# Proposal for a secure group chat application

Eli Meckler and Sophie Mori

We propose an extension of a basic chat application to provide users a more secure chat experience. This application uses the concept of a "chat room," where the group of users in the conversation is established in the beginning and unaltered throughout the conversation. Users send messages to a server, which then delivers the messages to the other users in the group. Users may also leave the chat and return at a later time and receive all the messages sent in the meantime.

Messages sent by one user should be delivered to the intended recipients in the group. The message should be unmodified and sent at the intended time. A user or outside observer who is not part of the conversation should not be able to read or tamper with the message. No user should be able to impersonate another user.

To guarantee this security, we need to provide the following services:

- Message confidentiality (all messages should be encrypted and not readable by eavesdroppers)
- Message integrity and authentication (the message should come from the supposed sender and should not be tampered with)
- Replay protection (a delayed or repeated message is rejected as a valid message)

In the following sections, we describe how we plan to establish a shared key amongst all users in a chat group, and how we plan to implement each service described above.

## Key establishment

The goals of key establishment are five-fold. First off, the established key $K_S$ must remain confidential. In order to establish a shared symmetric key $K_S$, every user in the channel must acquire this key securely. Secondly, $K_S$'s integrity must assured. If a transmission error or malicious tampering affects any sent message during the key establishment, the users should be aware of this. Third, both users must be sure that the other user is who they claim (non-repudiation). Otherwise, A may be sharing $K_S$ with a malicious third party unknowingly. Fourth, key establishment should have perfect forward secrecy: should either user's computer be compromised after the session ends, $K_S$ should not be obtainable. Finally, both users must be sure $K_S$ is communicated on a channel secured with a fresh key; otherwise, the messages may be read due to a previous security compromise. We propose the Diffie-Hellman (DH) key exchange between the creator of the chat and all joined users as a method of sharing $K_S$.

*Signed Diffie-Hellman Encryption*

Let A be the user establishing a chat room. A will create $K_S$ using a cryptographic random number generator. When a user B joins the chat, A performs the signed Diffie-Hellman protocol to develop a private channel with key $K_{AB}$. A first generates a large prime $p$ and a private number $x$. She then sends B a message with A's name (as the sender), B's name (as the recipient), the large prime $p$, and $g^x \bmod p$. B then generates his own private number $y$ and responds with his name, $g^y \bmod p$, and a digital signature of both modulo values and both users' names. A, to confirm to B that she has received all information, replies with her name followed by a signature of both modulo values. Both A and B can then calculate $K_{AB}$ as $(g^y)^x \bmod p$ and $(g^x)^y \bmod p$ respectively. A can then send $K_S$ to B in a signed message. Finally, both A and B delete all information needed to recover $K_{AB}$, and the key exchange protocol ends.

Key exchange messages will have a header that third parties in the chat room will recognize and ignore, so the users are not shown these exchanges

This method meets all of the requirements of key exchange. Firstly, $K_S$ will only be readable by users who know $K_{AB}$. Keys produced by the DH protocol is computationally intractable to calculate without knowing x or y. Both integrity and non-repudiation are guaranteed by signing the messages. $K_S$ has perfect forward secrecy with respect to the key exchange if both A and B delete $x$, $y$, and $K_{AB}$ upon completing the key exchange. Finally, A and B can both verify that $K_{AB}$ is fresh since they know their contribution ($x$ and $y$, respectively) is fresh.

**Message confidentiality**

The goal of message confidentiality is to prevent someone who is not in the group chat from acquiring the meaningful plaintext messages sent within the group. Therefore, all messages must be encrypted in a manner that is safe from attacks. Some methods of encryption are easier to break than others. We propose using AES-256 cipher block chaining (CBC) symmetric key encryption in conjunction with the shared key $K_S$ obtained from the key establishment phase as described above.

*AES-256 CBC Encryption*

To encrypt a message for the group, a user first adds padding to the message to make the message's length a multiple of the block size and calculates an unpredictable random initial value IV. Then, the message is run through the CBC block cipher taking IV as the initial value and $K_S$ as the key: each block of input is XORed with the previous block's encrypted output before being encrypted itself (the first block being XORed with IV). Finally, the user sends the encrypted message and the IV to the server which then distributes it to the users in the chat room. To decrypt, the receivers decrypt each block with $K_S$ and XOR the result with the previous block's encrypted form (the first block being XORed with IV). In our implementation, we will use TLS padding with the padding length byte repeated throughout the padding.

CBC encryption schemes with a small block permutation space are susceptible to collision attacks. If an attacker observes identical encrypted blocks, the attacker may find the XORed value between two plaintext messages. Knowing part of all of one message would then reveal information about the other message. We drastically lower the possibility of generating identical blocks by using a cipher with a large block length (AES-256).

CBC may be vulnerable to a padding oracle attack, in which an attacker can send variations of encrypted messages and decode a given message by interpreting the resulting padding error messages. However, in our implementation, receiving users will simply throw out messages that contain errors and will not return an error message, so an attacker cannot use any of the users or the server as an oracle.

Our proposal relies on unpredictable initial values for CBC encryption; if the initial values are predictable, a chosen-plaintext attack can be used to recover information from earlier encrypted messages by guessing which IV will be used to encrypt the attacker's own chosen messages. This is avoided by simply assigning a cryptographic pseudo-random variable to be the IV of each message. Cryptographic pseudo-random variables, are, by definition, considered unpredictable in security contexts.

## Message integrity and authentication

The goal of message authentication is to confirm that a received message comes from the stated sender, has not been changed by another party, and cannot be repudiated by the sender. Here, we can use the idea of a digital signature, which provides all these services. The signing user uses a key-pair generation function to generate both a public key $K^+$ and a private key $K^-$. Then the user uses a signature generation function using his private key to generate a signature s for the message. The receiver can then verify the signature with a third function that uses the signing user's public key, the message, and the signature. We will use RSA encryption, or Public Key Cryptography Standards (PKCS) #1.

*RSA Encryption*

RSA encryption works via public and private key production using large primes. A user chooses two large primes $p$ and $q$ and calculates their product $n$. The user then calculates the totient $\varphi(n)$, or the number of integers less than $n$ that are also coprime to $n$. The user then chooses an integer $e$ such that $1 < e < \varphi(n)$ and such that $e$ is coprime to $\varphi(n)$. Finally, the user computes $d$ such that $de \equiv 1 \bmod \varphi(n)$. All these operations are easy operations because the user already knows $p$ and $q$. The pair $(e, n)$ is released as the public key, and from now on, others wishing to communicate with the user may encrypt their message $m$ as $c = m^e \bmod n$. The user still holds the private key $d$, so the user can easily decrypt the message by using the operation $m = c^d \bmod n$. In order to break RSA encryption, the attacker would have to compute $d$ from the pair $(e, n)$, which is a problem equivalent to factoring $n$. This is believed to be a hard problem not solvable in polynomial time.

*PKCS #1*

RSA does encounter some issues when used on its own. The encryption is deterministic, meaning an attacker can match new messages to known ciphertext and quickly uncover the plaintext messages. Additionally, short messages have very easily-calculable decryptions, which, if the encrypting exponent is known (which we must assume), is computationally feasible to break. Finally, RSA encryption is homomorphic, where the encryption of the product of two messages is the same as the product of their separate encryptions, in mod n. This unfortunately leads to the possibility of chosen ciphertext oracle attacks.

All of these issues can be fixed by extending the encryption scheme with the Public Key Cryptographic Standard (PKCS) #1 formatting which adds nondeterministic formatting and message padding to RSA encryption.

### Replay protection

Replay protection ensures an attacker cannot simply parrot a message sent by another user and have it be mistaken for a new message. The goal is to ensure each message has a temporal marker that specifies the time, be it relative to other messages or absolute. We propose using relative message sequence counters to ensure replay protection.

*Message Sequence Counters*

Every user A maintains a counter $C_{sndA}$. This tracks the number of messages sent to the group by A. For group member B, A maintains $C_{rcvB}$, the number of messages sent by B. A would maintain such a counter for each other group member as well. Every time A sends a message to the group, A increments its sending sequence counter to $C_{sndA} = C_{sndA} + 1$ includes it in the message and digital signature. Upon receiving a message from A, B will check that $C_{sndA} = C_{rcvA} + 1$. If so, the message is fresh, and B will accept it and update its receiving sequence counter to $C_{rcvA} = C_{sndA}$. Otherwise, the message is old, and B would not accept the message. If a user leaves a group temporarily and rejoins, the user's system will read polled messages from the server one at a time, complying with the above protocol. By the end of this process, the user will be up to date with the rest of the chat room.

Using message sequence counters as opposed to timestamps creates a simple solution to users being able to confirm that old messages were fresh when they were sent, regardless of if the receiving user was online. Upon polling the server, the user will receive the messages in order and can easily verify the sequence counters. Note that this methods assumes the server is reliable and the connection is reliable—any dropped packets will cause all future packets to be rejected.

## Conclusion

We will implement the secure multi-party chat application as explained above by extending the provided plaintext chat application. Almost all changes will be client-side, as we are ultimately not trusting the server with any information or protocol handling. The PyCrypto package provides many of the cryptographic primitives, and we will make use of those in our implementation.