Developing a Secure Instant Messaging System for Upper Level Clearance Communications

Alexander S. Beaty

CS 4395, Senior Project

Department of CST

University of Houston-Downtown

Advisor: Dr. Shengli Yuan, *email*: [yuans@uhd.edu](mailto:yuans@uhd.edu)

Department of CST

University of Houston-Downtown, TX 77002

Advisor's signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Author's signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[**Abstract**](#_30j0zll)[4](#_30j0zll)

[**I. INTRODUCTION**](#_1fob9te)[5](#_1fob9te)

[**II. TERMS AND DEFINITIONS**](#_3znysh7)[8](#_3znysh7)

[**III. TLS**](#_2et92p0)[9](#_2et92p0)

[a. SSL/TLS – Background](#_tyjcwt) [9](#_tyjcwt)

[b. TLS Protocol- Design](#_3dy6vkm) [11](#_3dy6vkm)

[c. TLS Protocol- Implementation](#_1t3h5sf) [13](#_1t3h5sf)

[**IV. BASIC NETWORKING MODEL**](#_4d34og8)[16](#_4d34og8)

[a. Prototype 1 - Design](#_2s8eyo1) [16](#_2s8eyo1)

[b. Prototype 1 - Implementation](#_17dp8vu) [18](#_17dp8vu)

[c. Prototype 2 - Design](#_3rdcrjn) [18](#_3rdcrjn)

[d. Prototype 2 - Implementation](#_26in1rg) [19](#_26in1rg)

[**V. ACCESS CONTROL**](#_lnxbz9)[21](#_lnxbz9)

[a. Secure Login Protocol – Design](#_35nkun2) [21](#_35nkun2)

[b. Database – Design and Implementation](#_1ksv4uv) [22](#_1ksv4uv)

[c. Authentication through password hashing - Background](#_44sinio) [24](#_44sinio)

[d. Password Hashing - Implementation](#_2jxsxqh) [28](#_2jxsxqh)

[e. Secure Login Protocol – Implementation](#_z337ya) [29](#_z337ya)

[f. Certificate Fingerprints Extraction – Implementation](#_3j2qqm3) [30](#_3j2qqm3)

[g. Registering a new user – Design](#_1y810tw) [32](#_1y810tw)

[h. Registering a new user - Implementation](#_4i7ojhp) [32](#_4i7ojhp)

[**VI. CONFIDENTIALITY**](#_2xcytpi)[33](#_2xcytpi)

[a. Symmetric AES encryption – Background](#_1ci93xb) [33](#_1ci93xb)

[b. Symmetric AES encryption – Implementation](#_3whwml4) [34](#_3whwml4)

[c. Diffie-Hellman key exchange – Background](#_2bn6wsx) [35](#_2bn6wsx)

[d. Diffie-Hellman Protocol – Design](#_qsh70q) [37](#_qsh70q)

[e. Diffie-Hellman Protocol – Implementation](#_3as4poj) [39](#_3as4poj)

[i. Client-Server](#_1pxezwc) [39](#_1pxezwc)

[ii. Client-Client](#_49x2ik5) [40](#_49x2ik5)

[**VII. INTEGRITY**](#_2p2csry)[42](#_2p2csry)

[a. RSA Signature Verification Protocol for DH – Design](#_147n2zr) [43](#_147n2zr)

[b. RSA Signature Verification Protocol for DH Parameter Exchange – Implementation](#_3o7alnk) [44](#_3o7alnk)

[c. HMAC Integrity Check Protocol for Chat Messages – Design](#_23ckvvd) [44](#_23ckvvd)

[d. HMAC Integrity Check Protocol for Chat – Implementation](#_ihv636) [45](#_ihv636)

[**VIII. VALIDATION**](#_32hioqz)[46](#_32hioqz)

[a. WireShark Results](#_1hmsyys) [46](#_1hmsyys)

[i. Prototype 1 – without TLS and without encryption](#_41mghml) [46](#_41mghml)

[ii. Prototype 1 – without TLS and with encryption](#_2grqrue) [47](#_2grqrue)

[iii. Final Version – with TLS and encryption](#_vx1227) [48](#_vx1227)

[**IX. FUTURE WORK**](#_3fwokq0)[49](#_3fwokq0)

[**X. CONCLUSION**](#_4f1mdlm)[51](#_4f1mdlm)

[**XI. REFERENCES**](#_2u6wntf)[53](#_2u6wntf)

[**XII. SOURCE CODE CREDIT**](#_19c6y18)[54](#_19c6y18)

[**XIII. ACKNOWLEDGMENTS**](#_3tbugp1)56

# Abstract

This paper explores the development of a secure instant messaging system. Its three main goals are access control, message confidentiality, and data integrity. The Secure Chat system addresses these goals using reliably tested cryptographic techniques. It uses TLS for basic connection security, password hashing for access control, encryption for message confidentiality, hash functions for data integrity and a well-recognized protocol for creating keys for encryption. The encryption keys for chat messages are single-use and known only to the clients, therefore the server cannot decrypt their messages.

The architecture uses a classic client-server model. The centralized server relies heavily on public/private key certificates that are tied to a user's identity. Almost every security feature in the Secure Chat system involves certificates and would not work without them. The goal of this project is to show how strong identity binding increases the security of a centralized system and minimizes its weaknesses. The drawbacks of this system are loss of anonymity, lack of convenience and relinquishing control to a central server. This paper will prove by example that these drawbacks are more like trade-offs that are negatively correlated with high levels of security.

# INTRODUCTION

There are two main styles of network protocols used for applications connected over the internet: the client-server model and the peer-to-peer model. The former is centralized and the latter is decentralized, with every peer acting as a server for other peers to connect to. For a secure instant messaging system, many users would prefer the latter because they do not have to place all their trust in the managers and developers of the server regarding their concerns of privacy and security. But a centralized server has a few strategic advantages over the peer-to-peer model.

First, there is only one piece of hardware to maintain. Second, once the client software has been distributed, it cannot be protected from the threat of malicious code being inserted into it or replacement of the entire client binary with a trojan. The client code would have to contain all of the security features and do all of the authentication work to ensure it is connecting only with legitimate clients. Therefore, a centralized server can more efficiently allow reputable connections and establish a uniform connection protocol so that rogue or masquerading clients will be refused. This paper shows an example of an instant messaging server that can demonstrate these requirements.

The second role this paper has is to highlight the balance between anonymity and established identity in security. By implementing a secure chat server that emphasizes the later extreme i.e., known identities, this project supports the argument that having all users in a system connected to a permanent identity provides the highest degree of secure communications and access control. Such systems are regularly used in the military where security is obviously the highest priority, and where Public Key Infrastructures (PKIs) are implemented in a similar fashion to this project.

Chapter III covers the use of TLS to secure the primary connection between client and server. TLS is a first line of defense and it is a protocol that contains most of the security features of this application before those features are implemented again by the application itself. This is where the PKI is first used since the server and every client have unique public key certificates that TLS uses to authenticate both sides automatically. Chapter III also explains how TLS provides confidentiality and message integrity. All this is done just above the TCP layer, not by the application itself.

Chapter IV explains the design of the basic client-server architecture for the Secure Chat Server itself, starting with Prototype 1, the problems encountered, a possible solution that was researched and development of Prototype 2.

Chapter V will address access control. It discusses the database design for a user account and the decisions made to choose the design. Then it describes the password authentication scheme. Like most login systems, this secure instant messaging server stores the hash of a user's password instead of the password itself. Unlike insecure login systems, it employs an advanced hashing algorithm called PBKDF2 which uses key stretching to increase the entropy of a password. This is good enough for most systems, but for this system an additional measure is taken to ensure correct identification. The user must login on the device with his certificate on it because the fingerprints of that certificate are checked at the same time the password is validated. Lastly this chapter discusses the Register New User protocol.

Confidentiality is addressed in Chapter VI. The Diffie-Hellman protocol is used to create a shared key between the client and server to encrypt login data, and a new key is created between two logged in users in the same manner to encrypt their communications during that particular session. Each of these keys is discarded when both parties are finished using them. This provides “forward secrecy.”

Integrity is addressed in Chapter VII. Providing integrity means proving whether a message has been tampered with. The first half of the chapter discusses RSA signatures for the DH protocol. The second describes HMAC hashing for chat messages. In the Secure Chat system, an encrypted message is sent along with its hash. When the encrypted message is received, a new hash is calculated from it and compared to the original hash. This way, an attacker should not be able to modify a message without the receiver noticing it.

Chapter VIII presents the results of using Wireshark to sniff packets and determine that they have been encrypted.

Chapter IX will address future work that should be done to improve usability and scalability. The paper's conclusion is in Chapter X.

# TERMS AND DEFINITIONS

* 1. **Sockets** – connection endpoints in Java. The SSL secured version is an SSLSocket.
  2. **Encryption –** the process of permuting data into an illegible, meaningless form to protect it.
  3. **Decryption –** the process of reversing encrypted data into its original form.
  4. **Asymmetric key pairs –** a public key and a complimentary private key. One is used to decrypt what the other has encrypted.
  5. **Certificate** – a file containing identifying information and the public key of an asymmetric key pair. The file has been “signed” by another certificate’s private key.
  6. **Signing –** a private key encrypts a piece of data to prove authenticity. The signature is “verified by using the public key to decrypt it.
  7. **Threads** – subprocess of a running program that run concurrently to execute a task.
  8. **Hash** – A unique identifier for data calculated using a one-way function. It must have the property of being non-reversible and unique for any input.
  9. **JRE** – Java Runtime Environment. The JRE executes processes in the Java Virtual Machine (JVM). It acts like an operating system for the JVM.
  10. **Foreign keys –** Primary keys belonging to another table in a database. Primary keys are attributes that contain unique values in a table allowing each row to be identified. Foreign Keys act as links to those identifiable rows from another entity.
  11. **XOR** – a logical operation that produces “true” only when inputs differ. "A or B, but not, A and B"

# TLS

## SSL/TLS – Background

A secure instant messaging service will need to use the same trusted security protocols that almost all applications use today. Digital traffic, which travels across the Application Layer and over the Transport Layer of TCP/IP, needs to be encrypted. This job was first given to the Secure Sockets Layer (SSL), a protocol published in 1995, which established the rules for peers connected over the internet to communicate securely. The first step is called the “handshake,” in which the client and server authenticate each other using certificates and agree on a common cipher suite. The cipher suite prescribes a set of algorithms that usually include: a key exchange algorithm, a bulk encryption algorithm, and a message authentication code (MAC) algorithm.

A certificate is the signed public key belonging to an asymmetric key pair. It has been signed by the private key of another certificate called a Certificate Authority (CA) which is trusted by many sources. It is important to note that in reality a CA will only sign the certificate of an entity with provable documentation to back up its identity. This Secure Chat system assumes that every registered user has first gone through this process independently, not through the software. Chapter V, Subsection ‘g’ shows how the Secure Chat client program creates a new certificate for the user, but it is up to the user to have the certificate signed by the Server CA’s certificate and prove his real identity.

When the CA signs a certificate (or any piece of data) it uses its private key to encrypt the data it has “signed” or a hash of the data. Since only the corresponding public key can decrypt what was signed by that private key, the signature can be verified by anyone who has the CA's certificate, which is publicly and widely available. This forms the basis of trust that TLS lies on.

SSL has now been replaced by the Transport Layer Security protocol, or TLS, in 1999. Later it was prohibited from use by the Internet Engineering Task Force due to discovered security flaws (Dierks & Rescorla 2008)1. Security protocols and other tools continuously become obsolete, and even TLS version 1.0 is now no longer acceptable by the Payment Card Industry Data Security Standard (Ramakanta, 2017)2. This project uses TLS v1.2, the latest version. However, it is not enough to use TLS without configuring an application properly, as was pointed out by Atighetchi et al. (2013) in their paper “Safe Configuration of TLS Connections - Beyond Default Settings.”3 Several vulnerabilities were described, including an attack on the CBC encryption mode used by TLS v1.0, and an information leakage called CRIME which works on any TLS configuration that uses compression, among other vulnerabilities.

One of the suggestions made by Atighetchi et al. (2013, p.418)3 to safeguard against these threats was to impose a strict adherence to using the latest version of TLS (v.1.2) by all parties during the handshake process. The reason behind this policy is that before the client and server initiate mutual authentication, they have to “negotiate,” or agree on the version of the protocol which is accessible by the party with the lowest version. This leaves the door open for malicious actors to perform attacks that lower versions are susceptible to, either by exploiting a session that was down-negotiated purposefully, or by the attacker forcing that to occur.

Another safeguard the authors outlined was to substantiate a “crumple zone,” or first line of defense made of proxies that are designed to absorb attacks. All TLS connections would go through the crumple zone and all data will be examined for threats there first before being passed along to the intended receiver. This application was designed with these threats and safeguards in mind.

## TLS Protocol- Design

When using TLS with the Java Secure Sockets Extension (JSSE), there is little room for creative experimentation, as well there shouldn't be. The JSSE Application Programmer Interface (API) implements the SSL/TLS protocol on its own, but it relies on certificates that exist on a hard drive. In other words, it does not create temporary certificates for the duration of the session. Even when it is used in default mode without custom made certificates, JSSE will consult a file called cacerts that contains a plethora of certificates from the leading companies that act as Certificate Authorities.

The JSSE API uses an SSLContext to generate SSLSocketFactories which in turn generate SSLSockets (see Figure 1 below). The SSLContext is initialized by a KeyManager and TrustManager. The TrustManager holds individual public key certificates of trusted parties. The KeyManager holds key entries that consist of a “private key accompanied by the certificate chain authenticating the corresponding public key,” according to the Java Cryptography Architecture (JCA) Reference Guide4. During the handshake, the KeyManager will present this certificate chain to the other party as proof of identity for the party it represents. The other party then consults its TrustManager to see if the certificate chain it received matches any of the ones in its database.

Examples from the Java Secure Socket Extension (JSSE) Reference Guide show the correct way to configure the key store, trust store, Key Managers, SSLContext, and SocketFactories. The design of this program followed the conventions laid out in those examples in both the client and server programs.

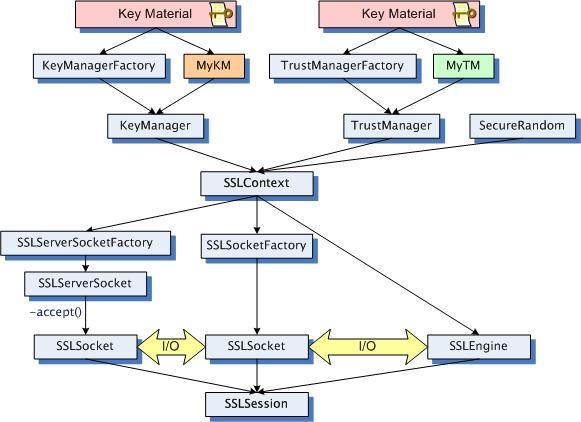


Figure 1: JSSE Classes and Interfaces

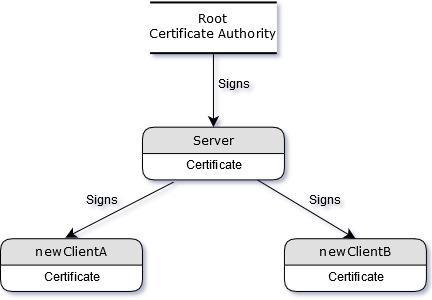
The process of initialization and generation between classes in JSSE.

Image credit: Java Secure Socket Extension (JSSE) Reference Guide

## TLS Protocol- Implementation

As mentioned above, the implementation of the SSL sockets setup code in this project was very similar to the examples in the JSSE Reference Guide. Although the Java Developer’s Kit (JDK) Security API contains methods for generating public keys and private keys, it “has no public certificate APIs that would allow you to create a certificate from a public key,” (*Generating and Verifying Signatures > Weaknesses and Alternatives*)5. Instead, certificates must be made outside the program using Java keytool.exe. The keytool program comes with every installation of the JDK and is used to make self-signed certificates. The algorithms used by keytool for creating asymmetric key certificates are the Digital Signature Algorithm (DSA) and Rivest–Shamir–Adleman (RSA) algorithm. RSA is the preferred standard for SSL certificates because it allows stronger cipher suites to be used (*owasp.org*)6. Early versions of this project used DSA certificates because by default they are made by keytool. Later, new RSA certificates were made to replace them. Test-certificates were made for two clients (newClientA and newClientB) and one for the server. As mentioned previously, keytool only makes self-signed certificates, which would not be used in an actual security environment since they have no certificate chains established that show a trusted third party (CA) vouches for them.

OpenSSL is a security tool that has the capability to use certificates to sign other certificates. First, OpenSSL was used to create a self-signed RSA certificate that acts as the Certificate Authority (CA). Then, keytool was used to create Certificate Signing Requests (.csr files) for each of the three test-certificates. A .csr file is given to OpenSSL to generate the public key certificate on behalf of the key pair it was made for. First the Server Certificate was signed by the CA using OpenSSL. Next, the Server Certificate acted as an intermediate certificate authority and signed the test-certificates newClientA and newClientB using OpenSSL. This established a certificate chain hierarchy, where the CA is the root all three certificates (see Figure 2 below).

Figure 2: Certificate Hierarchy for Secure Chat system

.

Even though the client test-certificates were signed directly by the Server Certificate, they must each import the root CA into their KeyStores so that their certificate chains are valid. The Server Certificate must do the same. Each of the three entities has a TrustStore which holds the signed public certificates of the parties it will directly communicate with, as well as the certificate at the root of their chains. The TrustStore used by a client has the certificate of the Server and the root CA. The TrustStore for the Server has the certificates of newClientA and newClientB, as well as the CA.

All of these import operations are done manually using keytool.exe. It is apparent then that managing the server requires every new user certificate to be imported into the server’s TrustStore one-by-one. This is a convenience trade-off that in mandatory for mutual authentication.

The “Sever Hello” is the first response to the “Client Hello” of the SSL handshake (see Figure 3). During this step, the server looks for its certificate in its KeyStore and sends its certificate to the client. Then the server sends a message requesting the client's certificate. This request is referred to as mutual authentication. Next the client sends its certificate in response to this request of mutual authentication. The client then verifies the server's certificate by consulting its TrustStore to find a matching certificate. The matching certificate includes the root CA as part of its chain, therefore the received server certificate must be signed by the same root CA. This is followed by a key exchange in which both sides generate a shared session key. Finally, the two parties signal to each other to “Change Cipher Spec” which means they are ready to begin encrypting messages. The client and server send each other a “Finished” message and the handshake is completed. From this point on, all application data is encrypted by SSL. When they are finished communicating they will send each other “Close Messages.”

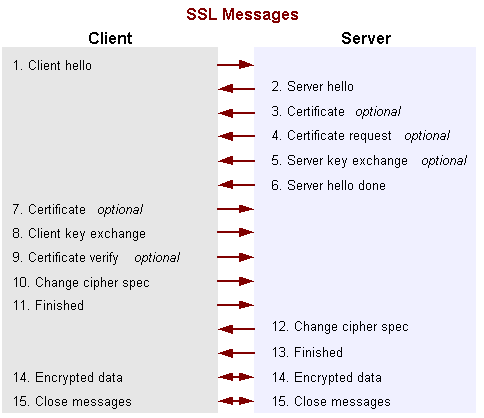
Figure 3: Sequence of Messages Exchanged in SSL Handshake

Image Credit: Java Secure Socket Extension (JSSE) Reference Guide

# BASIC NETWORKING MODEL

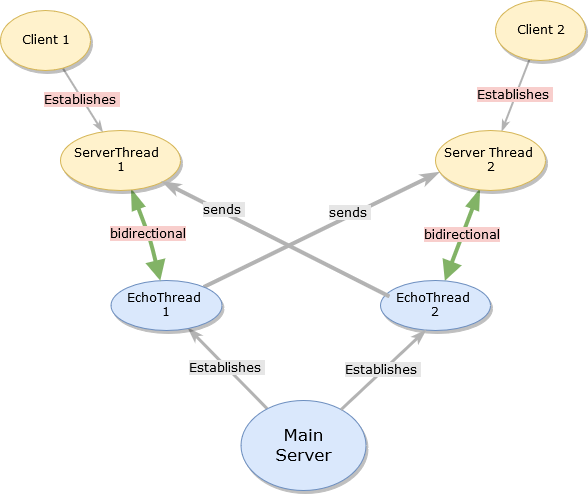
## Prototype 1 - Design

The first prototype of the client-server architecture for the instant messaging system was based on a multi-threaded design by user CanadianLobster on Instructables.com which was debugged and rewritten by user fliedonion on gist.github.com. It is a chat-room style messaging system and does not support private conversations or use a presence-lookup system or publish-subscribe system. Users can only talk to whoever else happens to be in the chat room.

Multi-threading was chosen because it allows synchronous I/O (also known as blocking I/O) to be coordinated during execution. When the OS performs read and write operations, those system calls “block” the execution of a program until data is available, therefore multi-threaded architectures are developed as a work-around.

Prototype 1 can be compared to the common “one thread, one client” model in which the server runs a loop that accepts a client and creates a new thread to handle that client. This strategy allows the main method of the Secure Chat Server (referred to as EchoServer) to continue accepting new connections without blocking until a new client is accepted. Otherwise, it must cater to one client at a time without accepting any new clients. The thread created by EchoServer is called an EchoThread and its purpose is to receive messages from a client and echo them out to all other connected clients (See Figure 4). Each EchoThread iterates through a list containing all other EchoThreads active on EchoServer, and sends the message it received from its client to all other clients.

This methodology effectively creates a chat-room where all connected clients are talking to all other currently connected clients. On the client side, messages are sent from its main method (called EchoClient) to a helper thread called a ServerThread, which passes those messages on to its EchoThread and also receives messages from its EchoThread. The original author of this design did not want EchoClients to block while waiting for the user to type a message, and so the author's solution was to create a shared list between an EchoClient and its ServerThread called “MessagesToSend.” EchoClient added messages to MessagesToSend, which was synchronized to avoid concurrent modification. The ServerThread would check MessagesToSend after it checked for input from the EchoThread, then send messages if there were any.

Figure 4: Prototype 1 – Basic Networking Model

This model shows communication flows for chat messages.

## Prototype 1 - Implementation

When using regular, unsecured sockets, this design works fine. After converting it to use SSLSockets, messages sent to EchoThreads were not relayed to the ServerThreads. This was the first major bug in the development process. After a few months of frustrating trial and error, the problem was narrowed down to the reading of data in the ServerThread. Recall that Prototype 1 used multi-threading to work-around blocking I/O. Because the ServerThread handles input and output for an EchoClient in a procedural fashion, it first checks for incoming messages, then writes outgoing messages. When checking for incoming messages, it calls the available() method on its socket input stream to check it for data and return the number of available bytes. if no data is present the program can skip over a read call instead of blocking the progress of the thread.

It became apparent that available() always returned zero even when a message had just been sent to that input stream. Further investigation revealed that the method always returned zero when used in conjunction with SSLSockets because all data sent over an SSL connection is encrypted by the socket, and it would need to be decrypted by reading from the socket before the number of bytes can be reported by available(). Essentially, development reached an impasse between SSLSockets and the blocking I/O of the multi-threaded client-server architecture.

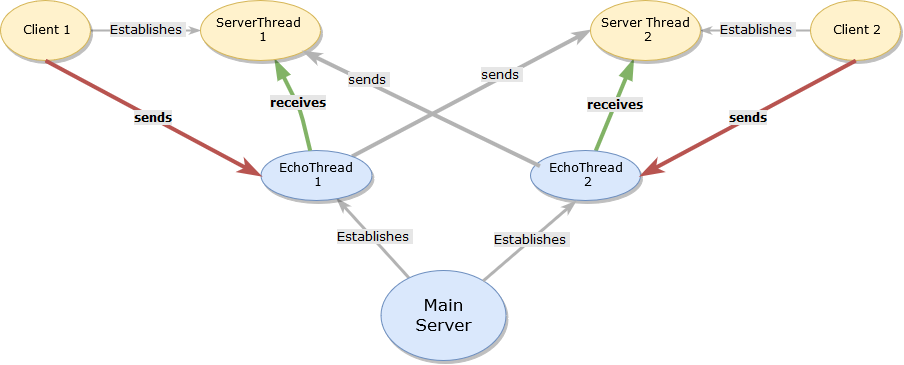
## Prototype 2 - Design

Prior research on Java Secure Socket Extension (JSSE) had shown an alternative to blocking I/O using JSSE's SSLEngine class in combination with Java's “New I/O” or “non-blocking I/O” API known as Java NIO. This alternative was considered as a solution to the problem in Prototype 1 without changing its overall structure. The ServerThread needed to be allowed to skip past the read call if there weren't any incoming messages or be notified if there were. In NIO, the OS is queried to see if it is ready for I/O instead of simply demanding that it carry out an I/O system call. The Selector class does this on behalf of SocketChannels that were registered with it. This complex solution becomes more complicated when SSL/TLS is involved.

As we will see in the next section, the TLS protocol is initiated by a multi-step process between the client and server called the “handshake.” A Java SSLSocket performs all the steps in the handshake and all other communications over SSL automatically as soon as bytes are written to its underlying input stream. SocketChannels are used in place of SSLSockets in NIO, and they do not encapsulate the SSL functionality. Instead, an SSLEngine wraps and unwraps each handshake or other payload to be sent by a SocketChannel. Since NIO is asynchronous, the order and timing of each handshake payload cannot be anticipated. All handshake data that is sent and received must be wrapped or unwrapped by the SSLEngine. Then, the appropriate I/O operation is registered with the Selector to indicate readiness. The process is repeated until the handshake is complete.

## Prototype 2 - Implementation

Fortunately, before the NIO server could be implemented the underlying design flaw of Prototype 1 was discovered and Prototype 2 was created instead (see Figure 5 below). Synchronous/Blocking I/O was a problem because the ServerThread was responsible for reading as well as writing to the EchoThread, but the same issue was not necessary to avoid in EchoClient as had been assumed. In other words, it did not matter if EchoClient blocked while waiting for keyboard input because it had no other tasks to complete. The simplest solution was delegating message output to EchoClient and reducing the role of ServerThread to just reading from EchoThread and letting both threads block whenever needed. Since both threads are running concurrently there is no detriment to performance from the user's perspective.

Figure 5: Prototype 2 – Basic Networking Model 

Redesigning the model permits I/O to block when necessary.

# ACCESS CONTROL

The Secure Chat Secure Login Protocol controls access through intricate password hashing in combination with verification of that user's certificate fingerprints. This ensures that the user is accessing the chat server via the device that has his certificate stored on it. The Secure Login Protocol makes use of the JDK's keytool.exe program to extract the certificate fingerprints by running keytool in the JRE. Subsection 'a' describes the interaction of the main classes that transfer data from client to server, to database to server, back to client from server. The remainder of this chapter is organized as follows:

* Subsection 'b' - Database – Design and Implementation
* Subsection 'c' - Authentication through password hashing – Background
* Subsection 'd.' - Password Hashing - Implementation
* Subsection 'e’ - Secure Login Protocol – Implementation
* Subsection 'f’ - Certificate Fingerprints Extraction – Implementation
* Subsection 'g.' - Registering a new user – Design
* Subsection 'h.' - Registering a new user – Implementation

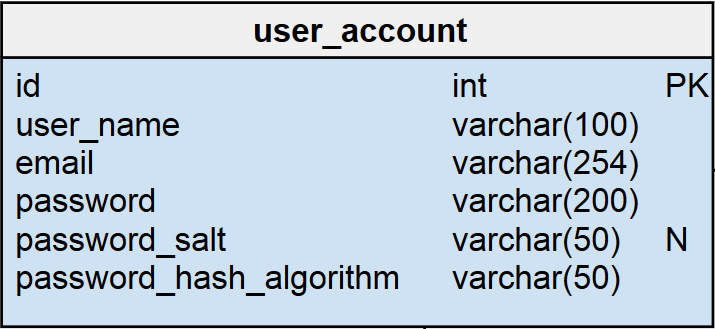
## Secure Login Protocol – Design

The Secure Chat Secure Login Protocol involves direct communication between the UserLogin class on the client side, and the LoginHandler class on the server side. The LoginHandler utilizes the MyJDBChandler class to query the MySQL database. It recruits the SaltHashPassW class and ValidateHashedPassW class for password hash creation and validation respectively.

When the client program starts, a dialog is displayed asking the user if he is “New” or “Returning.” A window appears and the user types in his User Name, Email, Password, Certificate Alias, and Password for Certificate Alias in the appropriate fields. Then a new dialog appears asking for the user to choose to initiate the client-client Diffie-Hellman protocol with the next user who logs in or to respond to a Diffie-Hellman request that was already initiated by someone. After a secured TLS connection is established and the client and server have generated a shared secret key through Diffie-Hellman protocol, they begin the Secure Login Protocol explained in subsection 'e' (Secure Login Protocol – Implementation).

## Database – Design and Implementation

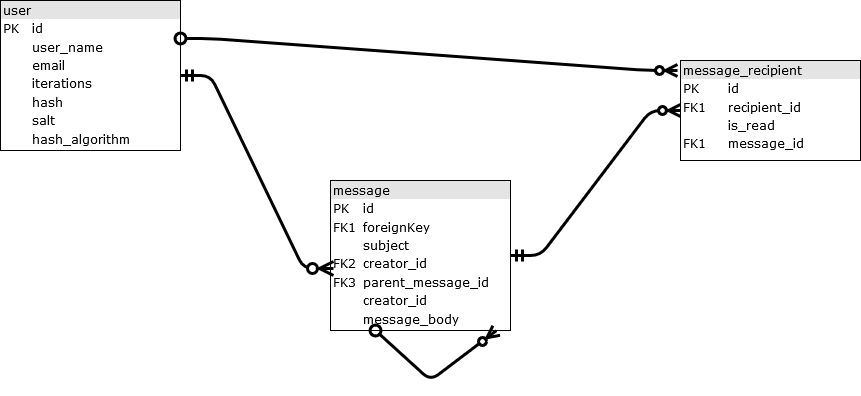
The database was designed to be able to store a user's login credentials, which are the username, email, and password data. If the database is only used to store the information necessary to authenticate a user, then the relational model contains only one entity for a user's profile and login credentials. Initial research suggested keeping this entity simple, including the following attributes: “id” as the primary key, user\_name, email, password, password\_salt, and password\_hash\_algorithm (Kozubek-Krycuń)7 as shown below in Figure 6.

Figure 6: Basic Database Design

Storing the algorithm name allows system to be updated.

Image credit: Kozubek-Krycuń, A., How to Store Authentication Data in a Database.

Further research suggested a more complex relational model that allowed for message persistence, or the ability to retrieve messages from a previous chat session the next time a user logs in. This model would have three entities: user, message, and message\_recipient (Kher)8. The user table would be the same as the previous model's user table. The message table would contain the usual information for a message: id as the primary key, creator\_id, message\_body, and create\_date. In order to create a conversation chain between them, each message would have a reference to the message it is replying to with parent\_message\_id, which would be a self-referential foreign key (See Figure 7 below).

Figure 7: Second database design

This design allows for message persistence (offline storage of messages).

Image Credit: Kher, S., Database Model for a Messaging System.

This design was forfeited in favor of the first model for three reasons. First, the complexity of the database model would require the networking model to be redesigned. It assumes all messages exist only during transport and do not persist after they are sent. The redesign would introduce new requirements for secure message storage and retrieval, encryption of each stored message and access controls to ensure messages are available only to their authors and recipients. Second, message persistence implies that users must surrender their complete ownership of these messages to the database administrator of the server. They can no longer assume that their communications will not be read by a third party or attacker.

In a non-persistent environment, it would be possible to snoop on live communications but the window of opportunity is extremely short, and user controlled end-to-end encryption to prevent spying has been undertaken in this project to prevent that. Third, a more complex schema would lead to a time consuming normalization process. The process usually involves creating new entities to avoid update anomalies. Each new entity might expose vulnerabilities.

The final version of the database model is shown below in Figure 8. The singular entity is `user\_account`. The attributes are `id`, `user\_name`, `email`, `iterations`, `salt`, `hash`, `hash\_algorithm`, and `SHA256\_fingerprints`. The primary key is `id`. The unique indexes are `email`, `user\_name`, and `SHA256\_fingerprints`. The fingerprints are those of a user's certificate. The database was populated with 101 entries, with three initialized for testing. The database was created in MySQL Workbench 8.0 and MySQL Community Server 8.0.

Figure 8: Final database design

Simpler design now incorporates fingerprints.

## Authentication through password hashing - Background

Obviously, the password itself must not be stored. Instead, when a user creates an account a hash of his password must be calculated and stored in its place. When a user logs in, the password he types is hashed using the same method from the first time when his account was created. The hash of the typed password is compared byte-by-byte to the password stored in the database under his user profile. Storing hashes is obviously more secure than storing plaintext passwords, but less obvious than storing encrypted passwords. The latter method has the drawback of relying on an encryption key held by a database admin (Ducklin)9. The key could be compromised by an attacker or a rogue admin. In contrast, since a hash is a one-way function, the plaintext cannot be retrieved by reversing the algorithm.

Attacks on hashes usually exploit collisions. Weak hashing algorithms are considered fallible because they cannot perform in a uniformly random way for all possible values, meaning there is a higher chance that given two different inputs, the same hash can be output. This is known as a collision. An attacker who knows about this vulnerability will perform a brute force attack faster. This attack refers to the method of simply calculating all possible hashes from all possible inputs and looking for a match in the output. Therefore it is called 'brute force attack' rather than 'leveraging knowledge about the algorithm attack.' Since hashing algorithms are not possible to reverse, a brute force attack simply goes forward again and again with all possible inputs. To make the job easier, an attacker can use a “rainbow table” containing pre-calculated hashes to save time. If he is lucky he will find the original password or a collision giving the same result.

According to Ducklin 9, encrypting passwords can lead to another vulnerability: certain algorithms are deterministic, such as DES, meaning the output is not random given the same inputs. Two users with the same password will have the same ciphertext version of the password. Another information leak is that DES output is proportional to the input, which leaves a hint to the length of the original password. Normal hashing produces fixed length output, so it addresses the latter issue, but not the former issue of determinism. Salting addresses both issues. Salting is the practice of adding a random string of bytes to the input data. This greatly reduces the chance of two users who have the same password getting the same generated hash because their salts are randomly chosen. The salt can be safely stored in the database because an intruder who obtained them both still needs the original password in combination with its salt to produce the hash. Salting therefore makes rainbow tables ineffective.

Contemporary computing power is more than enough to quickly perform a brute force attack, even when the hashing algorithm is considered secure like those of the SHA-2 family of hashes. Ducklin points out that “hash-cracking servers that cost under $20,000 five years ago (2012) could already compute 100,000,000,000 or more SHA-256 hashes each second.”9 The point illustrates the need to slow down brute force attacks, making them computationally and financially expensive. To that end complex protocols have been developed for increasing the entropy, or strength, of a hash.

PBKDF2 is an open source algorithm that “stretches” a password by repeatedly using a hash algorithm on the password for several iterations. The number of iterations should be increased each year to keep pace with password cracking hardware. 40,000 iterations were chosen for this project because that was a recommended amount as of August 2017 (Ducklin)9. The hashing algorithm employed by PBKDF2 in this project was HMAC SHA-256.

HMAC SHA-256 works with a key as input in addition to the plaintext data being hashed. In this case the key is a salt and the plaintext is a user's password. First a selection of bits in the salt are flipped in an XOR (exclusive-or) operation. This result combined with the password are used as input to the SHA-256 algorithm producing an intermediate hash. Second, the salt is permuted again by flipping a different set of bits and the result is added back to the first hash produced. The product is fed back into SHA-256, producing the final hash (See Figure 9 below).

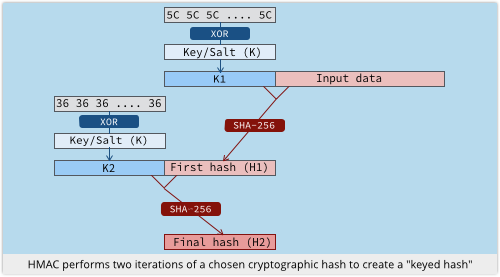
Figure 9: Procedure for HMAC with SHA256

Image credit: Ducklin, P., Serious Security: How to store your users passwords safely.

This algorithm may sound strong enough on its own, but the process described in the last paragraph has only two cycles. So the motivation of PBKDF2 with HMAC SHA-256 is now clear. PBKDF2 provides increased password entropy by iterating HMAC SHA-256 several thousand times (See Figure10 below). This forces an attacker to perform the same number of iterations when trying to crack passwords, and because of the salt he must attack each password one at a time.

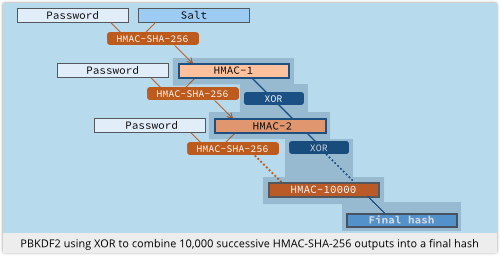
Figure 10: Mechanics of PBKDF2

Image credit: Ducklin, P., Serious Security: How to store your users passwords safely.

## Password Hashing - Implementation

Source code for password hashing and validation was based on examples from Lokesh Gupta, which use the Java Security and Java Cryptography APIs. To create a new password hash, the SaltHashPassW class first generates a 16 byte salt. The salt along with the user's plaintext password are used to create a Password Based Encryption (PBE) Key Spec, which is the underlying material for making the key. The PBEKeySpec is initialized for 40,000 iterations and a key length of 64 bytes. Next, a SecretKeyFactory is initialized for the “PBKDF2WithHmacSHA256” algorithm. Finally, the SecretKeyFactory generates a 64 byte hash based on the PBEKeySpec. This hash and its salt are sent to the server when a new profile is created.

The ValidateHashedPassW class checks the validity of passwords and certificate fingerprints. Using the same methods as the SaltHashPassW class, it generates a test hash from the password supplied by the user and the salt corresponding to the user's profile stored in the database. The test hash is compared to the stored hash by XOR-ing each pair of bytes sequentially and accumulating the differences. Two hashes that have the same length and same sequence of bytes will have a difference of zero. This same method is used to compare the characters of the SHA256 certificate fingerprints.

## Secure Login Protocol – Implementation

The LoginWindow class displays the first three windows mentioned in Subsection ‘a’ when a user starts the client program. The login credentials are passed from the LoginWindow object to the EchoClient object which handles the rest of the client functionality. After establishing an SSL connection and generating a secret key shared with the server, the EchoClient passes the login credentials to the UserLogin object, as well as the socket, the location of and password for the user's keystore, the answer-values to the “new or returning” and “Diffie-Hellman initiator or receiver” questions, and lastly, a userAES object used for encrypting and decrypting.

The UserLogin object will determine whether to register a new user or log in a returning user. If it is a returning user, UserLogin sends the answer-value for “Diffie-Hellman initiator or receiver” to the LoginHandler. Next, UserLogin sends a "RETURNING\_USER" message to LoginHandler to prepare it to receive the login credentials. The UserLogin object encrypts and sends the User Name, email, given password and certificate fingerprints (as described in subsection 'f' below) to the LoginHandler, and waits for a validation response.

On the server side, the socket is passed to the LoginHandler along with the serverAES used to decrypt and encrypt messages. When the LoginHandler is constructed, first it receives the answer-values to the “Diffie-Hellman initiator or receiver” question. Then it initializes a MyJDBChandler object, loads the JDBC driver and establishes a connection to the MySQL secure chat database. Next, it receives the request for either "NEW\_USER" or "RETURNING\_USER." It receives and decrypts the User Name, Email, given password, and certificate fingerprints. The MyJDBChandler object calls searchTable() to prepare a query to search the user\_account table based on given User Name and Email. The query can be constructed in three ways:

1. SELECT id, salt, hash, iterations, hash\_algorithm, SHA256\_fingerprints
2. SELECT id, salt, hash, SHA256\_fingerprints
3. SELECT id, iterations, hash\_algorithm

The 1st query is used for authentication purposes but the 3rd query is for the use case of upgrading the passwords to a new number of iterations or new hash algorithm. The MyJDBChandler executes the specified query and returns the result set as an array of strings where the elements are:

[0] = id; [1] = salt; [2] = hash; [3] = fingerprints [4] = iterations; [5] = hash\_algo;

Finally, the LoginHandler verifies the login credentials by instantiating a ValidateHashedPassW object and passing to it the given password hash, stored hash, salt, given fingerprints, stored fingerprints, and number of iterations used in PBKDF2. The ValidateHashedPassW object validates the given password and given certificate as described above in Subsection 'd.' Password Hashing – Implementation. A boolean value is set true only if both the password and certificate match those in the user's stored profile. An access string of "granted" or "denied" is sent back to UserLogin which will return a true or false value back to the EchoClient. Lastly, EchoClient will notify the user by displaying a message of either: "Access denied. Exiting program" or "Access granted. Welcome back " + User Name. If access was granted the client moves on to the client-client Diffie-Hellman Protocol. If access is denied or the user is registering a new profile, the program closes immediately.

## Certificate Fingerprints Extraction – Implementation

Thefingerprints of a peer's public certificate can only be found by using the -list command of Java keytool.exe. In much the same fashion as this program uses keytool -genkeypair by running it in the JRE when a new user is registered, the chat client program calls keytool -list to print out all details of the certificate including its SHA256 fingerprints. The source code was based on the examples provided by Alvin Alexander who built it off of Michael C. Daconta's examples.

The ProcessBuilderExample class is the driver class which takes a list of Strings containing the command to run and some parameters. Depending on the supplied command, it adds the remaining parameters to the list and passes the list to the SystemCommandExecutor. ProcessBuilderExample calls executeCommand() on the SystemCommandExecutor and waits for the Standard Output (StdOut) and Standard Error Output (StdErr) to be returned and prints them.

SystemCommandExecutor identifies which of the three keytool.exe commands are to be executed: -genkeypair, -list, or -certreq. In this case it is -list, and so after setting up and starting the process, it initializes two ThreadedStreamHandlers to process the output, one for StdOut and one for the StdErr as mentioned above. The ThreadedStreamHandlers are identified as inputSteamHandler and errorStreamHandler. The inputSteamHandler gets the input stream of data coming from the process' StdOut and the errorStreamHandler gets the error stream of data coming from the process' StdErr.

Both ThreadedStreamHandlers read and buffer every line of output and error output from the process and append these strings to StringBuilders. The StringBuilders are passed back to the SystemCommandExecutor which returns them to the ProcessBuilderExample where they are printed out to the user.

Keytool.exe -list prints out a long list of details about the certificate. In order to retrieve the fingerprints of the certificate from the output, a new class called KeytoolListCertSH was created, inheriting the ThreadedStreamHandler class. KeytoolListCertSH filters the output from the -list command and searches for the keyword “SHA256.” The line of text following that keyword has the fingerprints of the certificate in hexidecimal format. This data is passed back to the ProcessBuilderExample object and finally, returns to the UserLogin class where it is encrypted and sent along with the other login credentials.

## Registering a new user – Design

The Secure Chat Register New User protocol design and implementation are very similar to the Secure Chat Secure Login Protocol for existing users. First, a hash of the new password and its salt are created. Second, a certificate is created by calling Java keytool.exe in the JRE using a protocol similar to the Certificate Fingerprints Extraction protocol described in Subsection 'f.' A class called KeytoolStreamHandler that inherits ThreadedStreamHandler was designed specifically for the -genkeypair command to make new RSA certificates. After the salt, hash and new certificate are made, the new user's login credentials are sent to the LoginHandler and inserted into the database. Lastly, the new user is forced to quit the application. Offline, the user proves his identity to the Server CA so it can sign his certificate before the new user can log back in. Once it is signed, the certificate must be added to the server TrustStore as described in Chapter III, Subsection ‘c.’

## Registering a new user - Implementation

When the client program is launched, the user clicks “New” and this choice is conveyed to the UserLogin and LoginHandler. The user enters their new credentials at the Login Window but the certificate alias and password must already exist in the system. Otherwise, the TLS connection will not be made. In the UserLogin class, a "NEW\_USER" message is sent to the LoginHandler. Then, a SaltHashPassW object is used to generate a password hash with 40,000 iterations as described in Subsection 'd.'

To make a new certificate, the -genkeypair command is supplied to the ProcessBuilderExample and the SystemCommandExecuter uses a KeytoolStreamHandler to run the command. When keytool -genkeypair is run, it asks the user a series of questions to fill out the details of the new certificate owner's identity such as “What is the Common Name?” “What is the Organization?” “What is the two-letter country code?” KeytoolStreamHandler anticipates these questions and sends the answers that were supplied by the user to the running keytool process. Keytool stores the new self-signed certificate in the home directory. The fingerprints are extracted from the new certificate the same way as with an pre-existing user. The new salt is sent first, then User Name, Email and Password hash and fingerprints are sent to the Login Handler in the same manner as described in Subsection 'e.' Finally, the MyJDBChandler inserts the new user credentials and associated data into the database

For practical reasons, one step in the process of signing certificates for testing purposes could not be executed by the program itself. Namely, calling keytool.exe -certreq through the JRE was not achievable during production, because any attempt produced the Error Message: “Cannot create .csr file in location C:/xxx/yyy - Access denied.” This may be that the OS does not recognize Administrator Privileges when keytool is executed in the JRE. Using the JRE is simply a convenience to the user and is very platform dependent.

However, for the security design purposes of this project, all certificates must be signed as shown in Chapter III, subsection ‘c.’ Therefore automating this process could lead to security exploits which an attacker could take advantage of to have his certificate signed and trusted. The identity of the person the certificate was created for should be verified thoroughly by a human (system administrator) or else a complex system that could validate identity from external sources.

# CONFIDENTIALITY

Data confidentiality is achieved through symmetric encryption with AES in CBC mode. Session keys, or “ephemeral” keys, are only used for one period of transactions, then discarded. In this way the Secure Chat system provides forward secrecy. They are generated between a client and the server for secure login. New session keys are generated between client and client for encrypting chat messages. All session keys are created with the Diffie-Hellman algorithm. The purpose of regenerating ephemeral keys between clients is to prevent the server from being able to decrypt the users' communications.

## Symmetric AES encryption – Background

AES is the Advanced Encryption Standard, first published in 1998. In 2001 the U.S. National Institute of Standards and Technology (NIST) established it as the standard for electronic data encryption (FIPS 197) 10. AES operates on a fixed-length set of bits known as a “block.” The bits in the block are substituted and permuted to alter their composition. This is the basis for encryption. The AES cipher takes a specified key as input with the plaintext, producing deterministic output. It is a symmetric key algorithm because the same key is used to encrypt as to decrypt.

There are several block cipher modes of operation that allow AES to be used to encrypt multiple blocks of data and increase security. This program uses Cipher Block Chaining (CBC) mode. First, an initialization vector (IV) comprised of a random sequence of bytes is created and XORed with the first block of plaintext (P1). The output is fed into the AES cipher and encrypted with the key. This produces the first block of ciphertext (C1) which is XORed with the second block of plaintext (P2), and once again, fed back into the cipher to produce C2. The process repeats with each block of plaintext being XORed with the previous ciphertext block before being encrypted.

One of the weaknesses of CBC is that each block of ciphertext is dependent on every block of plaintext that was processed before it. An attacker can change one bit of a plaintext block or the IV to affect all following ciphertext blocks. CBC also must use a padding scheme to increase the length of the plaintext to a multiple of the block size.

## Symmetric AES encryption – Implementation

An AES class was created to facilitate AES encryption and decryption operations. The source code is based off of code provided by Marian Iskander. Each actor (client or server) is given an AES object initialized with the shared secret byte array created through a Diffie-Hellman key exchange that the pair of actors participated in. The 16 bytes of the shared secret become the key through the SecretKeySpec object, initialized for “AES.” All AES objects use the cryptographic transformation "AES/CBC/PKCS5Padding," which specifies that the Cipher object will use AES in Cipher Block Chaining mode with PKCS5Padding. This program uses the standardized PKCS5Padding to address the weakness of CBC mode mentioned in the last paragraph.

The AES class encrypt() method takes a string of plaintext and returns an encrypted string in base64 binary format. A Cipher object is initialized with the SecretKeySpec and set for “ENCRYPT\_MODE.” The Cipher performs the encryption on the plaintext and produces a byte array of ciphertext. During encryption, the Cipher produced a random Initialization Vector (IV) internally. The IV is extracted and inserted into the finalData byte array. The ciphertext bytes are appended to the finalData array. Finally, the finalData containing IV and ciphertext together is converted to a base64 binary string.

The decrypt() method of AES is carried out almost in reverse of the encrypt() method. The main difference being that when the Cipher is reinitialized for “DECRYPT\_MODE,” it does not generate and use a random IV. Instead, it must use the same IV that was used to encrypt the message. First, the input string encodedInitialData- which is the output of the encrypt method- is converted from base64 binary to a byte array. The IV is taken from the first 16 bytes of encodedInitialData and the ciphertext is taken from the remaining bytes. After the Cipher is initialized with the SecretKeySpec and IV, the decrypted plaintext bytes are produced and converted to a string.

It is important to note that this implementation of the AES class increases the strength of Confidentiality protection by forcing the Cipher object to be re-initialized each time it is used to encrypt. Doing so generates a new IV for each encrypted message. If this system relied on a static Cipher object and static IV, an attacker could more easily compromise all messages' confidentiality or integrity if he was able to retrieve the IV.

## Diffie-Hellman key exchange – Background

As seen on Wolfram.com, “The Diffie-Hellman protocol is a method for two computer users to generate a shared private key with which they can then exchange information across an insecure channel.”11 To demonstrate we'll use the most common example names in cryptography: Alice and Bob.

First they agree on two prime numbers 'p' and 'g', where p is very large and g is a primitive root modulo 'p.' These prime numbers can be publicly disclosed. Second, Alice picks a large, random number 'a' for her “private key” and Bob similarly chooses a number 'b' for his “private key.” Third, Alice computes her “public key”: 

Alice sends her “public key” A to Bob. These are not keys like in asymmetric encryption, but the names are used to emphasize what is kept private and what is sent publicly or “in the clear.” Meanwhile, Bob similarly computes his “public key”:



Bob sends his “public key” B to Alice.

After both parties receive the “public keys,” they can compute the same secret shared key K as:



Alice takes B and computes the shared key K thus:



Bob takes A and computes K similarly:



This shared key K is used for symmetric encryption with AES. During this protocol, an eavesdropper named Eve can collect g, p, A, and B.

To get K, Eve would have to compute:  without knowing 'a' or 'b.'

To get 'a' she must find the Discrete Logarithm of: 

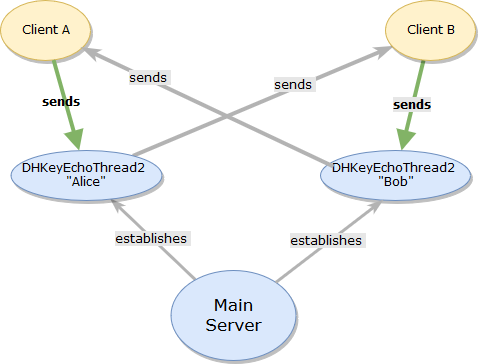
which is computationally infeasible to do for very large prime numbers.

## Diffie-Hellman Protocol – Design

Following tradition and the example code in Oracle's JCA Reference Guide4, two classes were created for the Diffie-Hellman two-way key exchange: DHKeyAlice for initiating the protocol, and DHKeyBob for responding to the request. These are the two primary classes for sending the DH prime numbers and generated DH “public key” numbers to each other, as well as all other computations involved.

There are two use cases for applying the Diffie-Hellman (DH) protocol in this program: client-server and client-client. In both use cases there is one party acting as “Alice” using DHKeyAlice and one party acting as “Bob” through DHKeyBob. The former use case is simpler because both sides have a direct connection to each other. The latter involves a system similar to the Basic Networking Model of Prototype 1 (see Figure 11). It works by relaying messages from DHKeyAlice to DHKeyBob through a class called DHKeyEchoThread2. The naming convention stems from the following: DH stands for Diffie-Hellman, EchoThread is the class it inherits, and “2” because it facilitates the DH protocol between 2 parties only.

The JCA API can be used to arrange a DH key exchange between three or more parties, but for the sake of time constraints this project has only implemented a two-way exchange. The protocol incorporates RSA signatures to protect the integrity of the DH parameters that are exchanged. This will be explained in chapter VII subsection 'a.'

Figure 11: Diffie-Hellman Networking Model 

Similar to Basic networking model of Prototype 2

## Diffie-Hellman Protocol – Implementation

### Client-Server

The JCA API encapsulates the majority of the work involved in the DH key agreement and exchange. In the client-server DH key exchange, the steps are as follows:

1. The server instantiates a DHKeyAlice object and writes the integer '1' to the client's DHKeyBob in order to kick start the TLS handshake.
2. The two sides exchange RSA certificates for signature verification (explained in Chapter VII Subsection 'a'
3. DHKeyAlice generates a KeyPair object containing a DHPublicKey and DHPrivateKey. They both have copies of the 'p' and 'g' prime numbers. The DHPublicKey has a 'y' value that is analogous to the “public key” 'A' value mentioned in the previous section. It is 256 bytes long. The DHPrivateKey has an 'x' value analogous to the “private key” 'a' value. Its length is 1024 bytes.
4. A KeyAgreement object is initialized for DHKeyAlice with the 'g,' 'p,' and 'x' values.
5. DHKeyAlice sends an X509 encoded byte array of the “public key” 'y' value (with 'g' and 'p') to DHKeyBob along with a signature of the value. Then DHKeyAlice waits for the encoded “public key” from DHKeyBob. The encoded byte arrays are usually between 812 and 814 bytes long.
6. DHKeyBob receives the X509 encoded “public key” bytes and signature. The signature is validated. If it is valid the process continues, if not the process ends.
7. DHKeyBob converts “public key” bytes to a PublicKey object. It contains the 'g' and 'p' prime numbers and the 'y' “public key” value. The 'g' and 'p' primes are extracted and given to a DHParameterSpec object. This is used to initialize the KeyPair object containing a DHPublicKey and DHPrivateKey for DHKeyBob (as in step 3 for DHKeyAlice).
8. A KeyAgreement object is initialized for DHKeyBob (as in step 4 for DHKeyAlice).
9. DHKeyBob sends its encoded 'y' value to DHKeyAlice along with a signature of 'y.'
10. DHKeyAlice receives the X509 encoded “public key” bytes and signature. The signature is validated. If it is valid the process continues, if not the process ends.
11. DHKeyAlice creates the “public key” of DHKeyBob as seen in step 7.
12. DHKeyAlice calls doPhase() on its KeyAgreement object. Finally, the KeyAgreement object can generate the shared secret. The length of the shared secret is sent to DHKeyBob.
13. Once DHKeyBob receives the length of the shared secret generated by DHKeyAlice (256 bytes), it can allocate space for a byte array to hold the shared secret. Finally, DHKeyBob independently generates the same shared secret (as in step 12).

### Client-Client

As mentioned in Subsection 'd.' of this chapter, Diffie-Hellman Protocol – Design, making a shared key between two clients is the second use case for using the Diffie-Hellman protocol. The dynamic changes dramatically in the second use case. The order of the specific data transfers is planned out meticulously and must proceed synchronously.

Recall from Chapter V Subsection 'a,' after submitting login credentials to the LoginWindow, the 1st user chooses to initiate the Diffie-Hellman Protocol by clicking “Yes, wait for correspondent.” Once the 1st user is granted access, the server initializes a DHKeyEchoThread2 by passing itself, the socket connected to the client, and a boolean value “initiator” set to 'true.' It starts the thread and adds it to its DHThreads list. Because “initiator” is set to true, the DHKeyEchoThread2 enters the alice() method of the thread and immediately calls “wait” on a mutex (mutual exclusion) lock object.

The mutex lock blocks the progress of the thread. A 2nd user starts the client program, submits their login credentials to the LoginWindow, and must choose “No, I am 2nd correspondent” to receive the DH request after successfully logging in. The server initializes a DHKeyEchoThread2 in the same way except the “initiator” value is set to 'false.' Because “initiator” is set to 'false,' the DHKeyEchoThread2 enters the bob() method of the thread and immediately calls “notify” on the mutex lock object. Doing so releases the mutex lock and allows the thread executing alice() to continue. The “initiator” client creates a DHKeyAlice object and the 2nd client creates a DHKeyBob. They use them the same way as described above in Subsection 'e'-i to create and send the DH parameters to their respective DHKeyEchoThread2s.

At the beginning of the alice() method, the DHKeyEchoThread2 iterates through the server's DHThreads list to obtain the reference to the 2nd correspondent, “bob” DHKeyEchoThread2. The “alice” DHKeyEchoThread2 uses this reference to access the output stream of the “bob” thread. The “alice” thread will use this output stream to transmit the DH parameters directly to the “bob” client. In the same way, the “bob” thread communicates directly to the “alice” client. The result is a new DH key for clients to encrypt all chat messages for the session.

Two regression faults were encountered in this system. After incorporating the integrity protocol for messages, initiator client (“alice”) exited the DHKeyEchoThread2 before the receiving client (“bob”) could obtain all DH parameters. After Phase One, the “alice” DHThread no longer needs to wait for any other parameters from Bob and exits the thread, passing the socket to the EchoThread to begin chatting. This left the “bob” client waiting to receive the length of “alice's” output to complete the key generation, but instead it received a negative error value. To remedy this a second mutex lock was added to the end of both the communication methods alice() and bob() in the DHKeyEchoThread2. This forces the “alice” DHThread to wait until the “bob” DHThread notifies it has finished.

After modifying DHKeyEchoThread2 to accommodate the RSA signature exchange protocol, the second regression fault was encountered in this system. When Bob tried to execute Phase 1 and generate the shared secret, a ShortBufferException was thrown. This indicated another thread timing and concurrency issue. Most likely the DHKeyAlice client completed the protocol early and passed its socket to the chat room method thus breaking the connection before DHKeyBob can receive the data.

Since these server-side proxy threads only represent the output of their client to the other client, they must also be synchronized with their clients. They are not “aware” of the progress of their clients. Therefore, the DHKeyBob client needed to signal to its proxy when the DH exchange would be finished. The solution was to have DHKeyBob write '0' as its signal and have its proxy read it.

# INTEGRITY

The third major goal of this project is protecting the integrity of data and chat messages. This does not mean preventing manipulation of the data in transit, but instead detecting such manipulation. To achieve integrity there must be some form of proof that a data transmission came from the legitimate sender.

As referenced earlier, at the beginning of the Diffie-Hellman protocol both peers exchange RSA certificates. The certificates contain the RSA public keys used to verify the signatures of DH data that were made with the corresponding RSA private keys. This prevents a “man in the middle” (MITM) attack wherein a third party intercepts messages from Alice to Bob and relays his own messages to Bob impersonating Alice. Any such attempt would be detected by the Secure Chat RSA Signature Verification Protocol.

Chat messages between clients are protected from MITM attacks using Message Authentication Codes (MACs). A type of MAC that requires a cryptographic key known as an HMAC is generated for each chat message and verified by the other client. The key used to make the HMAC hash is the same Diffie-Hellman shared key created between the clients, which means an outside party who does not have the key could not forge the message hash.

## RSA Signature Verification Protocol for DH – Design

Chris Alexander and Ian Goldberg12 examined a secure instant messaging chat plug-in called Off-The-Record (OTR) Messaging. In their paper, “Improved User Authentication in Off-The-Record Messaging,” they analyzed how OTR used long-lived public/private key pairs to sign data exchanged during DH key generation. This is actually a recreation of the Diffie-Hellman Ephermeral (DHE) protocol used in TLS. It is called ephemeral because the keys that are generated through the process are only used for one session, then discarded. This practice is known as Forward Secrecy. Long-lived asymmetric key pairs are used to guarantee authenticity of the identity of the participants in DHE.

Oracle example code from the Java Tutorial: “Lesson: Generating and Verifying Signatures.” The main classes are GenSig and VerSig and so those names were retained for this project. Instead of creating new PublicKey/PrivateKey objects each time the original source code is run, the strategy was modified to exchange the persistent signed certificates. As mentioned in Chapter VI Section ‘e. - Diffie-Hellman Protocol – Implementation,’ on step 2. of the Client-Server DH Key Exchange, both parties trade RSA certificates for signature verification. The same way a certificate is signed by a Certificate Authority, each party encrypts a hash of the data with their private keys to make a signature. The original data and the signature are sent to the other party. The receiver verifies the sending party's signature using the public key extracted from the sender's certificate to decrypt the hash. The receiving party generates a new hash of the original sent data, and the decrypted ‘signature’ hash is compared to the new hash. If these match then the sent data is confirmed to have come from the expected sender.

The long-term certificates used are the same ones that clients and the server exchanged for SSL/TLS authentication. It is this fact that implies that the certificates belong to the parties they represent, otherwise the mutual authentication in TLS would fail. This is the safeguard for authentic certificates between client and server. There is a second safeguard for the second use case of client-client DH exchange, which is by this point both clients have passed the login process which requires the fingerprints of the user's certificate. If the users do not trust the server, they can exchange their fingerprints directly and compare them to those obtained from their correspondent offline.

## RSA Signature Verification Protocol for DH Parameter Exchange – Implementation

The GenSig class is initialized with the participant's private key and a Signature object is set for "SHA256withRSA." The private key is passed to the Signature object and for the remainder of the DH exchange it is used to generate all signatures on the participant's behalf. The public key obtained from the correspondent is used to initialize The VerSig class and a Signature object is set for "SHA256withRSA." For the remainder of the DH exchange, it is used to verify that all signatures came from the correspondent. The above steps take place after Step 3 of the DH protocol outlined in Chapter VI, Subsection 'e'-i.

In Step 5 of the DH protocol, “Alice” uses the GenSig object to create and send a signature by calling its sendSignature() method and passing it the output stream and bytes of dataToSign. The Signature object is “updated” with the data and creates the signature. In Step 6 “Bob” receives the signature, calls the method verifySignature() on the VerSig object and passes his input stream and data to be verified. The Signature object is “updated” with the data and verifies the signature. VerSig returns a boolean value indicating the result.

## HMAC Integrity Check Protocol for Chat Messages – Design

The protocol for integrity checking of sent messages was also based on the OTR project as discussed by Chris Alexander and Ian Goldberg12. This project uses the HMAC SHA-256. Recall from Chapter V Subsection 'e' that HMAC SHA-256 was used in combination with PBKDF2 to generate hashes of passwords. In that instance HMAC takes a salt as input with the password. In the message integrity use case, the salt is replaced by the DH key generated between clients. A MITM could not forge the hash unless he had access to the shared DH key.

The Message class encapsulates the ciphertext and the HMAC hash of the ciphertext. The static method computeHash() of the Message class takes the ciphertext message and an initialized AES object as input. Recall that the userAES object contains the shared DH key. The computeHash() method was made “static,” or universally accessible, to make sure hashes are calculated and verified outside the Message objects. This keeps the secret key from being exposed while the Message object is in transit, by uncoupling the AES object from the Message class.

After it is sent, the receiver calls compareHash() to calculate a new hash and compare it to the original hash stored in the Message to prove that the ciphertext was not modified in transit. The comparison is performed in the same manner as a byte-by-byte password comparison as explained in Chapter V Subsection 'd. - Password Hashing - Implementation.'

## HMAC Integrity Check Protocol for Chat – Implementation

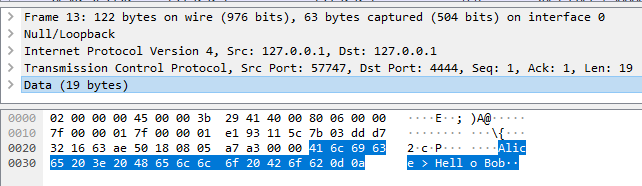
When the user presses the “send” button, EchoClient encrypts the message and passes the ciphertext and userAES to the computeHash() method. The resulting byte array, called “cipherhash,” is placed in a Message object with the ciphertext and sent to the client's EchoThread. EchoThread does not open or analyze the contents of the message, it only forwards it to the ServerThreads of the communicating clients. When the ServerThread receives a Message it calls the compareHash() method and passes the received ciphertext and the client's userAES object containing the DH key to the method. If the comparison fails, ServerThread prints out a message to the user, "Message was tampered with in transit!"

# VALIDATION

This chapter demonstrates the efficacy of the cryptographic at the application level by displaying screenshots from WireShark packet captures. It will also document the effects of TLS.The application data is highlighted in blue. In the highlighted section, the left column has the hexidecimal output and the right column has the ASCII translation. The reader will notice encrypted data looks like a random, nonsense sequence of characters.

## WireShark Results

### Prototype 1 – without TLS and without encryption



The un-encrypted message from client at port 57747 to server at port 4444 says “Alice > Hello Bob”

### Prototype 1 – without TLS and with encryption

The encrypted message from client at port 57745 to server at port 4444 is unintelligible.

### Final Version – with TLS and encryption

The message sent through a TLS connection has been encrypted by TLSv1.2

# FUTURE WORK

The scope of this project was effectively limited by the nature of its design and for intentional reasons. Improvements to this project should address the scalability, portability, and convenience when it does not conflict with security.

A chat room with multiple participants would not be able to communicate with each other in this project. The DH two-way system only allows a pair of correspondents to make a secret key and thus they can read each other’s messages but other participants in the chat room would see just the ciphertext of their messages. For example, Alice and Bob are speaking English in a chat room they share with Ling and Hong who only speak Chinese. They all have access to the chat room but their conversations are not mutually intelligible. Developing a DH protocol for 3 or more participants is a future goal of this project.

Regardless of the 2-person DH limitations, it is still a Chat room, so all participants must be part of the same organization and trust each other. E.g., they all have the same level of clearance. Since participants cannot block certain people from entering the chat room, their conversation would not be completely private, they would be open to other people within the trusted organization. This could be improved with private chat sessions in addition to chat rooms.

Other scalability issues include the lack of Publish-Subscribe or Presence-Notification and Lookup systems. Most large scale instant messaging systems allow users to find other users and know when they are online. That way, they could choose who to send a message to instead of entering a chatroom and speaking to whomever enters it next. Such systems introduce a host of security concerns. Previous work on this subject was covered intensively by Malgorzata Wrzesinska in his master's thesis “A Secure Instant Messaging System.”13 Future work on this project should develop such systems.

There are also technical reasons this project is not scalable, such as the one-thread-per-client networking model. To keep CPU overhead down, thread pooling should be used, or perhaps the NIO alternatives discussed in Chapter IV Subsection 'c.'

Extracting fingerprints using the keytool -list command in the JRE is provided as convenience to the user, but is platform dependent. Since fingerprints are treated essentially as a secondary password, the responsibility of storing, retrieving and verifying them could be left to the users, but more portable solutions should be researched.

In some security environments the certificate file is external to the platform of the client program. In such cases it could be stored on cryptographic tokens such as hardware security modules and smartcards. This program assumed that the user manually installed a certificate file on his device or platform and configured the client program to locate that file upon login when he supplies his certificate alias and password. For testing purposes the file location was hard-coded into the software. Upgrades to this system could include interfaces for hardware devices to store and read certificates. Doing so could also convenience and allow for portability to other platforms.

In the DH Key Exchange protocol, the two parties exchange RSA public certificates. As described at the end of Chapter VII section ‘a,’ each party must implicitly trust the identity of their correspondent since the server authenticated the correspondent and his certificate. However, assuming that all participants import the certificates of their correspondents into their TrustStores, an extension to this software could allow a participant to match his correspondent’s certificate against the list of known certificates in the TrustStore to verify a correspondent’s identity.

# 

# CONCLUSION

The Secure Chat system demonstrates the use of strong cryptographic features to achieve access control, confidentiality and integrity in an instant messaging system. This paper proved that a client-server architecture can be maximally secured when all users are pre-identified and user profiles are tied to certificates. Certificates are involved in every security feature of this system: TLS, the Secure Login Protocol, RSA Signatures Verification for DH Protocol, and HMAC Integrity Check for Messages Protocol.

Because certificates are strongly tied to real-world identities, it makes it almost impossible to deny a persona or act anonymously. This is apparent when certificates are exchanged at the beginning of the TLS protocol. Since there is no protection against traffic snooping at this point, an observer can learn who is connecting to the server and at what times. That is why this product was developed with the military in mind as the customer, or some other small organization that requires utmost security, but whose members are all known and vetted within it.

The compromise between anonymity and real identity favors the latter when security is the highest priority. This is due to the strength of RSA certificates. It is no coincidence that RSA has been in use since 1977. RSA is not convenient for standalone encryption, it is slower than AES and its encryption capability is deterministic, so it becomes easier to break given a pattern of ciphertext produced by the same key pair over a long period of time.

Instead, RSA’s strength comes from authentication, which is strongest when used mutually between client and server. Since by its design a client-server architecture weak point is the centralized server, mutual authentication is crucial. The server cannot blindly trust any client that connects to it. Strong identification takes the burden of security off the server and onto the client.

This paper also identified the balance between security and convenience. The certificates must be signed by the Certificate Authority offline. This is far from convenient to the user. It was not practical to automate in this project (see Chapter V, Subsection ‘h.’), but those issues aside, signing a certificate probably should not be automated. In real-world security scenarios, the verification of a client's identity is done manually between the client and the Certificate Authority before the certificate is issued. The client must provide physical proof of identity to the CA and doing so creates a solid foundation of trust for the software to lie on.

Another inconvenience is that new certificates cannot be used immediately, they must first be added to server TrustStore, one-by-one (see Chapter III, Subsection ‘c.’). This also presents a scalability issue for the speed of expanding the user base. However, this is a one-time cost per user and the benefit is increased security. As mentioned in Chapter V, Subsection ‘h,’ new users who are completing the Registering Protocol are required to connect to the server with a pre-registered certificate from another user. This implies there is a chain of initiation among users, in which a senior member inducts a junior member. In high-level clearance situations it is easier to trust a smaller group of individuals, and the likelihood of their unauthorized disclosure is minimal.

# REFERENCES

1. Dierks, T., & Rescorla, E. (2008, August). *The Transport Layer Security (TLS) Protocol, Version 1.2*. Retrieved from <https://tools.ietf.org/html/rfc5246>
2. Ramakanta. (2017, January 3). *How To Enable TLS 1.2 Protocol*. Retrieved from <http://www.techpaste.com/2017/01/enable-tls-1-2/>
3. Atighetchi, M., Soule, N., Pal, P., Loyall, J., Sinclair, A., & Grant, R. (2013, December 12). Safe configuration of TLS connections. *IEEE, Conference on Communications and Network Security,* 415-422. doi:10.1109/CNS.2013.6682755
4. Java Cryptography Architecture (JCA) Reference Guide, <https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html>
5. Weaknesses and Alternatives. (n.d.). Retrieved from <https://docs.oracle.com/javase/tutorial/security/apisign/enhancements.html>
6. Transport Layer Protection Cheat Sheet. (n.d.). Retrieved from <https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet#Rule_-_Only_Support_Strong_Cryptographic_Ciphers>
7. Kozubek-Krycuń, A. (2015, July 9). How to Store Authentication Data in a Database. Part 1. Retrieved from <http://www.vertabelo.com/blog/technical-articles/how-to-store-authentication-data-in-a-database-part-1>
8. Kher, S. ( 2015, October 29). Database Model for a Messaging System. Retrieved from http://www.vertabelo.com/blog/technical-articles/database-model-for-a-messaging-system
9. Ducklin, P. (2013, November 20). Serious Security: How to store your users passwords safely. Retrieved from <https://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/>
10. Announcing the ADVANCED ENCRYPTION STANDARD (AES) *. Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST).* November 26, 2001. Retrieved October 2, 2018
11. Diffie-Hellman Protocol. (n.d.). Retrieved from <http://mathworld.wolfram.com/Diffie-HellmanProtocol.html>
12. Alexander, C., & Goldberg, I. (2007). Improved user authentication in off-the-record messaging. *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society - WPES 07*. doi:10.1145/1314333.1314340
13. Wrzesinska, M. (2002, June). A Secure Instant Messaging System. Retrieved from http://students.mimuw.edu.pl/SR/prace-mgr/wrzesinska/thesis6.html

# 

# SOURCE CODE CREDIT

Java Secure Socket Extension (JSSE) Reference Guide, https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

User “CanadianLobster” https://www.instructables.com/id/Creating-a-Chat-Server-Using-Java/

User “fliedonion” https://gist.github.com/fliedonion/1002293af6fd043fbd6e729c13018562

Lokesh Gupta, <https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/#PBKDF2WithHmacSHA1>

Alvin Alexander, <https://alvinalexander.com/java/java-exec-processbuilder-process-3>

Michael C. Daconta, <https://www.javaworld.com/article/2071275/core-java/when-runtime-exec---won-t.html>

Marian (Iskander) Zaki, [jafarit@uhd.edu](mailto:jafarit@uhd.edu) , iskanderm@uhd.edu

Java Cryptography Architecture (JCA) Reference Guide, https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html #DH2Ex

# ACKNOWLEDGMENTS

I'd like to thank the following people and organizations for their help on this project:

Dr. Yuan for his knowledge and guidance as my project mentor.

Professor Iskander for introducing me to the subject of Network Security, and for the assignment and source code this project was based on.

All the developers who published example source code online (see section XI).

The community at StackOverflow.com. So many of my questions had answers on these forums.

The Oracle company and its Java team for all the libraries, APIs, and for their tutorials and examples on how to use them.

My parents for supporting me and listening to me drone on about network security and programming topics.