Nathan Reed

TauNet Private Messaging Server Aplication

Software Design Document

Date: 11/01/2015

TABLE OF CONTENTS

1. INTRODUCTION

1.1 PURPOSE

The purpose of this project was to come up with an agreed upon protocol that would allow us as a class to individually design our own software that would link our private servers together on a private network on which we could send encrypted messages to each other.

1.2 SCOPE

This software is designed to run on the raspberry pi 2. It will allow members of the TauNet group to send and receive encrypted messages over a public internet connection.

1.3 OVERVIEW

There are three main components to accomplish encrypted communication between pis.: a client connection which creates a socket to the desired recipient of the message; a server which listens for incoming messages on a given port; and code for the the encryption and decryption of the message (in our case we used the cipher saber 2 encryption algorithm. In the following sections I will explain each of these sections in more detail.

1.4 REFERENCE MATERIAL

As resources, I used a number of google searches along with enlisting the help of my classmates, and Bart Massey, the professor. docs.python.org/3/library/socket.html was a great resource for socket examples and Bart's github account was a very valuable resource for the the cipher saber 2 code for testing and the design algorithm.

2. SYSTEM OVERVIEW

To receive a message the server listens on the designated port for incoming messages to that port. In our case the port is 6283. When the program is running idle it is listening. When an incoming message is received by the by the server it is decrypted and appended to the messages list. To send a message, the message is read in and encrypted using the cipher saber 2 encryption. The server connects to the target server's IP address and designated port via a socket and sends a message.

The user has the option of either sending a message to a select list of TauNet members or checking their messages.

The program is designed with three modules. The previous description of it's basic functionality gives you an idea of how these modules interact.

3. SYSTEM ARCHITECTURE

3.1 ARCHITECTURAL DESIGN

The basic program structure is simple. I will break it down into three sections or modules and explain how the interact.

The client class which manages a connection with a remote user. It uses python sockets to connect to the target IP address and send a message. If it cannot connect it throws an error. This is only used for sending messages.

The server class also uses python sockets to listen on the designated IP and port. For my case I chose to listen on the wild card address. When a message is received each byte in the stream is received individually and appended to a buffer until there is no more to be read. This message is then appended to a list of received messages.

To accomplish sending and receiving messages I use two threads for concurrency. This was accomplished with python's threading library. When the program is run, the server class starts listening on it's own thread. The server is always listening. The client starts a separate thread when it wishes to send a message. I'm not sure of the robustness of the code, but I have not had any related errors as of yet.

The encryption part of the program consists of an implementation of cipher saber 2's version of the rc4 encryption algorithm, an encryption function which adds a 10 byte random IV password (key) and a decrypted function which deconstructs the encrypt. To facilitate this, everything is done in bytes. Python Byte arrays made easy work of it.

3.2 DESIGN RATIONALE

I chose the architectural design I did because it allowed me to break the project down into pieces that were big enough to run meaningful tests but not so big they were unmanageable. This architecture seemed to be the natural breakdown of the program.

4. DATA DESIGN

4.1 DATA DESCRIPTION

Aside from pythons data structures: lists, byte objects and byte arrays, there are not significant data structures to speak of. Bytes are very important in this program as is the use of lists. Starting off I had a really difficult time using byte objects and byte arrays. Python is a new language for me and the documentation is not always the greatest. But it came down to working in bytes only and converting everything to bytes right away.

5. COMPONENT DESIGN

Pseudo- code for each of the components is as follows. Some of this may look familiar as I'm copying and pasting from github.

rc4

```
    rc4(n, r, k):
+   l <- length k
+   -- Initialize the array.
+   S <- "zero-based array of 256 bytes"
+   for i in 0..255
+       S[i] <- i
+   -- Do key scheduling.
+   j <- 0
+   repeat r times
+        for i in 0..255
+            j <- (j + S[i] + k[i mod l]) mod 256
+            S[i] <-> S[j]
+    -- Finally, produce the stream.
+    keystream <- "zero-based array of" n "bytes"
+    j <- 0
+    for i in 0..n-1
+        i' <- i mod 256
+        j <- (j + S[i']) mod 256
+        S[i'] <-> S[j]
+        keystream[i] <- S[(S[i'] + S[j]) mod 256]
+    return keystream
```

Encrypt

```
-- "Ciphersaber-2 encrypt message" m "with key" k "and"
+-- r "rounds of key scheduling"
+encrypt(m, r, k):
+     n <- length m
+     iv <- "appropriately-chosen 10-byte IV"
+     k' <- prepend k to iv
+     keystream <- rc4(n, r, k')
+     ciphertext <- "zero-based array of" n + 10 "bytes"
+     for i in 0..9
+         ciphertext[i] <- iv[i]
+     for i in 0..n
+         ciphertext[i + 10] <- m[i] xor keystream[i]
+     return ciphertext
```

Decrypt

```
+-- "Ciphersaber-2 decrypt ciphertext" m "with key" k "and"
+-- r "rounds of key scheduling"
+decrypt(m, r, k):
+     n <- length m
+     iv <- m[0..9]
+     "delete the first 10 characters of" m
+     k' <- prepend k to iv
+     keystream <- rc4(n - 10, r, k')
+     plaintext <- "zero-based array of" n - 10 "bytes"
+     for i in 0..n-10
+         plaintext[i] <- m[i] xor keystream[i]
+     return plaintext
```

Server

-Using socket library
-pair server to host and port
-create a listening socket to listen for incoming connections
-bind the listener to the host_pair
-listen on designated port

Receiving messages:
-create empty bytes buffer
-read in bytes one at a time and append to buffer using socked recv()
-if message was empty print message and exit
-otherwise run decryption and append message message repository list

Client

-Using socket library
-create a socket
-connect socket to target
-send message
-shut down and close socket

6 HUMAN INTERFACE DESIGN

6.1 OVERVIEW OF USER INTERFACE

The rest is a simple user interface which allows the user to choose whether they want to send or check messages. If they want to send, they can choose from a list, of members. If they want to check messages, it prints the message list.