Tea Time

CIS 446 Term Project

By: Jodi Joven & Souad Omar 😀

Table of Contents

Introduction

For this project, we have made a secure end-to-end encrypted messaging app called TeaTime! The goal for our app is to create a functional, private, and encrypted communication system between our users, with security being at the forefront of all our features. This document will provide an overview of the system, core features, database schema, encryption methodology, potential security flaws, and mitigation strategies for these concerns. By analyzing an encrypted messaging algorithm in product-like conditions, we can understand the benefits and weaknesses of encryption technologies.

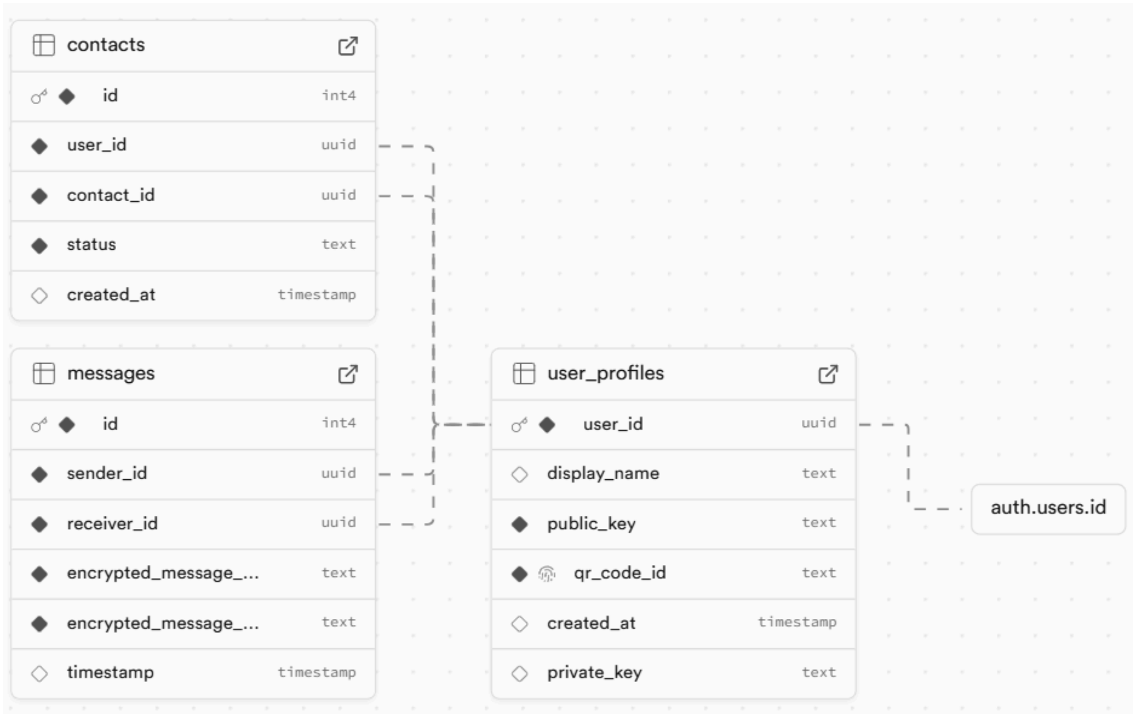
Overview and Core Features

TeaTime allows users to securely send and receive encrypted messages using RSA techniques. Users can add other authorized users as contacts if they have authorized their account using an additional verification method, and these contacts are able to freely and safely converse with each other while simultaneously knowing their data is protected. The frontend is built using React Native and TypeScript, utilizing the gluestack-ui component library in order to create an intuitive, clean frontend. Supabase is used for real-time data management and backend services, allowing for a streamlined verification process that authorizes a user's contact information while storing data securely.

For the purpose of this project, the React Native application is developed using Expo: a framework that supports native mobile development. We specifically are using Expo Go, which is a limited sandbox environment that allows us to develop the app and view our changes in real time on our personal mobile devices. Expo allows us to easily set up a foundation for our mobile app that supports smooth navigation and user interaction, allowing us to focus on the security of our encrypted messaging algorithm.

Database Schema

The database schema for TeaTime can be seen as follows:



There are four main tables in the database: auth.users.id, user_profiles, contacts, and messages. The auth.users.id table is a preconfigured table provided by Supabase that authorizes users who register for an account with TeaTime. A unique user ID is created upon registration and an automated email will be sent to the user's email address in order to verify the email address they provided.

Once a user successfully verifies their account, an entry will be made within the user_profiles table using the unique user ID created in the auth.users.id table. In addition to storing the user's provided display name and date of account creation, a unique QR code is generated for the user that other TeaTime users can scan through the app in order to add another user as a contact. Most importantly, a unique public key and private key are generated for the user which will be used in the message encryption process.

The contacts table contains all contact relationships between every user. An entry within this table will store the original user ID, the ID for the user's contact, the status of the connection (either Accepted or Denied), and the date the contact request was initiated.

Finally, the messages table stores all encrypted messages sent and received between users. The sender's ID, receiver's ID, encrypted message from the receiver, encrypted message from the sender, and timestamp from the message are securely stored within this table. Plain text from the user is not stored within the database and the encryption and decryption process is handled entirely by the React Native application.

The security provided by the Supabase database allows us to implement an RSA encryption algorithm for these secure messaging, which we will discuss in detail in the next section.

Encryption Implementation

Tea Time uses RSA to keep all messages secured and end-to-end encrypted. When a new account is created, a pair of RSA keys are generated, both a public and a private key. The public key is available and shared with other users in order to encrypt messages that are sent. However, the private key is kept secret and is used to decrypt messages. The keys are generated by using the node-forage library, allowing us to have 2048-bit RSA encryption with OAEP padding and SHA-256 hashing. Using this algorithm, which is standard in many popular messaging apps, allows us to keep users' messages safely encrypted.

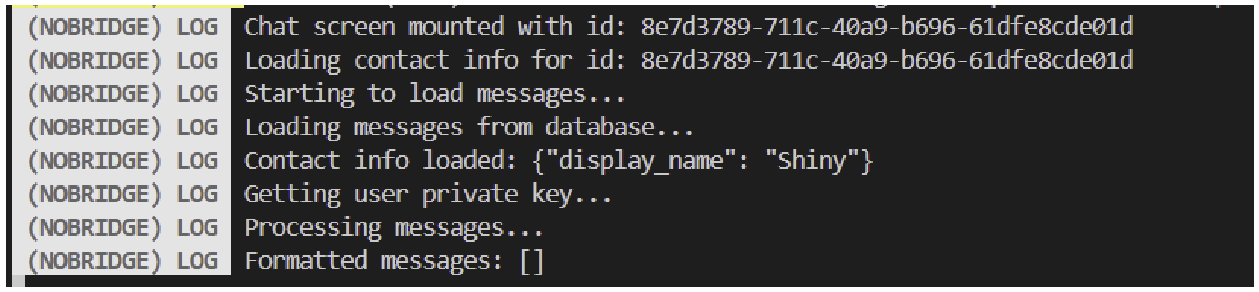
When a message is sent through our app, it is encrypted twice, once with the recipient's public key and once with the sender's public key. Both of these versions are stored in our messages database (via Supabase) under encrypted_message_for_receiver and encrypted_message_for_sender, respectively.

No plain text of the messages are ever stored in the database. Instead, its messages are only readable to users after they have been decrypted by their private key. This method ensures that only the sender and receiver can access the message contents, even if the database were exposed. It balances strong security with a smooth user experience by allowing both parties to see the full conversation without compromising end-to-end encryption.

While this method provides strong end-to-end encryption, it does run into some performance issues. Encrypting each message twice, once with the recipient's public key and once with the sender public key, needs additional processing time. This can be an issue especially as the conversation grows or messages are sent very quickly in rapid succession, which can overload the app. Despite the performance issues, we decided to prioritize the privacy of our users and their conversations.

Encryption Demonstration

When you open a fresh chat, this is what is shown: the system attempts to decrypt all messages in the database, but there are none at the moment.

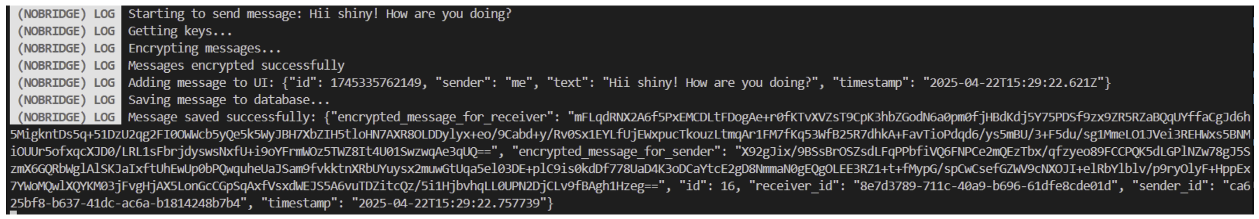


```
(NOBRIDGE) LOG  Chat screen mounted with id: 8e7d3789-711c-40a9-b696-61dfe8cde01d
(NOBRIDGE) LOG  Loading contact info for id: 8e7d3789-711c-40a9-b696-61dfe8cde01d
(NOBRIDGE) LOG  Starting to load messages...
(NOBRIDGE) LOG  Loading messages from database...
(NOBRIDGE) LOG  Contact info loaded: {"display_name": "Shiny"}
(NOBRIDGE) LOG  Getting user private key...
(NOBRIDGE) LOG  Processing messages...
(NOBRIDGE) LOG  Formatted messages: []
```

Here is how the empty conversation looks from the UI:

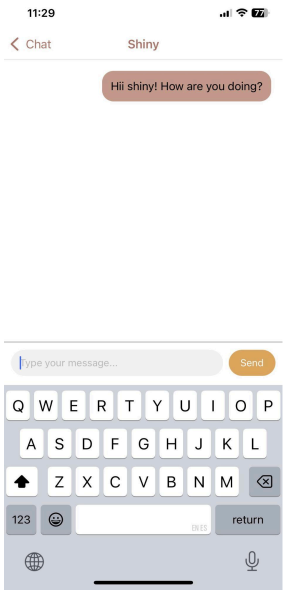
11:28
Chat
Shiny
Type your message...
Send

This is what happens when a message is sent:



```
(NOBRIDGE) LOG  Starting to send message: Hii shiny! How are you doing?
(NOBRIDGE) LOG  Getting keys...
(NOBRIDGE) LOG  Encrypting messages...
(NOBRIDGE) LOG  Messages encrypted successfully
(NOBRIDGE) LOG  Adding message to UI: {"id": 1745335762149, "sender": "me", "text": "Hii shiny! How are you doing?", "timestamp": "2025-04-22T15:29:22.621Z"}
(NOBRIDGE) LOG  Saving message to database...
(NOBRIDGE) LOG  Message saved successfully: {"encrypted_message_for_receiver": "mFLqdRNX2A6f5PxEMCDLtFDogAe+r0fKTvXVZsT9CpK3hbZGodN6a0pm0fjHBdKdj5Y75PDSf9zx9ZR5RZaBQqUYffaCgJd6h
5MigkntDs5q+51DzU2qg2FI0OWWcb5yQe5k5WyJBH7XbZIH5tloHN7AXR8OLDDylyx+eo/9Cabd+y/Rv0Sx1EYLfUjEWxpucTkouzLtmqAr1FM7fKq53WfB25R7dhkA+FavTioPdqd6/ys5mBU/3+F5du/sg1MmeLO1JVei3REHWxs5BNM
iOUUr5ofxqcXJD0/LRL1sFbrjdyswsNxfU+i9oYFrmWOz5TWZ8It4U01SwzwqAe3qUQ==", "encrypted_message_for_sender": "X92gJix/9BSsBrOSZsdLFqPPbfiVQ6FNPCe2mQEzTbx/qfzyeo89FCCPQK5dLGPlNZw78gJ5S
zmX6GQRbWglAlSKJaIxftUhEwUp0bPQwquheUaJSam9fvkktnXRbUYuysx2muwGtUqa5el03DE+plC9is0kdDf778UaD4K3oDCaYtcE2gD8NmmaN0gEQgOLEE3RZ1+t+fMypG/spCwCsefGZWV9cNXOJI+elRbYlblv/p9ryOlyF+HppEx
7YWoMQwlXQYKM03jFvgHjAX5LonGcCGpSqAxfVsxdWEJS5A6vuTDZitcQz/5i1HjbvhqLL0UPN2DjCLv9fBAgh1Hzeg==", "id": 16, "receiver_id": "8e7d3789-711c-40a9-b696-61dfe8cde01d", "sender_id": "ca6
25bf8-b637-41dc-ac6a-b1814248b7b4", "timestamp": "2025-04-22T15:29:22.757739"}
```

Here is how it looks on the App:

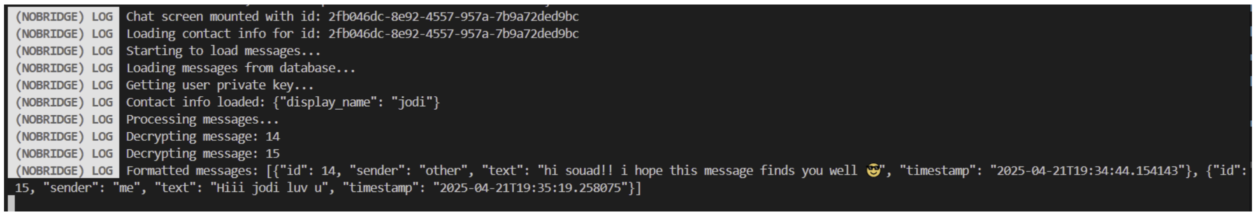

11:29

Chat Shiny

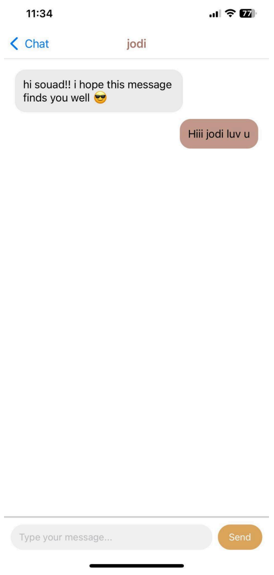
Hii shiny! How are you doing?

As you can see, it sends the message to the UI and then encrypts it using the public key, and successfully saves it to our messages database.

When you try to open a chat that contains previous conversations, this is the loading process:


```
(NOBRIDGE) LOG  Chat screen mounted with id: 2fb046dc-8e92-4557-957a-7b9a72ded9bc
(NOBRIDGE) LOG  Loading contact info for id: 2fb046dc-8e92-4557-957a-7b9a72ded9bc
(NOBRIDGE) LOG  Starting to load messages...
(NOBRIDGE) LOG  Loading messages from database...
(NOBRIDGE) LOG  Getting user private key...
(NOBRIDGE) LOG  Contact info loaded: {"display_name": "jodi"}
(NOBRIDGE) LOG  Processing messages...
(NOBRIDGE) LOG  Decrypting message: 14
(NOBRIDGE) LOG  Decrypting message: 15
(NOBRIDGE) LOG  Formatted messages: [{"id": 14, "sender": "other", "text": "hi souad!! i hope this message finds you well 😎", "timestamp": "2025-04-21T19:34:44.154143"}, {"id":
15, "sender": "me", "text": "Hiii jodi luv u", "timestamp": "2025-04-21T19:35:19.258075"}]
```

Here it is on the app too:


11:34

Chat jodi

hi souad!! i hope this message finds you well 😎

Hiii jodi luv u

It collects all the encrypted messages from the database and decrypts all of them before sending it to the UI for the user to read them! In this example, there were only two messages in the conversation (message id 14 and 15). However, if conversations get very long, it can take some time to decrypt all the messages that are in the database and load up the conversation.

Potential Security Flaws

While Tea Time does have a strong RSA e2e encryption algorithm, there are some areas of vulnerabilities that we might have to address in the future. The biggest issue we have at the moment is that both public and private keys are currently being stored in our Supabase DB. For our app, it made it very simple to access keys during the development process. However, it can be a major security concern, especially if the database were ever compromised, attackers could access users' private keys and decrypt their messages. In popular messaging apps, private keys never leave the user's device and are not stored in a central database. Another potential security concern is the fact that user's RSA keys are long-lived, meaning that once a key is assigned to a user, it never changes. This is dangerous because if a user's key was ever compromised, all previous messages can be easily decrypted. Currently, there is no system implemented to allow for key changes/rotations or even session-level keys to prevent this issue.

Mitigation Strategy Proposals

To improve the security of Tea Time in future versions, the first (and most important) step would be to move private key storage away from the centralized database and onto the user's device. This can be done using secure storage mechanisms such as the iOS Keychain or the Android Keystore. By keeping private keys locally and never transmitting them to the server/DB, we can ensure that only the user has access to decrypt their messages, even in the event of a server or database breach.

On top of that, we could also implement a system where keys are automatically regenerated or updated after a certain amount of time. We can also allow users to trigger these regenerations manually, especially if they suspect that their keys have been compromised. Another thing we could do instead of regenerating keys after a certain time limit is to have user-session based keys. With this strategy, a temporary key pair is generated for each login session, and messages exchanged during that session are encrypted with those keys. The session keys can either be discarded after logout or expire after a set time period. The biggest benefit of this strategy is that even if someone's key is compromised, previous messages will remain protected because they were encrypted with short-lived session keys that are no longer accessible.