CryptoChat: A Peer to Peer Encrypted Multiparty Live Chat Service

James Little (jameslittle230@gmail.com) • April 2018 • AIT-Budapest

Introduction

CryptoChat is a fully peer-to-peer browser-based encrypted chat service that uses strong encryption primitives and well-defined protocols to exchange messages in real time between multiple users. The service uses a combined WebSockets and HTTP server to send and store encrypted messages from sender to receiver. Messages are encrypted using AES, protected using a MAC, then further verified using the RSA cryptosystem. This combination ensures quick encryption and decryption, message integrity protection, and verification of both recipient and sender.

The client program, written in Javascript, runs as a web page in a browser. Users register on the web page with a username and password combination. This password is used to generate RSA keypairs, which become the foundation for the cryptographic protocol. Messages are routed through a chat server which communicates with the client using HTTP (for database access) and a WebSockets connection (for live messaging). The chat server never stores unencrypted message text or the private keys needed to decrypt a message; those keys are only stored on the user's computer using in-browser storage APIs.

System Architecture

The application is designed around two codebases: a persistent backend server running on a publicly accessible machine, and a web application downloaded to the browser and run on the user's machine. The server manages WebSockets connections between the browser and client, sending incoming messages to the appropriate client; and manages database connections, storing RSA keypairs, encrypted message payloads, and chat metadata. The client manages encryption and decryption and constructs the user interface.

The design and usage of the application is heavily based on Facebook's live chat service found at http://messenger.com. Users log in or register with a username and password pair. After logging in, users see a list of chats they have been in, along with the option to create a new chat. These chats associate users with a message stream: messages are sequentially added to a chat: a group of users permitted to send and receive these messages. Then, after selecting a chat, users can type messages into an input box; messages will be encrypted on the client and sent to each designated recipient.

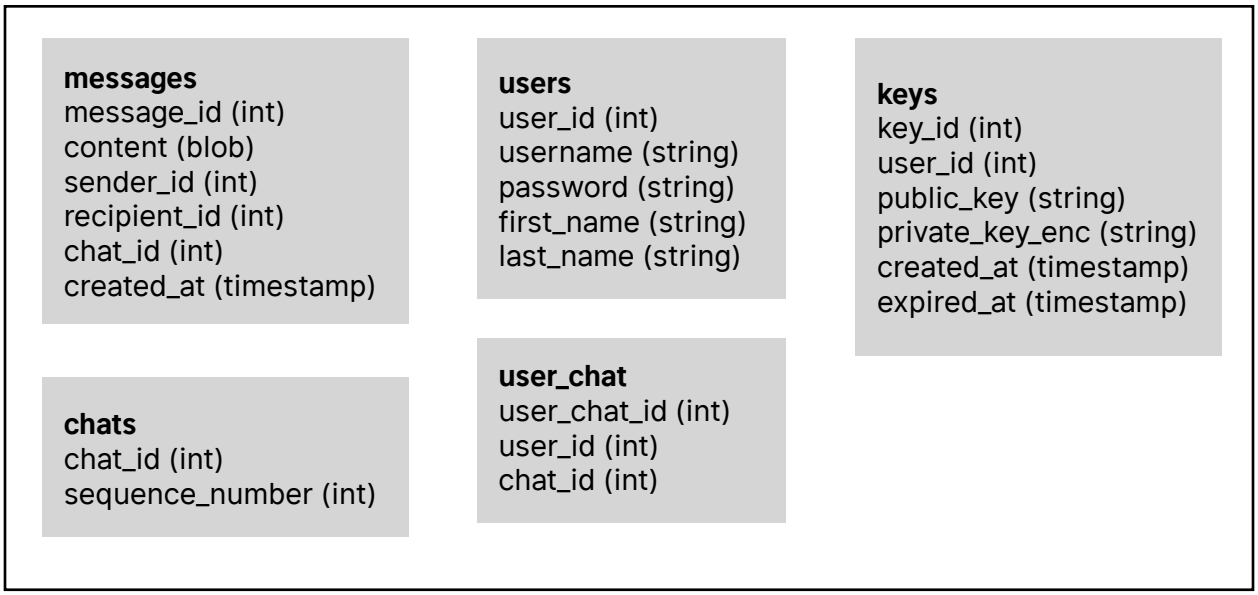


FIGURE 1: THE DATABASE SCHEMA USED ON THE SERVER

The system is designed such that the users only have to enter their username and password once per session. From there, the client creates and submits a new, random RSA keypair and expires the RSA key pair from the previous session. (This means that any messages sent to a user while that user is offline is encrypted using the RSA key pair from that user's last session.) Keys aren't rotated on a time basis, but if the user logs in enough, it will provide sufficient key rotation. Keypairs are stored in a database on the server, allowing users to decrypt messages for which they are the recipient at any point in time. However, messages received using old keys will be detected as invalid, since the application will cross reference the message's send time with the issuance and expiration dates of the keypair.

The database stores encrypted messages with their metadata. Each message is associated with a chat, which is a grouping of users that have a channel to communicate together. Users themselves have metadata associated with them. Finally, the database stores RSA keys, their metadata, and their user association. The full database schema is shown in Figure 1.

Internet Protocol Usage

WebSockets will only be used to route live information from sender to recipient, such as incoming messages and chat creation. All other communication between client and server, including past message access, key sharing, will take place using an HTTP API which will run concurrently, sharing database access. In other words, when the server needs to send information to the client, the WebSockets protocol will be used; when the client needs to request information of the server, HTTP will be used.

Security Model

The application treats all WebSockets communications as interceptable and modifiable, and all HTTP responses as visible (but not modifiable) to anyone except for the initial login request and response. Furthermore, it treats the web browser's LocalStorage API as secure. In a published application, all communications between server and client would be secured through HTTPS, and encryption keys would not be able to be stored locally since there is no realistic way a web browser can have a secure storage mechanism. However, with the previous two assumptions, we can nicely define an attacker model and define security requirements from that.

Attacker Capabilities

The server, having no knowledge of where a message comes from, must treat all incoming messages as potentially malicious, either modified in transit, replayed, or forged. Attackers might also have read (but not write) access to any data in the database. With these two considerations, the application must be able to ensure that no message content can be decrypted without ensuring that the message is correct sequentially, is meant for the recipient, has been signed by the sender, and has not been modified in transit.

The server itself is not within the scope of attack. If an attacker gains access to the chat server to change source code, this will cause issues out of scope of the cryptographic model used to secure chat messages. However, the clients do not assume that any information coming from the server can be verified, since an attacker could intercept and modify any message coming from the chat server to the recipient.

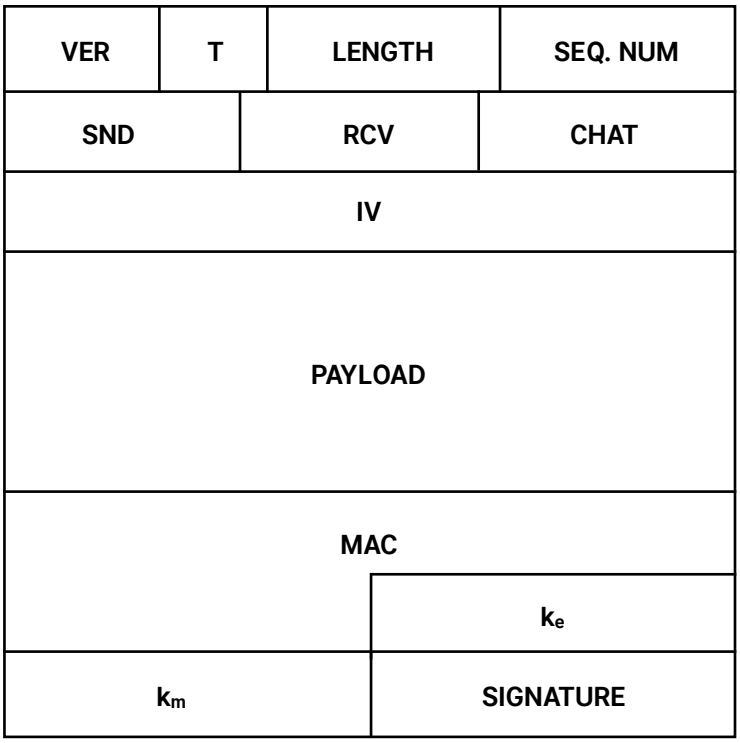
Secure Channel Protocol

Message Envelope

Message payloads are enveloped in various encryption primitives and sent over the socket connection as a binary blob. Figure 2 shows the message envelope protocol: the message is preceded by a version, a message type, a sender ID, a recipient ID, a chat ID, a payload length and a sequence number. This sequence represents the header of the message. Next, a random Initial Value (IV) for AES-CBC encryption is added. The next segment (of variable length) is the encrypted payload, encrypted using AES-CBC encryption using the CryptoJS Library (https://github.com/brix/crypto-js). Both the 16 byte long IV and the 32 byte long AES key are generated randomly by the client. After the encrypted payload comes the MAC of the headers, IV, and encrypted payload generated by CryptoJS using the SHA-256 based HMAC algorithm. Like the

AES key, the 16 byte long HMAC key is generated randomly.

Following the MAC comes the encryption key and the MAC key, both encrypted themselves using the RSA cryptosystem implemented by the JSEncrypt library (http://travistidwell.com/jsencrypt/) using the recipient's public key. Finally, the envelope contains a signature: the hash of the headers, payload, MAC and encrypted keys all encrypted using the sender's RSA private key.



Ver: Major/Minor protocol version (2 bytes)
T: Message type (1 byte)
Length: Payload Length (4 bytes)
Seq. Num: Sequence Number (4 bytes)
Snd: Sender ID number (4 bytes)
Rcv: Receiver ID number (4 bytes)
Chat: Chat ID number (4 bytes)
Header
IV: Initial Value for payload encryption (CBC Mode)
MAC: HMAC value for header, IV and payload
Ke: RSA-encrypted payload symmetric key
Km: RSA-encrypted HMAC key
Signature: RSA signature of hash of headers, payload, MAC, keys

FIGURE 2: THE MESSAGE ENVELOPE

Message Handling

When the user types a message in and presses the "Send" button, the envelope is generated on the client and is sent as a hexadecimal string using socket.io. When the message is received, the header is parsed and the sender, recipient and chat is determined. Using this information, the message is recorded to the database. If the user is determined to have an open WebSockets connection, the message is also sent to that connection. All encryption happens on the sender's client and all decryption happens on the receiver's client, the server acts as a go-between to save messages to the database and associate recipients with their active WebSockets connection.

Any message being sent to a chat that contains more than two people will be sent multiple times, encrypted with each recipient's private key. Each repeating message will be stored in the database, but each repeating message will have the same sequence number.

Key Establishment Protocol

RSA keypairs are used as the outermost layer of encryption, so RSA keypair management is heavily considered during the application lifecycle. RSA keypairs are generated once per user, per session (a session here is a login/logout cycle; every closed socket session is considered to be a logout). On each login, the client generates a new RSA keypair and expires the previous keypair. The keypair is stored in the database: the public key in cleartext and the private key encrypted with the user's password. The user's password, stored throughout the session, is used on the client side to decrypt the private key and encrypt messages. HTTP endpoints can be used to retrieve RSA keys, since any database information is assumed to be accessible by any logged-in user.

Message sequence numbers exist on a per-chat basis: in other words, each chat has an associated sequence number that starts at 1 and monotonically increases with every chat. Sending and receiving sequence numbers are stored locally on each client. When the user successfully receives a chat (and the message can be accepted), they notify the database to update the sequence number. The sequence number is stored so that correct numbering can continue between sessions. Sequence numbering is explicit: the sequence number is stored in plaintext in the message envelope and is part of the MAC data. Clients will use a replay detection window to ensure message validity.