**ChatX RSA Documentation**

**Overview**

ChatX RSA is a secure, end-to-end encrypted chat application that uses RSA-2048 encryption to ensure the confidentiality of messages exchanged between users. This documentation provides an in-depth understanding of the project's structure, functionality, and implementation details.

**Files:**

1. cryptoworker.js

Purpose: Handles encryption and decryption tasks within a web worker using the jsencrypt library.

Functions:

- `generateKeypair()`: Generates and stores a key pair.

- `encrypt(content, publicKey)`: Encrypts the provided string with the destination public key.

- `decrypt(content)`: Decrypts the provided string with the local private key.

2. index.html

Purpose: Defines the layout and user interface of the chat application using Vue.js.

Components:

Chat Container: Displays messages and sender information.

Room Selection: Allows users to join chat rooms.

Notification Log: Displays system notifications and messages.

Key Display: Shows public keys of users.

Message Input: Allows users to type and send encrypted messages.

3. page.js

Purpose: Manages the core Vue.js instance controlling the UI and handles user interactions.

Functionalities:

- Key pair generation and management.

- Socket.io event listeners for real-time communication.

- Message encryption and decryption.

- Room joining and user interaction handling.

4. `app.js`

Purpose: Sets up an Express server for serving the web application and manages socket events for real-time communication.

Functionalities:

- Room management and user connections.

- Message broadcasting and encryption.

- Public key sharing and user disconnection handling.

5. `rsacryptoworker.js`

Purpose: Similar to `cryptoworker.js`, handles encryption and decryption tasks using the jsencrypt library.

Functions:

- `generateKeypair()`: Generates and stores a key pair.

- `encrypt(content, publicKey)`: Encrypts the provided string with the destination public key.

- `decrypt(content)`: Decrypts the provided string with the local private key.

**How It Works**

1. Key Pair Generation:

When a user accesses the ChatX RSA application, it automatically generates an RSA key pair locally in their browser using the generateKeypair() function. The key pair consists of a public key (which can be shared with others) and a private key (which is kept secret).

2. Room Joining:

The user can enter a room name in the chat application. When the user joins a room, the application uses Socket.io to establish a connection with the server (app.js). The server manages different chat rooms and user connections.

3. Public Key Exchange:

Upon joining a room, the user's public key is sent to the server, which then broadcasts this public key to all other users in the same room. This sharing happens through Socket.io events like 'PUBLIC\_KEY'.

4. Sending an Encrypted Message:

When the user sends a message, it is first encrypted locally in their browser using the recipient's (another user's) public key. The encryption happens in the web worker (cryptoworker.js). The encrypted message is then sent to the server via Socket.io (page.js). The server broadcasts this encrypted message to all users in the same room.

5. Receiving and Decrypting a Message:

When a user receives an encrypted message, the message is first decrypted locally in their browser using their own private key. This decryption occurs in the web worker (cryptoworker.js). After decryption, the original message is displayed in the chat window.

6. Real-Time Communication:

The entire process of sending and receiving messages, encryption, and decryption happens in real-time, allowing for secure and private communication between users in the same chat room.

7. Room Management and User Disconnection:

The server (app.js) manages the rooms and user connections. It handles events related to room joining and user disconnection. If a user leaves the room or disconnects, the server notifies the remaining users in the room, allowing them to manage their active connections.

In summary, ChatX RSA provides end-to-end encryption by generating RSA key pairs locally in the user's browser. Messages are encrypted locally using the recipient's public key before being sent to the server, ensuring that only the recipient (who possesses the corresponding private key) can decrypt and read the message. This secure communication is achieved through a combination of web workers for local encryption/decryption and Socket.io for real-time communication between users and the server.

Screenshots:

1. First you must go to the project directory and start the project with npm start.

A screen shot of a computer

Description automatically generated

1. Open chrome or any browser and enter the localhost and the port specified in the application.

A screenshot of a computer

Description automatically generated

1. Once your browser is opened you will see a screen similar to this.

A screenshot of a computer

Description automatically generated

1. To open another instance of the application you need to open an incognito window and put in the same address as the previous window. To join the where the first instance in running you need to copy the chatroom number and paste it in the input box of the new window.

A screenshot of a computer

Description automatically generated

1. Once the users have joined you can see their public key in the keys section.

A screenshot of a computer

Description automatically generated

1. You can see the user’s socket ID on the terminal however the messages cannot be seen as they are encrypted.

A screenshot of a computer

Description automatically generated

**Code:**

Index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Open Cryptochat</title>

<meta name="description" content="A minimalist, open-source, end-to-end RSA-2048 encrypted chat application.">

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">

<link href="https://fonts.googleapis.com/css?family=Montserrat:300,400" rel="stylesheet">

<link href="https://fonts.googleapis.com/css?family=Roboto+Mono" rel="stylesheet">

<link href="/styles.css" rel="stylesheet">

</head>

<body>

<div id="vue-instance">

<div class="chat-container full-width" ref="chatContainer">

<div class="title-header">

<h1>RSA secure chat</h1>

</div>

<div class="message-list">

<div class="message full-width" v-for="message in messages">

<p>

<span v-bind:class="(message.sender == originPublicKey) ? 'green' : 'red'">{{ getKeySnippet(message.sender) }}</span>

> {{ message.text }}

</p>

</div>

</div>

</div>

<div class="info-container full-width">

<h1>CHATROOM</h1>

<div class="room-select">

<input type="text" class="full-width" placeholder="Room Name" id="room-input" v-model="pendingRoom" @keyup.enter="joinRoom()">

<input class="yellow-button full-width" type="submit" v-on:click="joinRoom()" value="JOIN">

</div>

<div class="divider"></div>

<div class="notification-list" ref="notificationContainer">

<h1>NOTIFICATION LOG</h1>

<div class="notification full-width" v-for="notification in notifications">

<div class="notification-timestamp">{{ notification.timestamp }}</div>

<div class="notification-message">{{ notification.message }}</div>

</div>

</div>

<div class="flex-fill"></div>

<div class="divider"></div>

<div class="keys full-width">

<h1>KEYS</h1>

<h2>THEIR PUBLIC KEY</h2>

<div class="key red" v-if="destinationPublicKey">

<h3>TRUNCATED IDENTIFIER - {{ getKeySnippet(destinationPublicKey) }}</h3>

<p>{{ destinationPublicKey }}</p>

</div>

<h2 v-else>Waiting for second user to join room...</h2>

<div class="divider"></div>

<h2>YOUR PUBLIC KEY</h2>

<div class="key green" v-if="originPublicKey">

<h3>TRUNCATED IDENTIFIER - {{ getKeySnippet(originPublicKey) }}</h3>

<p>{{ originPublicKey }}</p>

</div>

<div class="keypair-loader full-width" v-else>

<div class="center-x loader"></div>

<h2 class="center-text">Generating Keypair...</h2>

</div>

</div>

</div>

<div class="bottom-bar full-width">

> <input class="message-input" type="text" placeholder="Message" v-model="draft" @keyup.enter="sendMessage()">

</div>

</div>

<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.4.1/vue.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.0.3/socket.io.slim.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/immutable/3.8.1/immutable.min.js"></script>

<script src="/page.js"></script>

</body>

</html>

Page.js

/\*\* The core Vue instance controlling the UI \*/

const vm = new Vue ({

el: '#vue-instance',

data () {

return {

cryptWorker: null,

socket: null,

originPublicKey: null,

destinationPublicKey: null,

messages: [],

notifications: [],

currentRoom: null,

pendingRoom: Math.floor(Math.random() \* 1000),

draft: ''

}

},

async created () {

this.addNotification('Welcome! Generating a new keypair now.')

// Initialize crypto webworker thread

this.cryptWorker = new Worker('crypto-worker.js')

// Generate keypair and join default room

this.originPublicKey = await this.getWebWorkerResponse('generate-keys')

this.addNotification(`Keypair Generated - ${this.getKeySnippet(this.originPublicKey)}`)

// Initialize socketio

this.socket = io()

this.setupSocketListeners()

},

methods: {

/\*\* Setup Socket.io event listeners \*/

setupSocketListeners () {

// Automatically join default room on connect

this.socket.on('connect', () => {

this.addNotification('Connected To Server.')

this.joinRoom()

})

// Notify user that they have lost the socket connection

this.socket.on('disconnect', () => this.addNotification('Lost Connection'))

// Decrypt and display message when received

this.socket.on('MESSAGE', async (message) => {

// Only decrypt messages that were encrypted with the user's public key

if (message.recipient === this.originPublicKey) {

// Decrypt the message text in the webworker thread

message.text = await this.getWebWorkerResponse('decrypt', message.text)

this.messages.push(message)

}

})

// When a user joins the current room, send them your public key

this.socket.on('NEW\_CONNECTION', () => {

this.addNotification('Another user joined the room.')

this.sendPublicKey()

})

// Broadcast public key when a new room is joined

this.socket.on('ROOM\_JOINED', (newRoom) => {

this.currentRoom = newRoom

this.addNotification(`Joined Room - ${this.currentRoom}`)

this.sendPublicKey()

})

// Save public key when received

this.socket.on('PUBLIC\_KEY', (key) => {

this.addNotification(`Public Key Received - ${this.getKeySnippet(key)}`)

this.destinationPublicKey = key

})

// Clear destination public key if other user leaves room

this.socket.on('user disconnected', () => {

this.notify(`User Disconnected - ${this.getKeySnippet(this.destinationKey)}`)

this.destinationPublicKey = null

})

// Notify user that the room they are attempting to join is full

this.socket.on('ROOM\_FULL', () => {

this.addNotification(`Cannot join ${this.pendingRoom}, room is full`)

// Join a random room as a fallback

this.pendingRoom = Math.floor(Math.random() \* 1000)

this.joinRoom()

})

// Notify room that someone attempted to join

this.socket.on('INTRUSION\_ATTEMPT', () => {

this.addNotification('A third user attempted to join the room.')

})

},

/\*\* Encrypt and emit the current draft message \*/

async sendMessage () {

// Don't send message if there is nothing to send

if (!this.draft || this.draft === '') { return }

// Use immutable.js to avoid unintended side-effects.

let message = Immutable.Map({

text: this.draft,

recipient: this.destinationPublicKey,

sender: this.originPublicKey

})

// Reset the UI input draft text

this.draft = ''

// Instantly add (unencrypted) message to local UI

this.addMessage(message.toObject())

if (this.destinationPublicKey) {

// Encrypt message with the public key of the other user

const encryptedText = await this.getWebWorkerResponse(

'encrypt', [ message.get('text'), this.destinationPublicKey ])

const encryptedMsg = message.set('text', encryptedText)

// Emit the encrypted message

this.socket.emit('MESSAGE', encryptedMsg.toObject())

}

},

/\*\* Join the specified chatroom \*/

joinRoom () {

if (this.pendingRoom !== this.currentRoom && this.originPublicKey) {

this.addNotification(`Connecting to Room - ${this.pendingRoom}`)

// Reset room state variables

this.messages = []

this.destinationPublicKey = null

// Emit room join request.

this.socket.emit('JOIN', this.pendingRoom)

}

},

/\*\* Add message to UI, and scroll the view to display the new message. \*/

addMessage (message) {

this.messages.push(message)

this.autoscroll(this.$refs.chatContainer)

},

/\*\* Append a notification message in the UI \*/

addNotification (message) {

const timestamp = new Date().toLocaleTimeString()

this.notifications.push({ message, timestamp })

this.autoscroll(this.$refs.notificationContainer)

},

/\*\* Post a message to the webworker, and return a promise that will resolve with the response. \*/

getWebWorkerResponse (messageType, messagePayload) {

return new Promise((resolve, reject) => {

// Generate a random message id to identify the corresponding event callback

const messageId = Math.floor(Math.random() \* 100000)

// Post the message to the webworker

this.cryptWorker.postMessage([messageType, messageId].concat(messagePayload))

// Create a handler for the webworker message event

const handler = function (e) {

// Only handle messages with the matching message id

if (e.data[0] === messageId) {

// Remove the event listener once the listener has been called.

e.currentTarget.removeEventListener(e.type, handler)

// Resolve the promise with the message payload.

resolve(e.data[1])

}

}

// Assign the handler to the webworker 'message' event.

this.cryptWorker.addEventListener('message', handler)

})

},

/\*\* Emit the public key to all users in the chatroom \*/

sendPublicKey () {

if (this.originPublicKey) {

this.socket.emit('PUBLIC\_KEY', this.originPublicKey)

}

},

/\*\* Get key snippet for display purposes \*/

getKeySnippet (key) {

return key.slice(400, 416)

},

/\*\* Autoscoll DOM element to bottom \*/

autoscroll (element) {

if (element) { element.scrollTop = element.scrollHeight }

}

}

})

App.js

const express = require('express')

// Setup Express server

const app = express()

const http = require('http').Server(app)

// Attach Socket.io to server

const io = require('socket.io')(http)

// Serve web app directory

app.use(express.static('public'))

/\*\* Manage behavior of each client socket connection \*/

io.on('connection', (socket) => {

console.log(`User Connected - Socket ID ${socket.id}`)

let currentRoom = null

/\*\* Process a room join request. \*/

socket.on('JOIN', (roomName) => {

// Get chatroom info

let room = io.sockets.adapter.rooms[roomName]

// Reject join request if room already has more than 1 connection

if (room && room.length > 1) {

// Notify user that their join request was rejected

io.to(socket.id).emit('ROOM\_FULL', null)

// Notify room that someone tried to join

socket.broadcast.to(roomName).emit('INTRUSION\_ATTEMPT', null)

} else {

// Leave current room

socket.leave(currentRoom)

// Notify room that user has left

socket.broadcast.to(currentRoom).emit('USER\_DISCONNECTED', null)

// Join new room

currentRoom = roomName

socket.join(currentRoom)

// Notify user of room join success

io.to(socket.id).emit('ROOM\_JOINED', currentRoom)

// Notify room that user has joined

socket.broadcast.to(currentRoom).emit('NEW\_CONNECTION', null)

}

})

/\*\* Broadcast a received message to the room \*/

socket.on('MESSAGE', (msg) => {

console.log(`New Message - ${msg.text}`)

socket.broadcast.to(currentRoom).emit('MESSAGE', msg)

})

/\*\* Broadcast a new publickey to the room \*/

socket.on('PUBLIC\_KEY', (key) => {

socket.broadcast.to(currentRoom).emit('PUBLIC\_KEY', key)

})

/\*\* Broadcast a disconnection notification to the room \*/

socket.on('disconnect', () => {

socket.broadcast.to(currentRoom).emit('USER\_DISCONNECTED', null)

})

})

// Start server

const port = process.env.PORT || 3000

http.listen(port, () => {

console.log(`Chat server listening on port ${port}.`)

})

**Conclusion:**

In a world where online privacy is paramount, ChatX RSA offers a secure haven for private conversations. With powerful RSA-2048 encryption woven into its core, this chat application ensures that messages remain confidential and exclusively accessible to intended recipients. Its intuitive design and real-time messaging capabilities create a user-friendly environment, making secure communication effortless. ChatX RSA stands as a beacon of digital privacy, empowering users to connect without compromise, assuring them that their online dialogues are shielded from prying eyes.