
        count = 0;
        printf("\nPorts: ");

        while((knocks_enc[count] != 0) && (count < MAXNUMOFKNOCKS)){


                // OPEN SOCKET
                sockFileDesc = socket(AF_INET, SOCK_STREAM, 0);
                if(sockFileDesc == -1){
                        perror("\nERROR");
                        exit(0);
                }


                // CONNECT TO REMOTE HOST
                serverAddress.sin_family = AF_INET;
                serverAddress.sin_port = htons(knocks_enc[count]);
                //memcpy((char *)&serverAddress.sin_addr.s_addr,
servername->h_addr_list[0], servername->h_length);

                if(connect(sockFileDesc, (struct sockaddr *)
&serverAddress, sizeof(serverAddress)) < 0){
                        printf("%d ", ntohs(serverAddress.sin_port));
                }
                else{
                        printf("\nERROR: Connection made on port
%d\n\n", ntohs(serverAddress.sin_port));
                        close(sockFileDesc);
                        exit(0);
                }

                close(sockFileDesc);
                count++;

        }  // END WHILE


        // FINISH UP
        printf("\nSequence complete\n\n");
        return 0;
}
```

Appendix B: Source Code for `knockd.c`

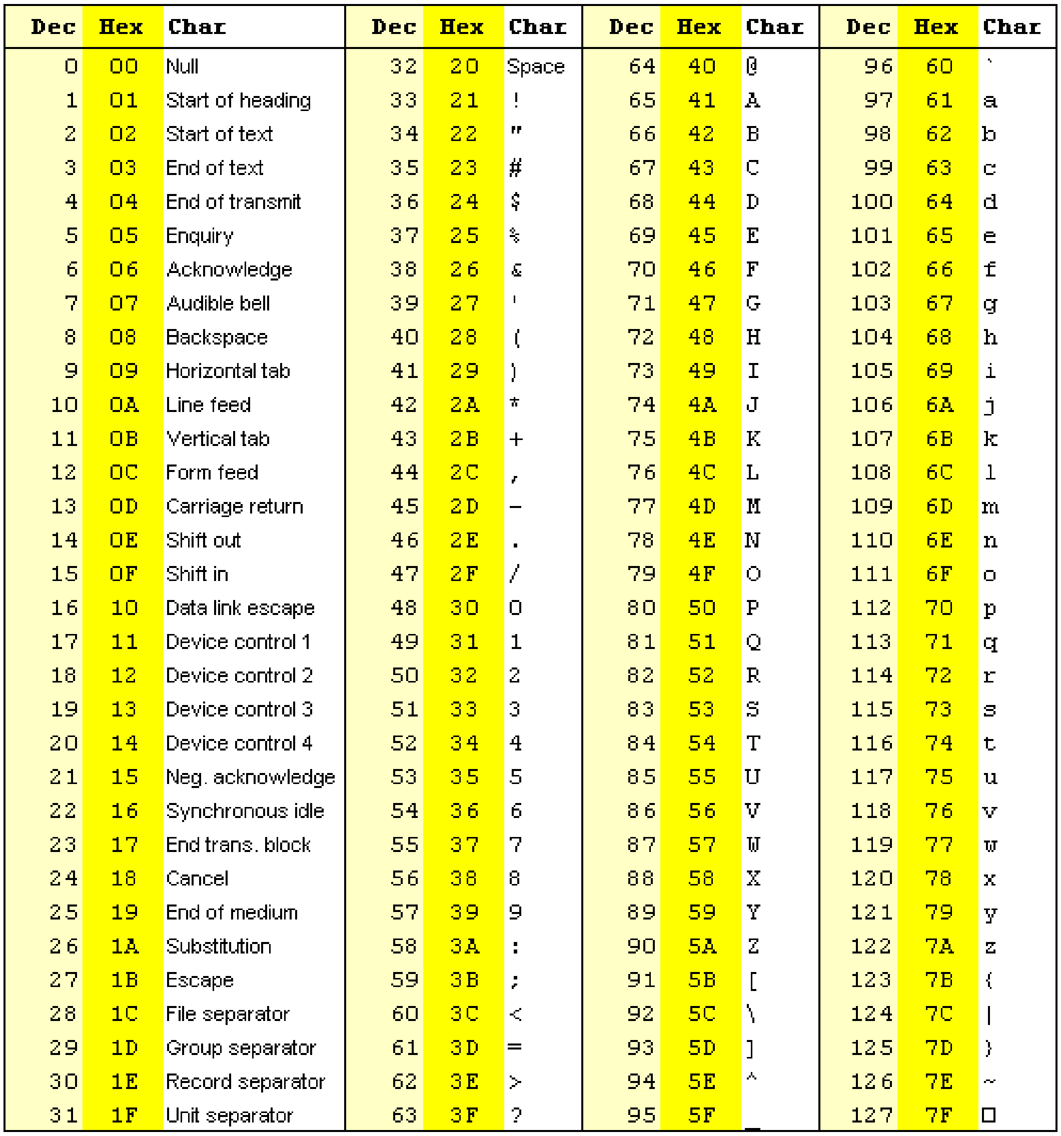
```
//knockd v0.1 by Matt Doyle


// #INCLUDES
#include <slack/std.h>  // from libslack (libslack.org), used for
daemonizing
#include <slack/daemon.h>  // from libslack (libslack.org), used for
daemonizing
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>  // regular expression support
#include <pwd.h>
#include <unistd.h>
#include <sys/types.h>
#include <md5.h>  // md5 support
#include <openssl/blowfish.h>  // blowfish support
#include <time.h>


// #DEFINES
#define NUMOFSUBEXPS 3  // number of subexpressions in our regular
expression
#define MAXNUMOFKNOCKS 8  // defines maximum possible # of knocks
#define MAXPASSWDSIZE 128  // defines maximum password length


// STRUCTURE DEFINITIONS
struct hostdata{
        char originIP[16];
        unsigned char blf_in[MAXNUMOFKNOCKS];  // encrypted string
produced from knock sequence
        unsigned char blf_out[MAXNUMOFKNOCKS];  // decrypted string
        unsigned short int knocks_enc[MAXNUMOFKNOCKS];  // encrypted
knock sequence received from IP thus far
        unsigned short int knocks_dec[MAXNUMOFKNOCKS];  // decrypted
knock sequence
        int knocksreceived;  // indicates how many knocks have been
received from this host
        struct hostdata *next;  // used to make linked list out of host
nodes
        time_t timestamp;  // indicates time we started receiving knocks
from this host
};


// FUNCTION PROTOTYPES
void init_hostdata(struct hostdata *host);
void eval_sequence(struct hostdata *host, BF_KEY bfkey, unsigned short
int offset);
void free_node(struct hostdata *host);
struct hostdata *add_host_to_list(struct hostdata *host);
struct hostdata *host_exists(char *remotehost);
void check_timestamps(time_t window);
void modify_ipfilter_firewall(struct hostdata *host);
```

```
// GLOBALS
char password[MAXPASSWDSIZE];  // decryption password
struct hostdata *hosts;  // linked list of host nodes


// START MAIN
int main(int argc, char *argv[]){


        // VARIABLES
        unsigned int sleeptime = 1;  // # of seconds to wait between md5
hash checks
        char md5hash[33];  // stores md5 hash
        char logfile[50];  // log file name
        char logline[200];  // stores lines from log file as we pull
them out
        char remotehost[16];  // stores subexpression from regexp match
        char localport[6];  // stores subexpression from regexp match
        char password[MAXPASSWDSIZE];  // encryption/decryption password
        regex_t regexp;  // stores compiled regular expression
        regmatch_t subexps[NUMOFSUBEXPS];  // stores matched
subexpressions of our regexp
        fpos_t position;  // stores file pointer
        FILE *logfileptr;
        BF_KEY bfkey;
        struct hostdata *currenthost;  // pointer to current host node
being dealt with
        time_t window = 10;  // amount of time to receive all knocks
from a host before freeing the node  // FIX!
        unsigned short int offset = 0;  // port range offset


        // CLEAR BUFFERS
        memset(md5hash, '\0', sizeof(md5hash));
        memset(logfile, '\0', sizeof(logfile));
        memset(logline, '\0', sizeof(logline));
        memset(remotehost, '\0', sizeof(remotehost));
        memset(localport, '\0', sizeof(localport));
        memset(password, '\0', sizeof(password));


        // INITIALIZE
        memcpy(logfile, "/var/log/messages", sizeof(logfile)-1);
        strcpy(password, getpass("Enter decryption password: "));
        BF_set_key(&bfkey, sizeof(password), password);
        logfileptr = NULL;
        hosts = NULL;


        // PROCESS COMMAND-LINE ARGUMENTS
        if(argc > 2){
                printf("ERROR: usage: %s  OR  %s offset\n", argv[0],
argv[0]);
                exit(0);
        }
        else if(argc == 2){
                if((offset = (unsigned short int) strtol(argv[1], NULL,
10)) > 65279){
                        printf("ERROR: Port offset > 65279\n");
                        exit(0);
                }
        }


        // GET INITIAL MD5 HASH OF LOGFILE
        if(MD5File(logfile, md5hash) == NULL){
                printf("ERROR: Log file %s not found.\n", logfile);
                exit(0);
        }


        // OPEN LOG FILE FOR READING
        if((logfileptr = fopen(logfile, "r")) == NULL){
                perror("ERROR");
                exit(0);
        }


        // COMPILE REGULAR EXPRESSION
        if(regcomp(&regexp, "^.+Connection attempt to TCP [0-9.]+:([0-
9]+) from ([0-9.]+):[0-9]+", REG_EXTENDED) != 0){
                printf("ERROR: Unable to compile regular
expression.\n");
                exit(0);
        }


        // SAVE FILE POINTER FROM INITIAL EOF
        fseek(logfileptr, 0, SEEK_END);
        fgetpos(logfileptr, &position);
        rewind(logfileptr);


        // START MAIN WHILE LOOP
        printf("Running...\n\n");
        while(1){


                // IF LOG FILE UNCHANGED, SLEEP THEN LOOP AGAIN
                if(strcmp(MD5File(logfile, NULL), md5hash) == 0){
                        check_timestamps(window);
                        sleep(sleeptime);
                        continue;
                }

                fsetpos(logfileptr, &position);

                // LOOP OVER ALL LOG FILE LINES FROM SAVED POINT ON
                while(!feof(logfileptr)){

                        check_timestamps(window);
                        memset(logline, '\0', sizeof(logline));
                        fgets(logline, sizeof(logline), logfileptr);

                        if(regexec(&regexp, logline, NUMOFSUBEXPS,
subexps, 0) == 0){

                                memcpy(localport,
logline+subexps[1].rm_so, (subexps[1].rm_eo-subexps[1].rm_so));
                                memcpy(remotehost,
logline+subexps[2].rm_so, (subexps[2].rm_eo-subexps[2].rm_so));
```

```
                                printf("%s knocking port %s\n",
remotehost, localport);


                                // ADD HOST ENTRY IF IT DOESN'T EXIST,
UPDATE IF IT DOES
                                if((currenthost =
host_exists(remotehost)) == NULL){

                                        currenthost =
malloc(sizeof(struct hostdata));
                                        init_hostdata(currenthost);
                                        hosts =
add_host_to_list(currenthost);

                                        strcpy(currenthost->originIP,
remotehost);
                                        currenthost-
>knocks_enc[currenthost->knocksreceived] = (unsigned short int)
strtol(localport, NULL, 10
);
                                        currenthost->knocksreceived++;
                                }
                                else{
                                        currenthost-
>knocks_enc[currenthost->knocksreceived] = (unsigned short int)
strtol(localport, NULL, 10
);
                                        currenthost->knocksreceived++;
                                        if(currenthost->knocksreceived
== MAXNUMOFKNOCKS)

eval_sequence(currenthost, bfkey, offset);
                                }



                                memset(localport, '\0',
sizeof(localport));
                                memset(remotehost, '\0',
sizeof(remotehost));
                                memset(subexps, '\0', sizeof(subexps));

                        }  // END IF

                        else continue;

                }  // END WHILE


                // SAVE NEW LOG FILE HASH AND NEW EOF POINTER
                MD5File(logfile, md5hash);
                fseek(logfileptr, 0, SEEK_END);
                fgetpos(logfileptr, &position);


        }  // END WHILE

        // CLEAN UP
        fclose(logfileptr);
        return 0;
}  // END MAIN
```

```
/* ********** FUNCTION DEFINITIONS ********** */


void init_hostdata(struct hostdata *host){

        memset(host->originIP, '\0', sizeof(host->originIP));
        memset(host->blf_in, '0', sizeof(host->blf_in));
        memset(host->blf_out, '\0', sizeof(host->blf_out));
        memset(host->knocks_enc, 0, sizeof(host->knocks_enc));
        memset(host->knocks_dec, 0, sizeof(host->knocks_dec));
        host->knocksreceived = 0;
        host->next = NULL;
        host->timestamp = time(NULL);

}


struct hostdata *add_host_to_list(struct hostdata *host){

        // VARIABLES
        struct hostdata *current;

        current = hosts;
        if(current == NULL)
                return host;
        while(current->next != NULL){
                current = current->next;
        }
        current->next = host;

        return hosts;
}


struct hostdata *host_exists(char *remotehost){

        // VARIABLES
        struct hostdata *current;


        current = hosts;
        while(current != NULL){
                if(strcmp(current->originIP, remotehost) == 0)
                        return current;
                current = current->next;
        }

        return NULL;
}


void eval_sequence(struct hostdata *host, BF_KEY bfkey, unsigned short
int offset){


        // VARIABLES
        unsigned short int upper = 0;
        unsigned short int lower = 0;

        printf("\nKnock sequence: %d %d %d %d %d %d %d %d\n", host-
>knocks_enc[0], host->knocks_enc[1], host->knocks_enc[2], host-
>knocks_enc[
3], host->knocks_enc[4], host->knocks_enc[5], host->knocks_enc[6], host-
>knocks_enc[7]);

        host->blf_in[0] = ((unsigned char) host->knocks_enc[0]) - 
offset;
        host->blf_in[1] = ((unsigned char) host->knocks_enc[1]) - 
offset;
        host->blf_in[2] = ((unsigned char) host->knocks_enc[2]) - 
offset;
        host->blf_in[3] = ((unsigned char) host->knocks_enc[3]) - 
offset;
        host->blf_in[4] = ((unsigned char) host->knocks_enc[4]) - 
offset;
        host->blf_in[5] = ((unsigned char) host->knocks_enc[5]) - 
offset;
        host->blf_in[6] = ((unsigned char) host->knocks_enc[6]) - 
offset;
        host->blf_in[7] = ((unsigned char) host->knocks_enc[7]) - 
offset;

        BF_ecb_encrypt(host->blf_in, host->blf_out, &bfkey, BF_DECRYPT);

        host->knocks_dec[0] = (unsigned short int) host->blf_out[0];
        host->knocks_dec[1] = (unsigned short int) host->blf_out[1];
        host->knocks_dec[2] = (unsigned short int) host->blf_out[2];
        host->knocks_dec[3] = (unsigned short int) host->blf_out[3];
        upper = host->blf_out[4] << 8;
        lower = (unsigned short int) host->blf_out[5];
        host->knocks_dec[4] = upper | lower;
        host->knocks_dec[6] = (unsigned short int) host->blf_out[6];

        //printf("Decrypted IP: %u.%u.%u.%u\n", host->knocks_dec[0], 
host->knocks_dec[1], host->knocks_dec[2], host->knocks_dec[3]);
        printf("Action: ");
        if(host->knocks_dec[6] == 1) printf("Open ");
        else if(host->knocks_dec[6] == 0) printf("Close ");
        else printf("NO ACTION ");
        printf("port %d to IP address %d.%d.%d.%d\n\n", host-
>knocks_dec[4], host->knocks_dec[0], host->knocks_dec[1], host-
>knocks_dec[2], ho
st->knocks_dec[3]);


        modify_ipfilter_firewall(host);
        free_node(host);
}


void free_node(struct hostdata *host){

        // VARIABLES
        struct hostdata *previous;
        struct hostdata *current;

        if(host == hosts){  // if the node to free is the first in the 
list
                current = host;
                hosts = host->next;
                free(current);
        }
        else{
                previous = hosts;
                current = hosts->next;
                while(current != host){
                        previous = current;
                        current = current->next;
                }
                if(current->next != NULL)
                        previous->next = current->next;
                else
                        previous->next = NULL;

                free(current);
        }
}


void check_timestamps(time_t window){

        // VARIABLES
        struct hostdata *current;

        current = hosts;
        while(current != NULL){
                if(difftime(time(NULL), current->timestamp) >= window)
                        free_node(current);
                current = current->next;
        }
}


void modify_ipfilter_firewall(struct hostdata *host){

        // VARIABLES
        int ruleexists = 0;
        int ruledeleted = 0;
        char firewallrule[74];  // stores string of our firewall rule
        char filebuffer[74];  // stores lines as they are read out of
rule file
        char buffer[6];  // storage buffer
        regex_t regexp;
        FILE *rulefile;
        FILE *tempfile;


        // INITIALIZE BUFFERS
        memset(firewallrule, '\0', sizeof(firewallrule));
        memset(buffer, '\0', sizeof(buffer));


        // COMPOSE FIREWALL RULE STRING
        strncat(firewallrule, " pass in quick on sis0 proto tcp from ",
38);
        sprintf(buffer, "%d.%d.%d.%d", host->knocks_dec[0], host-
>knocks_dec[1], host->knocks_dec[2], host->knocks_dec[3]);
        strncat(firewallrule, buffer, 15);
        strncat(firewallrule, " to any port = ", 15);
        memset(buffer, '\0', sizeof(buffer));
        sprintf(buffer, "%d\n", host->knocks_dec[4]);
        strncat(firewallrule, buffer, 15);
        // COMPILE REGULAR EXPRESSION
        if(regcomp(&regexp, firewallrule, REG_EXTENDED) != 0){
                printf("ERROR: Unable to compile regular 
expression.\n");
                exit(0);
        }


        // OPEN KNOCKD FIREWALL RULES FILE
        if((rulefile = fopen("./ipf.rules.knockd", "r+")) == NULL){
                printf("ERROR: Unable to open ipf.rules.knockd\n");
                exit(0);
        }


        // OPEN TEMPORARY FILE
        if((tempfile = fopen("./temp", "w")) == NULL){
                printf("ERROR: Unable to open temporary file.\n");
                exit(0);
        }


        // IMMEDIATELY INSERT RULE IF FILE EMPTY AND ACTION IS TO OPEN
        if(feof(rulefile) && host->knocks_dec[6] == 1)
                fprintf(rulefile, firewallrule);

        else if(feof(rulefile) && host->knocks_dec[6] == 0)
                {}  // do nothing

        // LOOP THROUGH OUR RULE FILE
        else{

                while(!feof(rulefile)){

                        fgets(filebuffer, 74, rulefile);
                        filebuffer[73] = '\0';

                        // IF RULE MATCHES AND ACTION IS TO OPEN...
                        if((regexec(&regexp, filebuffer, 0, NULL, 0) == 
0) && (host->knocks_dec[6] == 1)){
                                ruleexists = 1;
                                break;
                        }

                        // IF RULE DOESN'T MATCH AND ACTION IS TO 
OPEN...
                        else if((regexec(&regexp, filebuffer, 0, NULL, 
0) != 0) && (host->knocks_dec[6] == 1))
                                continue;

                        // IF RULE MATCHES AND ACTION IS TO CLOSE...
                        else if((regexec(&regexp, filebuffer, 0, NULL, 
0) == 0) && (host->knocks_dec[6] == 0)){
                                ruledeleted = 1;
                                continue;
                        }

                        // IF RULE DOESN'T MATCH AND ACTION IS TO 
CLOSE...
                        else if((regexec(&regexp, filebuffer, 0, NULL, 
0) != 0) && (host->knocks_dec[6] == 0)){
                                fprintf(tempfile, filebuffer);
                                continue;
                        }


                }  // END WHILE


                // APPEND RULE TO OPEN IF IT DOESN'T EXIST
                if((ruleexists == 0) && (host->knocks_dec[6] == 1)){
                        fprintf(rulefile, firewallrule);
                        fclose(rulefile);
                        system("ipf -Fa -f ./ipf.rules.knockd -f
/etc/ipf.rules");
                        fclose(tempfile);
                }


                // DELETE RULE IF IT EXISTS AND ACTION IS TO CLOSE
                else if((ruledeleted == 1) && (host->knocks_dec[6] ==
0)){

                        fclose(rulefile);
                        if(remove("./ipf.rules.knockd")){
                                printf("ERROR: Unable to delete rule
file.\n");
                                exit(0);
                        }

                        fclose(tempfile);
                        if(rename("./temp", "./ipf.rules.knockd")){
                                printf("ERROR: Unable to rename
file.\n");
                                exit(0);
                        }

                        system("ipf -Fa -f ./ipf.rules.knockd -f
/etc/ipf.rules");
                }


        }  // END ELSE


}
```

Appendix C: The Extended ASCII Character Chart



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

Appendix D: Related Work

There are currently a number of port knocking systems in development. The most comprehensive listing of these implementations can be found at www.portknocking.org. The primary example is Krzywinski's prototype port knocking system written in Perl (http://www.portknocking.org/view/download/perl). Krzywinski's code is provided as a proof-of-concept, and is not intended for production environments. Jon Snell has designed an experimental port knocking implementation using C (http://www.e-normous.com/nerd/combo.c). A Java-based implementation, jPortKnock, has been created and released for open source development (http://freshmeat.net/projects/jportknock). Corciovei Aretius has developed a Python-based client/server system for use with FreeBSD systems using the `ipfw` firewall package (http://nemesisit.rdsnet.ro/len/misc/portknock.html).

Judd Vinet recently released a client/server port knocking system, also written in C (http://www.zeroflux.org/knock/). Vinet's implementation is designed in a way that maps system calls to static knock sequences.

Finally, there is an open source project on www.sourceforge.net called Doorman (http://doorman.sourceforge.net). Doorman is a bit different from other port knocking implementations in that it only listens for a single UDP packet with a valid MD5 hash.

Bibliography

1. Krzywinski, Martin. Port Knocking. <http://www.portknocking.org> March 2004.

2. Krzywinski, Martin. “Howto: Port Knocking.” Linux Journal. June 16, 2003. <http://www.linuxjournal.com> March 2004.

3. Lehey, Greg. *The Complete FreeBSD.* Walnut Creek: Walnut Creek CDROM, 1999. 350-351.

4. Malda, Rob. “”Port Knocking” for Added Security.” Slashdot. February 5, 2004. <http://www.slashdot.org> February, 2004.

5. Malda, Rob. “Port Knocking in Action.” Slashdot. April 14, 2004. <http://www.slashdot.org> April, 2004.

6. Oppliger, Rolf. *Security Technologies for the World Wide Web.* Boston: Artech House, 2000. 77-78.

7. Schneier, Bruce. *Applied Cryptography.* New York: Wiley, 1996. 336-339.

8. Sonnenreich, Wes, and Tom Yates. *Building Linux and OpenBSD Firewalls.* New York: Wiley, 2000. 50-52.