Computing and Network Security: Instant Messenger

Matthew Kramer, Michael Music, and Sterling Price

Abstract

The purpose of this project is to create a secure instant messaging application. As specified in the project guidelines, the requirements of our application were to implement point-to-point communication between two clients, encrypt messages using a 56-bit key, an optional graphical user interface, and an optional key management protocol. We made the decision to meet these requirements and add both the additional features specified and those that we felt made the application more realistic. As such, we implemented a simple and secure authentication, encryption, and key exchange system that is managed by a central server that communicates with connected clients. We also satisfied the optional requirements by developing a key management system that is integrated into our remotely hosted database. Throughout the introduction of these features, our team routinely analyzed potential attack vectors and appropriately secured our system to counter these attacks.

Introduction

Applications which involve the transfer of messages between two parties across a potentially compromised medium has long been the subject of research and development. The networking systems used to transfer the data, the nature of the client and server within this system, and the cryptographic functions used to keep the messages confidential must be chosen carefully so as to not compromise the communication parties. As such, those three areas were the main focus during the development of our secure instant messaging application.

Before undertaking this project, the members of our team had studied a myriad of topics in cryptography and were aware of the uses of both symmetric and asymmetric cryptography, as well as hashing and how it pertains to securing communication. However, none of us had practical experience in implementing these schemes in fully networked applications. As such, studying and implementing the authentication, networking, and cryptographic systems used within this application proved to be an invaluable learning experience. In particular, our team gained knowledge in programming of database authentication systems, key exchange protocols, Triple-DES encryption, and the construction of custom application layer protocols. We also gained an enhanced security mindset through the constant consideration of potential attack vectors of our application.

Considerations

The primary consideration in the initial design phase of our application was that of the communication model itself. In this regard, we found that there were really only two possible choices: peer-to-peer and client-server. Due to our desire for database-backed authentication, a vision of having an application that allows for multiple clients to chat simultaneously, and the need for a robust key exchange scheme to match, we selected the client-server model.

After communication, our team then considered the interface our application would have with the user: in the end, the choice lay between a command line interface and a graphical user interface. Due to our early agreement on the use of Python as our primary programming language for the application, we made the quick decision to write a command line based application. This decision was largely made due to issues with multithreading and some of the more common Python GUI libraries like TkInter. Simply enough, the sheer number of threads running in our program eliminated a GUI as a potential option.

Our next consideration was authentication. While not required, our team considered using a remote database to store password hashes and other useful data for use in authentication within the application. We decided upon the use of such an authentication system, which would exist mostly on the client side of our instant messaging application, after successfully testing a remotely hosted database.

Computing and Network Security: Instant Messenger

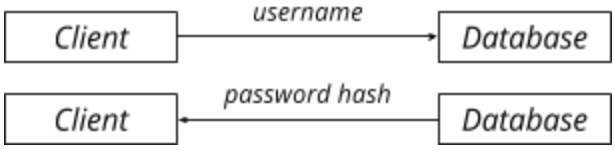
Matthew Kramer, Michael Music, and Sterling Price

A final and very noteworthy consideration that we had to make concerned the key exchange protocol that we were to use. Our program featured several iterations of custom key exchange protocols, starting initially with a plaintext key exchange before evolving into an XOR-based scheme in which the user's password was used to generate a key for both the client and server. This was used alongside the XOR function to encrypt client-server messages for the key exchange. However, in the end, the security flaws in this scheme were simply too apparent and led to the implementation of a Diffie-Hellman key exchange.

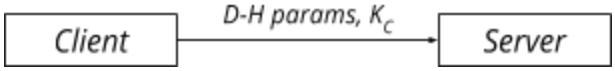
Cryptographic Scheme

The central and most important feature of our chat application is its cryptographic scheme. While daunting when examined in total, it may be better understood when broken down into the different processes used to accomplish its goals.

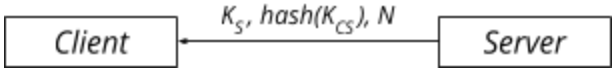
User Authentication: When the client application is first started, the user is prompted for a username and password. Once provided, the database that backs our application is queried for the password hash that corresponds to the username entered. This password hash is then compared against the provided password, once hashed using the **`bcrypt`** Python module, whose password hashing function is based on the Blowfish cipher. If no entry is found for this username, a new entry is created and the password entered is salted and hashed using the same function. Only once a user has been authenticated will the client application allow a connection to the server, meaning that any valid communication the server receives must be from an authenticated user.



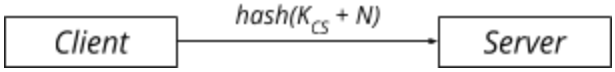
Client-to-Server Connection: Once a user has been authenticated, the client then attempts to connect to the server. Provided that they connect successfully, a Diffie-Hellman key exchange begins. The client first selects the generator and prime group before generating a public and private key for use in the exchange. Once complete, it then sends the parameters it selected along with the public key it generated to the server.



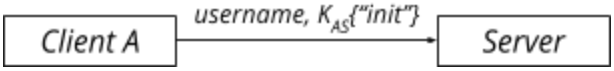
On the server side of the application, the message is received and the parameters extracted before generating a public and private key, just as the client did. This key pair is mapped to the username that sent the Diffie-Hellman parameters, so that it may be referenced in the future in order to encrypt and decrypt messages. Once established and stored, a shared secret is generated and used to create a session key between the client and server. This session key is then hashed using SHA-512 and sent to the client along with a randomly generated nonce and the server's public key.



The client may now generate the same shared secret and session key using the public key that the server sent. To verify the key and provide authentication to the server, the client compares the hash that the server sent with a newly calculated hash of the key that was just generated and the nonce that the client received from the server. If they match, the client then sends a final message to the server containing a hash of the session key concatenated with the nonce. This final message serves as confirmation that the key was generated correctly and authentication of the client that generated it.



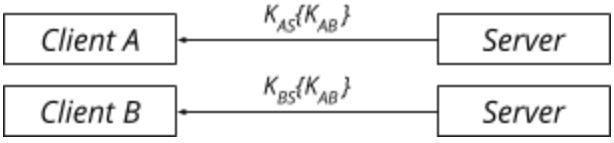
Client-to-Client Communication: Once a key has been established between the client and server, all messages sent between them are encrypted using it. However, this does not mean our application is ready to send messages between clients. Assuming that our application operated in this way, the content of messages sent between clients would be left momentarily unencrypted on the server between when it was decrypted when received from the sender and re-encrypted when sent to the destination. This means that the server would be able to read all content being sent between clients: a vulnerability and intrusion of privacy that we decided to eliminate. To do so, we establish a key between the two clients from the server side and send it to them encrypted with the key that was established in the previous step. It is important to note that this key is never stored by the server; doing so could compromise the integrity of any of the messages that passed through the server. The client key exchange operates in the following scheme: first, a client sends an initialization message containing the username of the client they wish to communicate with encrypted with the key



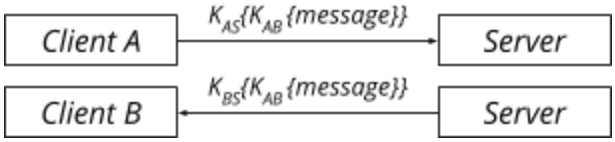
shared with the server.

The server then checks to see whether this user is online and connected to the server. If they are, a new connection is established and the thread handling key assignment will generate a new key for this conversation and send it to both clients.

These two clients may now begin sending



messages to one another using the keys that they have received from the server. These messages, as previously mentioned, are encrypted twice to prevent the server from seeing the messages being sent.



Development and Implementation

While the cryptographic scheme that we developed looks nice on paper, it was, of course, more challenging to implement in the code. Our first iteration of the application was a simple multithreaded server and client messenger without the complicated cryptography. In other words, a client and server program that could connect and communicate, but whose messages were sent in plaintext without any key exchange, encryption, or authentication. These programs - **`server.py`** and **`client.py`** - served as the foundation to which we added the layers of the scheme described before.

Server: The server application is initialized by creating a new socket and listening for new connections on that socket. Once initialized, a thread is started that handles the displaying of currently connected users. With this thread started, the server begins accepting client connections and passing them to a newly started thread to handle communication. Received messages are first parsed before being sorted using identifying information stored in the message such as sending user, destination user, and message number. These parameters were devised as part of the application layer protocol necessary for implementing communication between client and server. Once identified, the thread can then perform the necessary functions associated with a particular message.

Client: The client application starts by creating a new socket and attempting to connect to the server through that socket. Once successfully connected to the socket, the client starts a thread to begin receiving messages in a process much like the one used on the server. Then, once

ready to receive communication, the client begins requesting the username of the client to communicate with. After entering a username, the program begins sending initialization messages to the server, requesting a connection to the destination user specified. The client will receive a message indicating whether the user they wish to communicate with is online or offline, and the program will either continue to send initialization messages if offline or start an input thread if online. This input thread handles the input, encryption, and sending of messages to the destination user. With this basic functionality taken complete, we could make the cryptographic scheme our focus.

Authentication: The first step of implementing the cryptographic scheme that we had devised was to set up user authentication. Since we had decided on using a remote database, we chose to create a new database using Amazon Web Services' (AWS) Relational Database Service (RDS). This service allows users to create remotely accessible database instances for storing data conveniently and securely. Since all of our group members had prior knowledge of SQL, we chose to create a MySQL database instance with the best hardware available for the free tier: 20 gigabytes of SSD storage, 1 gigabyte of memory, and a single virtual Intel Xeon CPU. With the database initialized, we could begin creating tables for the data we would need persisted across sessions. In total, we created only three tables: one for storing usernames and password hashes, a second for storing conversation identifiers along with the time that a key was last generated for a conversation, and a final table for the usernames associated with a particular conversation. Once the database was in place, it was then a matter of communicating with the database from the Python application. To do this, we used the **`pymysql`** module, which provides easy-to-use functions for connecting to the database and calling stored procedures that we used to query and modify the tables that we created. We chose to use stored procedures instead of standard client-side queries because they pass the parameters as separate entities, allowing the database to parse the query as query language and treat the parameters as raw data, eliminating the ambiguity that enables SQL injection attacks. These stored procedures are simply functions stored in the database that can be called from the Python code, passing in the necessary parameters. The stored procedures that we developed The functions required for user authentication can be found in **`database.py`**. We thought it best to consolidate the functionality that required access to the database in a single program so that both the client and server could leverage it. The first database functions to be implemented were that of user creation and user authentication. Users are created using a username and password passed from the client application. The password is salted and hashed using the **`bcrypt`** module and passed to a stored procedure in the database. This stored procedure ensures that a user does not already exist with that username before creating a new entry in the database with the new user's credentials. To authenticate a user, the database is first checked to make sure that user exists in the database before retrieving the corresponding password hash. This hash and the password entered by the user who wishes to be authenticated is passed to the **`bcrypt checkpw`** function, which is able to verify if they match. The result of this operation either authenticates the user or notifies the user of an incorrect password.

Encryption: After clients could be authenticated, we tasked ourselves with deciding how to encrypt messages being sent. Initially, we used randomly generated 56-bit keys in combination with Single-DES. However, recognizing the weakness of this method of encryption, we decided to increase the complexity of the encryption and use Triple-DES with 192-bit (24-byte) keys. The functions that are used to perform the encryption and decryption of messages were developed by Todd Whitman and

imported into the application through a module called pyDES under the MIT License. In order to send encrypted messages, we used base-64 encoding so that the binary ciphertext could be encoded in a format that could be sent as a string through the socket connection.

Key Exchange: Once users were successfully being authenticated through the remote database and messages were being encrypted with arbitrarily generated keys, our focus was on the final part of the application: key exchange. This part of the cryptographic scheme truly only applies to the client-server key exchange because, once messages between the client and server are being encrypted using this key, a key exchange between clients is just as secure as having the server establish them. The **`diffiehellman.py`** program is largely dedicated to this segment of functionality, featuring a Python class that is capable of generating a public/private key pair, calculating a shared secret key, and verifying that a hashed secret key (along with an optionally appended nonce prior to hashing) matches that which is stored in the object. Once this class was developed and tested, we were then faced with the challenge of incorporating this handshake into the connection between the client and server. The client was first modified to randomly select the parameters for a new Diffie-Hellman object, initialize this new object (creating a public/private key pair) using the class constructor, and then send the chosen parameters along with the client's public key in an initialization message to the server. The server could now be modified to receive this message and, using the parameters to generate its own public/private key pair, could now map the Diffie-Hellman object to the client that sent the message. This object could then be used in the future for encrypting and decrypting messages at the server level. Using the client's public key, the server can now calculate the shared secret key which would be associated with that client. Then, through a series of messages that follow the cryptographic scheme we developed, both the server and the client are able to verify each other's identity and the key that was negotiated between them.

Execution

Our code was written in Python 3.6 and has been tested on both Windows and Linux machines. Before running the program, make sure that this version of Python and the following modules are installed on the system: **`pymysql`**, **`bcrypt`**, and **`pyDes`**. To install these modules, simply execute

```
pip install <module name>
```

To run the program, start the server before running any clients. If the client is started before the server is initialized, users will be able to enter their credentials and authenticate themselves, but will not be able to connect to the server in order to continue. Execute

```
python server.py
```

to start the server; once running, you should see the following output:

```
[SERVER]: Initializing...
[SERVER]: Initialization complete.
[SERVER]: Ready to accept connections.
```

Next, execute

```
python client.py
```

in another terminal or command prompt to establish a client. This may be done an arbitrary number of times in an arbitrary number of windows for an arbitrary number of clients. Once the client program has been started, you should see a prompt for a username:

```
[CLIENT]: Username:
```

Once a username is entered, you will next see a prompt for a password:

```
[CLIENT]: Password:
```

After logging in, you will be notified of successful authentication before the client then attempts a connection to the server.

```
[CLIENT]: User logged in successfully.
```

```
[CLIENT]: Connecting to server...
[CLIENT]: Connection successful.
```

Once connected to the server, you will be prompted for the username of the client that you would like to communicate with.

```
[CLIENT]: Destination:
```

After providing this username, you will be notified of the destination user's status (either online or offline). If offline, you will be notified and should wait for them to log in.

```
[CLIENT]: Destination user is offline.
```

If online, you will be notified and, from then on, able to send and receive messages.

```
[CLIENT]: Destination user is online.
-
```

Vulnerabilities

As proud as we are of our messenger and the things we have been able to accomplish, it is unfortunately still vulnerable. The weak point in the current scheme is the server: if it is compromised, an attacker could see the key that was just distributed to a pair of clients and be able to decrypt their messages with ease until a new key is generated. Short of this, and not considering any vulnerabilities that arise from factors outside of the system itself (users, workstations, etc.), there is very little that an attacker could do through eavesdropping to compromise messages. Perhaps the only reliable way in which to attack the server is by overloading the server with connections in an attempt to wage a denial-of-service attack. Even so, this attack would not compromise confidentiality or integrity, but only the accessibility of the service.

Conclusion

All things considered, our group accomplished more than what we set out to do with respect to both technical requirements and personal expectations. What started out as a simple, cryptographically-insecure application quickly evolved into fully authenticated clients communicating with a server acting as a key distribution center. In total, we collectively wrote over 1100 lines of code involving threads, sockets, encryption, key generation, hashing, database queries, and key exchanges. By carefully considering both the user and attacker perspective, our team was able to create an application that was both user friendly and attack resistant.