CLI Messaging Application

Nir Boneh

University of Colorado Boulder
nir.boneh@colorado.edu

Abstract

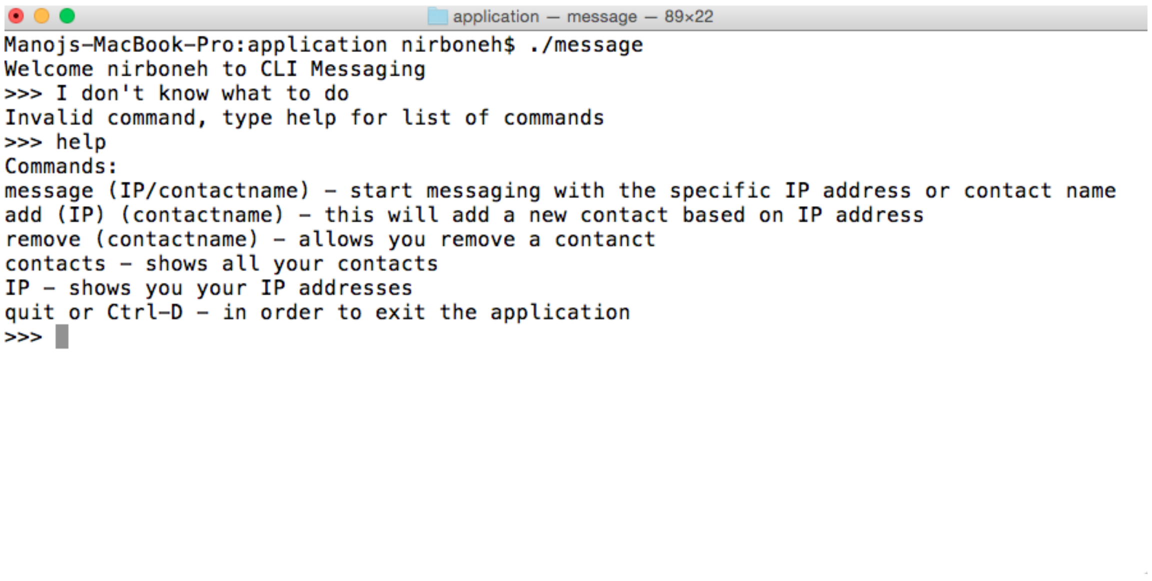
This project discusses the creation and design of a decentralized CLI unix-based messaging application written in C. Examining closely on how to eliminate security risks and introspecting on the usefulness of the application in various situations. The advantages of using this application over other similar application in areas such as security, practicality, and speed is also discussed. Charts showing speed comparisons of sending messages over lan or wan are also shown. It also goes into detail about implementation and includes information on socket programming, hashmaps, and OpenSSL. The source code for the project can be found at https://github.com/nboneh/CLIMessageApp and the repository can be used for downloading and installing the application.

1. Introduction

Instant messaging has become one of the most practical ways we communicate with one another. In this day and age, instead of having to send mail across the physical plane and wait one to four business weeks to hear back from family relatives or friends, we can simply pull out our mobile device, type a message and send it in nearly an instant. While such power as reduced our face to face communication [2], one can't deny the advantages this tool could bring working within a unix terminal. For us developers and UNIX enthusiasts, spending most of our time on a terminal managing files and building application; A CLI tool that allows messaging quickly from the terminal, would be nice to have added to our vast arsenal of gadgets that are within our efficient terminal fingertips. Not to mention such an application would be extremely practical in a situation where we are limited to just a terminal, such as with special types of machines or on remote ssh. While a great messaging CLI application does exist, called telegram [3], it does not serve the purpose of being a decentralized application. That means that big brother could be watching your messages not to mention that security attacks could happen to the the telegram server which is stated to be vulnerable to MITM (Man in the Middle) attacks according to a tech.eu article [7]. The other risk is that the server could go offline rendering the application useless. There is also a chat application that is decentralized called chat terminal [4]. Installation for this application is diffcult and requires many pre requisite items and even after installing those I wasn't able to get it to compile. It also misses the key feature of messaging which is being able to save messaging history. So a messaging application that is decentralized, secure, and easy to install is missing from the terminal tool arsenal, which is the main focus of the CLI messaging application.

2. Design

The CLI messaging application is written in C with over 1,000 lines of code and 800 lines of original code and is written for Unix based systems (Mac and Linux). It is easy to install, using git clone or downloading the source code from github and running make, it does however require that OpenSSL 1.0.xg is installed which might require more effort from Mac users. In this section we will go over the features and commands that CLI Messaging Ap-



application — message — 89×22

```
Manojs-MacBook-Pro:application nirboneh$ ./message
Welcome nirboneh to CLI Messaging
>>> I don't know what to do
Invalid command, type help for list of commands
>>> help
Commands:
message (IP/contactname) - start messaging with the specific IP address or contact name
add (IP) (contactname) - this will add a new contact based on IP address
remove (contactname) - allows you remove a contanct
contacts - shows all your contacts
IP - shows you your IP addresses
quit or Ctrl-D - in order to exit the application
>>> 
```

Figure 1: *The help command showing all available commands.*

plication gives its users and a basic rundown of the commands implementation, a more detailed explnation of how the complex message command works will be covered in the implementation section.

2.1. The IP Command

The IP command allows one to see his LAN and WAN IP addresses. This allows one to quickly share with his friends the way to contact him using the application. It works by having c call a small bash script using the system call that prints out the LAN and WAN IP to stdout.

2.2. Contacts Commands

There are three contacts commands that work to modify the same data. The add command allows one to add a new contact by typing in an IP address and a contact name. The remove command allow one to remove a contact based on his contact name. The contacts command allows one to see all of the current contacts. The contact data is store in a data structure called a hashmap which allows one to retrieve a value, in this case contact name, using a key, in this case IP address. An online open source project implementation hashmap was used for the application [6]. There is a forward and backward hashmap data structure one to look up IPs based on contact names and the other is to look up contact name based on IPs. The reason for this is if you want to send a message you will do so based on the contact name, and if you receive a message you will get the IP and need to know who it is from. The hashmap is saved on the disk in a text file and overwritten every time an add happens, remove happens, or the application exits, so the contacts are reloaded everytime the application is loaded.

2.3. Message Command

The message command allows one to set up a messaging channel communication with another node based on their IP or contact name. Upon receiving the request, the second node decides whether to accept or reject the request. If the other node accepts both nodes will perform a secure handshake and enter message mode. This is where anything you type is sent over to the other user with a timestamp and the message, the entire conversation is always printed out. If one node types in quit, the application will go back to regular interactive mode, for both nodes. The message conversation is saved on disk based on the other node's IP and will be reloaded the next time a messaging channel is setup. The message command has the most complex implementation and will be discussed more in detail in the implementation section.

3. Implementation

This section will go over socket communication and performing OpenSSL to send secure information between the two nodes.

3.1. Messaging between nodes

A communication between two nodes is setup via TCP/IP connection using socket programming. Socket programming in C can be quite diffcult and confusing, so I would like to thank Beej's Guide to Network Programming for helping me setup the code [1]. Before communication channel is open a communication socket has to be setup and there are two ends to this. A connect socket and a listen socket, one request connection and the other accepts the connection. To set up a socket an IP address and a port is required, the application's port is set to be 3490 as macro in the code. The listen socket calls on the accept system call where it hangs till a connect socket requests connection. So how is it possible to run a listen socket without freezing the application? With the use of pthreading. The application sets up a listen socket on a seperate thread when it first boots up. Upon the accept socket system call returning a global is set up to let the UI thread know that a communication has been established. Once the link is performed sending of messages can be done via send and recv system calls, the recv system also hangs so again a second thread has to be setup. Everytime a user types in a message the send call is called and sends it to the receiver. Both the send and receive write to the same file, so a mutex lock is required to keep the resource safe since they run on seperate threads. A mutex lock works by only allowing one thread into a code section at a time making sure only one thread writes to the file. If one thread hangs up or quits the recv function returns a zero which lets the other end user know that the remote has hang up. This node sets a global to true in the receive thread and goes back to interactive mode.

3.2. Avoiding security risks

There are two main security risks that are addressed when building the application. The first one is making sure to encrypt messages between two nodes so no one can decipher them, this is addressed using RSA through the use of a public key and private key. The second problem is being able to share the RSA public keys on a non-secure channel which is addressed using Diffie Hellman key exchange. The way that these algorithms are used in code pratice is through bash script calling on the OpenSSL library reading from a file and writing to a file. c calls on these bash scripts using the command function, writing to a file before and then reading from the output file, both files are then deleted using c's unlink system call.

3.2.1 RSA for secure communication

The general idea of RSA works as follows, each user has a public key and a private key which they can generate using the RSA algorithm. Upon generating these two keys, a user can share his public key with everyone, but keeps the private key to himself. Now anyone who wants to share secure messages with another user, encrypts the message using the user public key, but the message can only be decrypted using the private key. Therefore, no one can read messages sent to the user other than the intended user himself. This is the main method of communication when two users are sending messages back and forth using the application. The RSA algorithm involves generating two large prime numbers and multiplying them togther to get a composite. In order for anyone to crack the RSA algorithm it would require being able to figure out the two prime numbers using factoring which is extremely diffcult and would take a ridicilous amount of time doing by brute force, so one can trust RSA to be a rather secure algorithm.

3.2.2 Diffie Hellman key exchange

The only issue with the RSA method alone is that the user needs to share his public key with another user on a non-secure channel and this is where security risks are prone, this is where Diffie Hellman key exchange comes into play. One possible security risk that could happen when sharing a public key on a non-secure channel is called Man in the Middle Attack. Let's imagine two users Alice and Bob, and a malicious user called Chris, here is how a man in the middle attack will play out:

1. Bob sends a package to Alice: Hey Alice this is Bob, here is my public key: Bob's key.
2. Chris intercepts the message and swaps out Bob's key with his own key.
3. Alice receives the message and think that Bob's key is actually Chris's key.
4. Alice sends back an encrypted message using Chris's key sharing her public key.
5. Chris intercepts that message as well, and decrypts it with his key and changes Alice's key to his key.
6. Bob receives the message and assume Alice's key is Bob's key.
7. From this point on Chris has got all the power over the conversation, being able to see it and also change it as he likes.

The only way to avoid such a terrible scenario is using the Diffie Hellman key exchange algorithm to share the public keys, the math is somewhat complicated, but the logic is rather simple and can be explained using colors:

1. Bob and Alice share a public color which they both agree on, Chris can know about this public color.
2. Bob and Alice both generate a private color neither one knows of the other and neither does Chris.
3. Bob and Alice mix the public color with their private color and send it to each other, Chris can know about this color.
4. Bob and Alice mix this color with their private color again and they both get the same color, now they both know a color that Chris doesn't know.
5. Bob and Alice can now send the public key using this fourth color for encoding without Chris being able to manipulate the situation.

While it is true that a man in the middle attack could disrupt the handshake, it will not be able to gain power over the conversation, since it can not know the fourth color. If a man in the middle attack those disrupt the process, the handshake can be retried over and over till he gets bored and moves on. The last component is using the secret known color to encrypt the RSA public key, this is done through an cipher algorithm called AES, which encrypts message based on a value and then the message can be decrypted using that same value. So, basically the way the application avoids security risks is by sharing public keys using Diffie Hellman and from there messages are exchanged using RSA.

4. Evaluation

4.1. Methodology

In order to compare the efficiency of the application, a speed comparison is done with well known CLI centralized messaging application called Telegram. The CLI messaging application is tested over WAN and LAN networks and the Telegram is tested normally, since you have to send a message through a server which makes it WAN. The test is performed on short messages, medium messages, and long messages. To test the speed of messages on the CLI Messaging application a time shell script

Messaging Speed (Seconds)			
Message Length	Telegram	CLI Messaging App LAN	CLI Messaging App WAN
Short (10 chars)	0.764	0.376	0.174
Medium (30 chars)	0.768	0.382	0.188
Long (75 chars)	0.774	0.383	0.199

Table 1: *Shows speed comparison of CLI Messaging App on LAN and WAN network as well as telegram*

is called to print the time in milliseconds before RSA encryption and after the RSA decryption on the receiving end. The difference is then compared to get the speed. For the Telegram performane, what they like to call a bot is set up, and then using the bot send a message to my Telegram account is sent using CURL and then eveluated using the UNIX time command.

4.2. Results

These are some strange results, but here is my theory on what happened. When I was testing WAN I was testing my Mac communication with a Google Compute Engine Server and when I was testing on LAN I was testing my Linux VM and my Mac. So, the Google Compute Engine Server has less bottleneck in execution performance than the VM. My theory is that network I/O wasnâĂŹt the main culprit for speed execution, it was the RSA computation. As for telegram being slower, I assume it is due to them having to run messaging through a centralized server and having to serve many clients.

References

[1] Hall, Brian "Beej Jorgensen." Beej's Guide to Network Programming: Using Internet Sockets. Place of Publication Not Identified: Publisher Not Identified, 2009. Print.

[2] Hemmer, Heidi. "Impact of Text Messaging on Communication." Minnesota State University, Mankato, 2009.

[3] "Telegram a New Era of Messaging." Telegram. N.p., n.d. https://telegram.org/

[4] "LAN Chat and Text Conferencing in Easiest Way with Vypress Chat." Chat Terminal. N.p., n.d. http://www.vypress.com/free_chat/

[5] Malizia, Nicola. "Getting Started with Telegram Bots." Unnikked. N.p., 25 June 2015. https://unnikked.ga/getting-started-with-telegram-bots

[6] "Petewarden c_hashmap" GitHub. N.p., n.d. https://github.com/petewarden/c_hashmap

[7] Wauters, Robin. "Supposedly Super Secure Telegram App Is Vulnerable to MITM Attacks, Cybersecurity Expert Claims." Tech.eu. N.p., 29 Apr. 2014. http://tech.eu/brief/supposedly-super-secure-telegram-app-possibly-vulnerable/