Let's Chat

Abstract

Let's Chat is a fully-encrypted messaging system that enables users to safely and securely communicate. The server accepts and forwards messages to their intended recipients, the clients can interact securely with the server, and the users should feel confident that they are speaking with the persons they believe they are speaking with. In this report, I intend to examine security goals for the system, discuss the technical decisions I have made to address the aforementioned, present the challenges encountered to handle the mechanics of securely developing the application, and explore the future of the application.

Security Goals

The notion of Let's Chat is that it provides end-to-end encryption which enables a user to securely send messages to their intended recipient while ensuring complete privacy.

Let's Chat aims to accomplish the following goals:

- **Confidentiality**: users can send securely send messages without the fear of messages being disclosed
- **Integrity**: messages delivered by users remains unaltered and unmodified
- **Non-repudiation**: users can not deny having sent or received messages
- **Authenticity**: users are confident knowing that they are communicating with the intended party

Let's say that we have two parties, Alice and Bob. Alice wants to send Bob a message and Bob wants to send Alice a message. Assuming that this channel is not encrypted, this unencrypted channel is susceptible to man-in-the-middle attacks where malicious Malory can sit in between the channel and read and manipulate the incoming messages along with pretending to be Alice or Bob.

Alice needs to feel confident that her messages sent and intended solely for Bob can only be read by Bob. On the other hand, Bob needs to feel confident that his messages sent and intended for Alice alone can only be read by Alice. On both ends, the sender and the recipient entrust the system to provide end-to-end encryption (secure communication) and authentication (each user is who they say they are). Thus, ensuring complete privacy.

Let's Chat: Technical Infrastructure

Let's Chat is a web application built using HTML/CSS, vanilla Javascript, Node.js, a cross-platform server environment, and the Express framework, a layer built on top of Node.js that aids in managing servers and routing.

To begin with, Let's Chat utilises HTTPS to provide privacy by encryption during message transit. HTTPS uses the Transport Layer Security (TLS) network protocol that creates a secure link with an authenticated individual over the network. TLS uses an asymmetric public key infrastructure to secure communication by enforcing server authentication, confidentiality, and integrity. The private key is kept hidden by the application and any information encrypted by the public key is decrypted using the private key stored on the web server. Those who intend to communicate securely with the server can do so by using the public key. Those who wish to decrypt the data must use the private key.

Upon starting the application, the site will transmit an SSL certificate containing the public key required to initiate a secure session (using the SSL/TLS handshake between the client and the server) upon user connection.

To do this, I generated a self-signed SSL certificate locally which produced the following two files: the self-signed certificate file (server.cert) and the certificate's private key (server.key).

Though using a self-signed certificate is sufficient for this practicality, ideally, I would utilise a Certificate Authority signed certificate in a production setting to ensure that users can trust the identity of the messaging application.

In contrast to deploying my application on an HTTPS server, an HTTP server enables plaintext transmission over the internet which enables anyone observing the connection to view the transmitted information. If a user wishes to send sensitive information such as banking details and their social security number through Let's Chat, these sensitive pieces of information could be especially susceptible to security breaches. Thus, the reason why we would want to employ an HTTPS server.

Once the server has established a secure connection, the user is prompted to enter a username and select a room. Upon clicking "Join Chat," the user is redirected to the chat room.

To communicate with one another, Let's Chat uses the Socket.IO library which permits bidirectional, event-based, and low-latency communication between a client and a server.[1]

Along with Socket.IO, this application utilises Node.js' Crypto library along with CryptoJS, a collection of safe and reliable cryptographic algorithms implemented in JavaScript to provide confidentiality, integrity, authentication, and non-repudiation.

Confidentiality

To create a shared secret (symmetric key) for Alice and Bob, I created Elliptic Curve Diffie-Hellman (ECDH) key exchange objects for each user using the secp256k1 curve. Let's say that "a" is Alice, "b" is Bob, and "G" is the ECC (Elliptic-curve cryptography) generator point.

[1] https://socket.io/docs/v4/

We can exchange the values **(a * G)** and **(b * G)**, which are Alice's and Bob's public keys across the Node.js server to derive a shared secret, **(a * G) * (b * G)**, using their private keys accessible through locally stored Privacy Enhanced Mail (PEM) files.

ECDH follows a simple algorithm:[2]

- Alice creates an ECC key pair at random:
  - **{alicePrivateKey, alicePubKey = alicePrivKey * G}**
- Bob creates an ECC key pair at random:
  - {**bobPrivKey, bobPubKey = bobPrivKey * G}.**
- Public key exchanges between Alice and Bob happen across an insecure channel (Let's chat generates a "sessionKey.pem" file for users to access)
- The formula used by Alice is **sharedKey = bobPubKey * alicePrivKey**.
- The formula used by Bob is **sharedKey = bobPrivKey * alicePubKey.**
- Now, Alice and Bob have the same sharedKey, which is equal to **bobPubKey * alicePrivKey** and vice versa.

Given that Alice and Bob's private keys are never revealed, Bob's or Alice's private keys cannot be obtained through Man-In-The-Middle attacks. To encrypt and decrypt messages, I opted to use Advanced Encryption Standard (AES), a United States Federal Information Processing Standard (FIPS), because it was deemed an industry-standard after a five-year evaluation process of 15 rival designs.[3] By using AES in Cipher Block Chaining (CBC) mode with Public-Key Cryptography Standards seven (PKCS7) padding, users can encrypt and decrypt a series of bits as a single block by applying a cipher key to the entire block. Once two users join, a secret session key is generated locally using their ECC key pairs. Then, upon sending the first message, the user is prompted with entering the shared key to encrypt. To decrypt messages, the first message received requires the user to reenter the shared key. After the initial encryption and decryption, the system does not prompt users to enter the session key. If the wrong shared key is entered, this will result in an empty text message (though the username and timestamp are visible).

The usage of ephemeral session keys enables users to feel confident that a single shared key can not compromise the security of messages sent in previous sessions.

I decided to use 128-bit AES as opposed to 256-bit AES because 128-bit AES offers greater performance during the encryption and decryption process despite not being as secure.[4] Similarly, I have opted for AES as opposed to Rivest–Shamir–Adleman (RSA) because RSA requires significantly higher computational prowess than AES. Since users may want to send large messages, I did not want to compromise performance and speed.

Though brute-force computation can not crack AES, the algorithm is subject to the following attacks if not implemented correctly: related-key, side-channel, and known-key distinguishing.

[2] https://cryptobook.nakov.com/asymmetric-key-ciphers/ecdh-key-exchange
[3] https://cryptojs.gitbook.io/docs/
[4] https://www.cloudwards.net/what-is-aes/

Related-Key Attacks

The first method of exploiting a poorly implemented AES algorithm is known as a related-key attack. This type of attack is possible if the attacker can connect the known public key to the corresponding secret private key, allowing them to decrypt the data.[5]

Side-Channel Attacks

Computer signals, in theory, can be detected by monitoring physical electromagnetic leaks. If the private key is leaked, the attacker can decrypt the data as if they were the intended receiver.[5]

Known-Key Distinguishing Attacks

A known-key attack requires the attacker to know the key used to encrypt the data. With the key in hand, the attacker can decipher the encrypted data by comparing it to other encrypted data whose contents are known. This type of attack, however, is only effective against seven rounds of AES encryption, implying that even the shortest key length (128-bit) would be immune because it uses ten rounds. Hence, the chances of an attacker knowing the original key are extremely low.[5]

Integrity and Authentication

Let's Chat uses Hash-Based Message Authentication Codes (HMAC),[6] a message authentication mechanism that employs cryptographic hash functions, in combination with the SHA256 cryptographic hash function necessary to enforce integrity and authenticity. Given that Alice and Bob both now share a secret key, the messages sent to each other need to remain unaltered and verifiable. Each message sent is a hashed cipher created using the HMAC-SHA256 algorithm which takes in an encrypted message and the shared key as its parameters. Then, the receiver can check the message's integrity by re-computing the encrypted message's hash using the shared key and comparing it to the received HMAC. Since HMAC utilises symmetric key cryptography instead of using private and public pairs for the sake of simplicity, HMAC there arises security concerns regarding key distribution and the shared secret being compromised and jeopardising non-repudiation.

Non-repudiation

Non-repudiation guarantees that the sender of the message receives proof of delivery and that the recipient receives verification of the sender's identity. This ensures that neither party can subsequently dispute receiving the message. Though Let's Chat enables communication solely between two users where a straightforward HMAC validation may suffice, I developed the application with the intent to enforce non-repudiation without necessarily sharing a secret key amongst a substantial number of users.

To solve this security challenge and achieve non-repudiation while enforcing integrity and authentication, the future of Let's Chat will employ digital signatures using ECC. Let's Chat will generate private keys locally to a user's computer and store each user's public keys on the server.

[5] https://www.cloudwards.net/what-is-aes/
[6] https://cryptojs.gitbook.io/docs/

Alice first encrypts the message with Bob's public key sent to her by the server. Then, Alice creates a digital signature using the encrypted message computed using SHA-256 and a private key that gets attached to the scrambled message. The digital signature along with the encrypted message is sent to Bob. Then, Bob proceeds to verify that the message is authentic and sent from Alice by using Alice's public key. Lastly, if the signature appears unaltered, Bob decrypts the message with his private key. The compromise of the shared key alone or the long-term ECC keys alone would not jeopardise system security and enable attackers to decrypt old messages.

Though in this scenario Alice and Bob are manually encrypting and decrypting messages, Let's Chat will handle this behind the scenes so that upon initial conversation with another party, key exchange will be done once. Additionally, the application in the future will entail secure account creation and user login. If a user wishes to delete their account, all previously sent messages, along with the user's keys, will be deleted so that they can no longer read old conversations.

Further, Let's Chat could be improved by utilising the Signal Protocol (formerly known as the TextSecure Protocol) since Signal contains rare security features like forward secrecy and post-compromise security in addition to some conventional security properties like confidentiality, integrity, and authenticity.[7] Based on a shared secret key, two parties can exchange encrypted messages using the Double Ratchet method. The users will then agree on the shared secret key using a key agreement protocol (like what Let's Chat does using Elliptic Curve Diffie-Hellman) which enables them to then send and receive encrypted messages using the Double Ratchet. For every Double Ratchet transmission, the users generate fresh keys to prevent the calculation of earlier keys from later ones. Along with their messages, the users send Diffie-Hellman public values. To prevent later keys from being calculated from earlier ones, the results of Diffie-Hellman calculations are mixed into the derived keys. If a party's security is compromised, these characteristics provide some protection to earlier or later encrypted messages.[8]

[7]https://valsamaras.medium.com/the-signal-protocol-and-the-double-ratchet-algorithm-e3d01d1e403f
[8] https://signal.org/docs/specifications/doubleratchet/

References

(n.d.). CryptoJS - CryptoJS. Retrieved November 12, 2022, from https://cryptojs.gitbook.io/docs/

ECDH Key Exchange. (n.d.). Practical Cryptography for Developers. Retrieved November 11, 2022, from https://cryptobook.nakov.com/asymmetric-key-ciphers/ecdh-key-exchange

Hougen, A. (2021, May 26). *What Is AES Encryption & How Does It Work in 2022? [256-bit vs 128-bit]*. Cloudwards. Retrieved November 12, 2022, from https://www.cloudwards.net/what-is-aes/

Introduction. (2022, October 15). Socket.IO. Retrieved November 10, 2022, from https://socket.io/docs/v4/

The Signal Protocol and the Double Ratchet algorithm. (2021, April 2). Retrieved November 14, 2022, from https://valsamaras.medium.com/the-signal-protocol-and-the-double-ratchet-algorithm-e3d01d1e403f

Specifications >> The Double Ratchet Algorithm. (2016, November 20). Signal. Retrieved November 14, 2022, from https://signal.org/docs/specifications/doubleratchet/