Network Security (5473)

Project: Encrypted Messaging

Nebras Alnemer · Andrew Miller · Ted Zhu

Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210

alnemer.3@osu.edu, zhu.744@osu.edu, miller.5984@osu.edu

1. Abstract

Text messaging is a common method of communication, along with various other techniques. With the massive amount of data flowing through the network, it is somewhat assumed by most users that their messages will arrive to the intended recipient safely, without worrying whether or not a third party had gained access to their messages. There is plenty of desktop software and mobile applications to add an extra layer of security to a user's communication, yet the user must assume that the software works as intended without visual proof that their data is encrypted. The purpose of this project is to develop a mobile application to transfer an encrypted text message from one device to another via a central server, and demonstrate proof that the data was encrypted as intended. This report will outline the design and implementation of our application, thoughts on the results, and limitations and future enhancements.

2. Introduction

The purpose of this messaging application is to successfully send encrypted audio data from one device to another, and show proof of concept that the message was in fact encrypted. A user may create a text message their mobile device and choose to send. When the user hits send, the message is encrypted first, and then is sent to the server to be received by the other device.

The novelty of this project is that it will use end to end encryption, so that only the sender and receiver can understand the message being sent. This system will allow a sender to send a message to a recipient without fear of using a network that is open, such as a public wifi hotspot. The message will be encrypted before it even reaches the network, so there will be no way to recovering the message without knowing the secret keys. To demonstrate encryption process, the keys are manually sent by the user to the intended recipient. When the recipient receives a key, it is stored and can be used to decrypt future messages from the person that had sent the key.

Keys and messages are stored on a central server. The intended purpose of this server is to store messages and devices, along with associated IDs. These IDs are used to route messages from the sender to the recipient. All messages are encrypted, and keys are padded to improve security. Encryption schemes will be discussed later in the report.

3. Background

Considering the amount of information that flows through the internet, the concern for security and privacy has risen over the years. It's possible for a third party to eavesdrop on the communication of two other parties, and

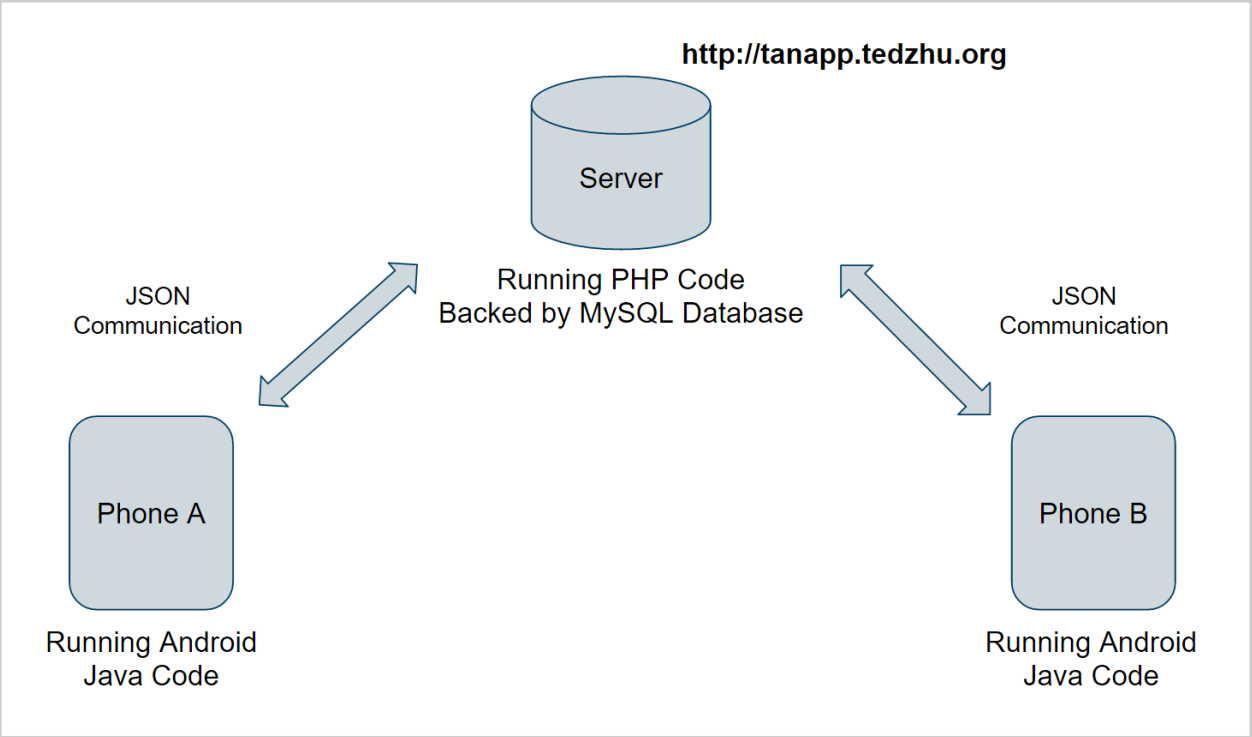

Figure 4.1: The Android Device to Server Communication Scheme.

learning how to prevent such attacks is important to many businesses, as well as individuals. Encrypting data that is being sent over the internet is perhaps the most common method to try and prevent eavesdropping. There are various techniques for encrypting information, and these techniques only get better as research and interest in privacy grows.

The reasoning for this application is to demonstrate the encryption process. All though there are existing applications for encryption, this is an attempt to visually show what encrypted data looks like while it is in transit, and how it is stored before it reaches its destination. This application also attempts to demonstrate the difficulties of key exchange, which is vital to the security of data.

4. Design

Two phones do not directly communicate with each other. Rather, their communication is mediated by an application server which both phones communicate with. The server runs on PHP code with a MySQL database backend, and is set up so that it can send and receive JSON data with the phones for transferring information about devices and messages.

Database Scheme

The database stores 3 tables: devices, messages, and dbpurges, as shown in Figure 4.2. The two primary tables, Devices and Messages store the actual information supporting the Encrypted Messaging app. Each user is represented as a Device, because we expect encryption keys to be stored on a per-device basis rather than per-person basis. The devices are unique identified with an auto-incrementing integer, and the values are used in the `sender_device_id` and `recpient_device_id` fields of a message record.

Each message record also includes the type that it is, "KEY" or "TEXT", and the actual encrypted body of the message. It is up to the implementing clients of the Android Messaging App to use the type and data_text fields as appropriate in the designed encryption scheme.

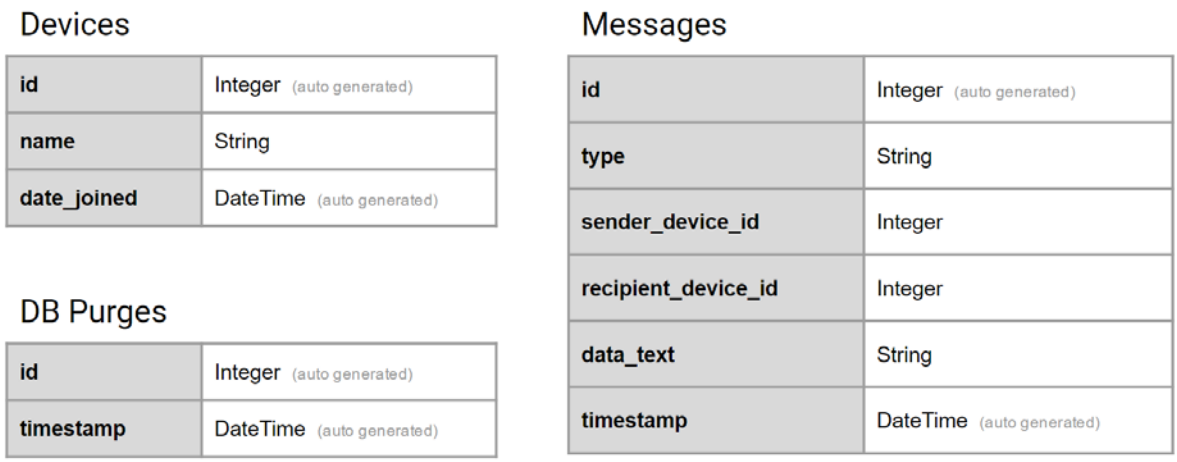

Figure 4.2: The database scheme organization for the application backend.

JSON

The data format used for communication with the server was JSON. JSON stands for JavaScript Object Notation, and is a structured way to handle and parse data. The server stores its data as JSON, and when a device requests data, it will parse the receiving data as JSON and will be able to extract the key information that it is looking for.

Communication

When a new user registers a name, the name gets added to the server along with an associated ID. The registration of the name is done via a POST request, and the response of the POST request contains the given ID. This ID is stored on the local device so the device will be able to tell which messages are being sent to it.

When a user attempts to send a message or a key, the application will send a GET request to retrieve the list of registered names. The response of the request will contain the names and device IDs of each user. The names of each user is populated in a list that the sender is able to select from. After selecting the intended recipient, the message is sent via a POST request is added to the server. The payload contains the message contents, the recipient ID, and the sender's ID.

When another user attempts to retrieve messages or keys, their application will send a GET request and retrieve the list of messages from the server. The application then searches the list of messages and extract all the messages that contain that device's ID in the intended recipient field.

Home Page

The homepage for the server outputs the data from the database in a formatted, human-readable tables which enabled diagnosing issues during development much easier. This also helped to show the inner workings of our implementation during the final presentation demo of our project. Each table's header was also a link which outputs the data in JSON format, which also aided in development, so we could see the exact spelling of the database fields, showing examples of how the Android clients should format their POST request bodies so that the information had matching spelling for their fields.

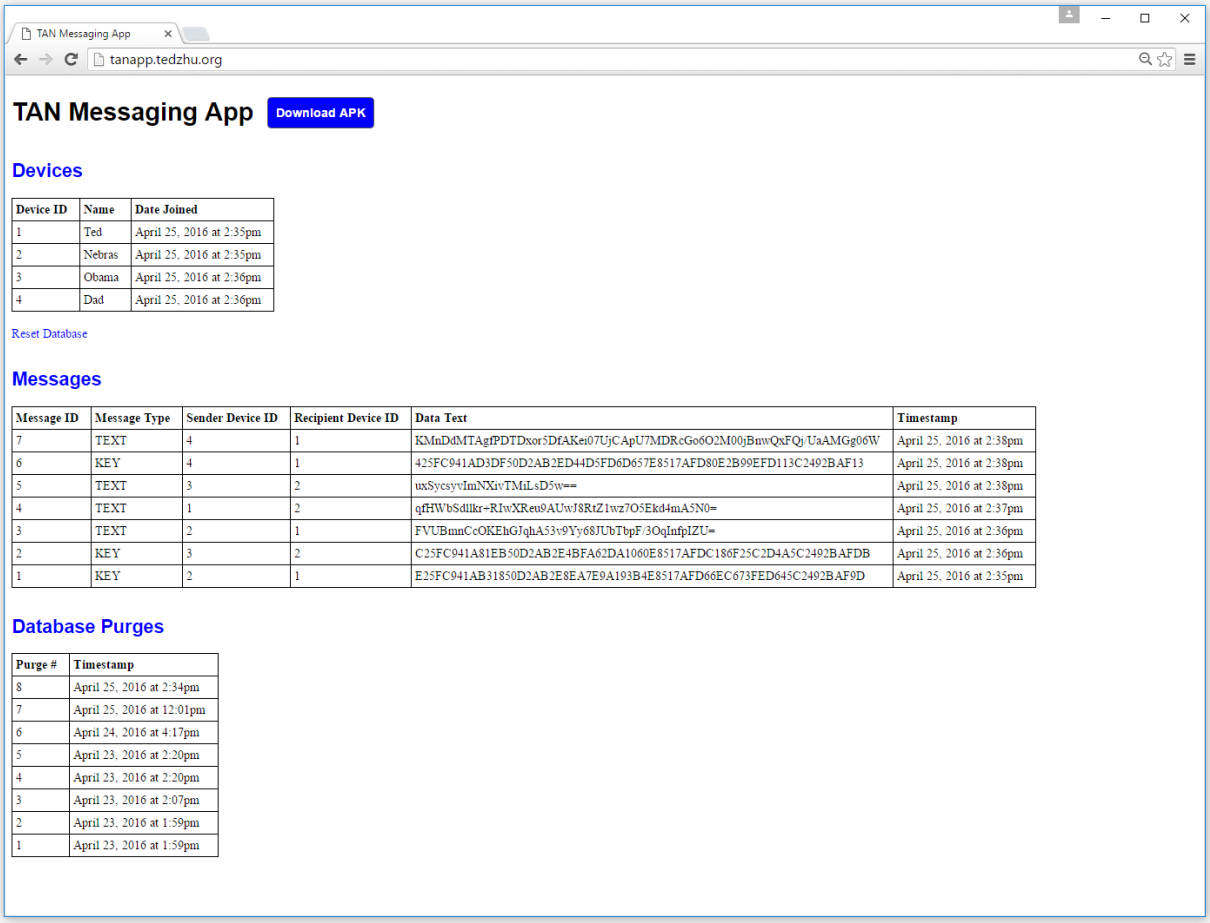

Figure 4.3: The home page of the TAN Messaging App.

5. Encryption and Decryption Process

The novelty of the application is that the message that is sent is never saved to the sender's device. The message itself is encrypted and then sent, and all traces of the message are gone. Due to complications with the encryption process, namely the lack of available cryptographic libraries that were robust enough to ensure close to perfect secrecy, the encryption process implemented is not as secure, but mitigation steps were taken in order to lessen the chances of success for an adversary listening to the message being sent.

Key Generation

The application is built under the AES cipher, utilizing a 128-bit AES key for encryption and decryption. Early in the development of the application, we discovered the difficulty in sending and receiving a key; Diffie-Hellman was the preferred method, given the key is generated and agreed upon by both devices, rather than one device generating the key and sending it to the recipient device.

We used the javax.crypto library initially, and attempted to use the SpongyCastle library that had been developed in recent years since the Java library was no longer robust enough to be secure. We then discovered that SpongyCastle was not compatible with Android Studio, the IDE that we were using for development. This

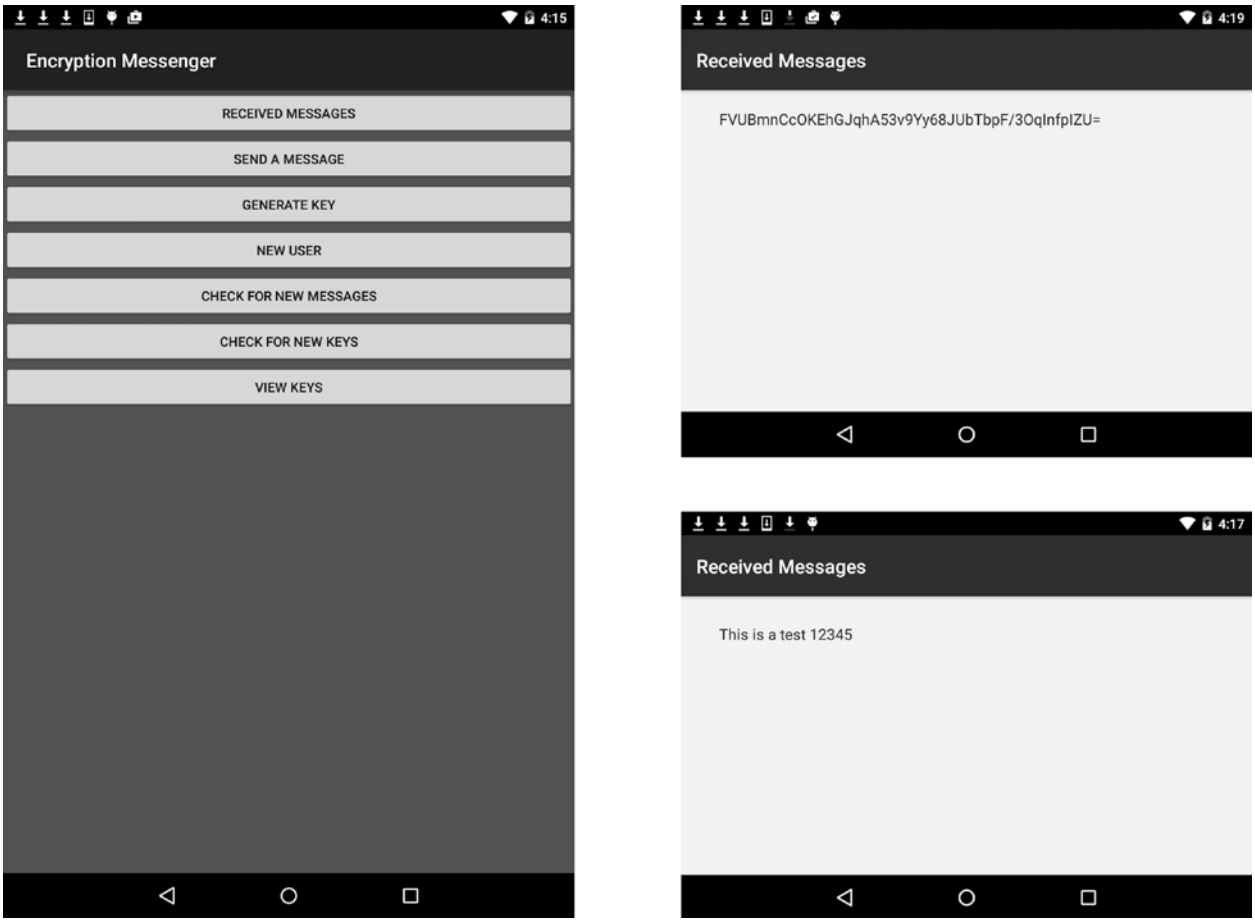

Figure 5.1: Three app screens showing the main menu, and a message before and after decryption.

resulted in requiring the application to be built under the Java library and modifying the implementation of the application to our needs. The key generation portion was fairly straightforward; javax.crypto has a key generation algorithm that only requires a key type delimiter to generate the key. We decided ultimately to use a symmetric encryption scheme. In this case, we input the delimiter as "AES", and the generation algorithm did the rest.

To sent the key, however, was a different problem. With no way to use Diffie-Hellman for key agreement or a way to encrypt the key in a way the recipient device could decrypt it and use it, we had to ensure two things: first, that the key is able to be received and decrypted by the recipient 100% of the time and continue to be able to use that key easily, and second, that the key would be very difficult to figure out if an adversary got a hold of it. The first problem was very simple to solve. To ensure safe keeping of the key on both the sender's and receiver's device, we implemented Shared Preferences, a type of local database that can store the primitive types of data such as strings, longs, ints, etc, in key-value pairs. Since the key that is generated is a byte array, the key needed to first be changed into a hexadecimal string. After the key was converted, it could then be saved as the string to the generating device under any name desired.

The requirement after it was generated was to then send it to the recipient, but under the previous problems mentioned, there was no way to guarantee that the recipient could decrypt the key if we had encrypted it. The only option left was to send it as a plaintext string. This is by no

means secure in any way, so we took a few mitigation steps. Since the key was not instantiated as an AES key that could decrypt an AES encrypted input, an attacker would have to know we were using the AES scheme. In addition, we padded the string with another 32 bytes of data to make the full length end up being 64 bytes. This was to make it so that if an attacker managed to get the message containing the key, they would have to know we were using a 128-bit key, not a 256-bit key. The key was also padded at different intervals throughout the key, not just appended to the end. They would also need to know where the key was modified and remove the added strings. We decided that this was enough for the application to be slightly stronger against attacks at an initial implementation.

Therefore, the process of sending the key is: first, the key is generated and saved to the sender's device. Second, when the key is sent, it is immediately padded with the extra strings and sent to the server to then be sent to the recipient. When the recipient has the key from the server, the extra strings are removed and the resulting key is saved to the device.

Encryption Process

The encryption process was fairly straightforward with the use of the javax.crypto library. To encrypt a message, the message must be converted into bytes and then run through the desired cipher using the generated key, which must be converted into the correct key specification. In our application, as outlined earlier, we take in a string of text as input to the encryption method. Since the key was saved as a string, we have to take the string out of the database and then reconvert it into the byte array to be fed into the encryption algorithm.

The string is converted in the reverse process that it was converted into a string, and then it is specified as an AES key. To input a text message, the user must press the "Send Message" button and select "Enter Message" on the following screen. They are then given a popup window to input their message. Once that is finished, we encrypt the message by using the message's bytes and the key we generated previously. The output of the cipher is in bytes, so we must once again convert the bytes into a string to be sent. To do this, we encode the byte array into a Base 64 string and send it to the intended recipient. The encoded message is sent to the server where it is rerouted to the intended recipient.

Decryption Process

The decryption process is the reverse of the encryption process, however we must then show the decrypted string to the recipient. To first receive the message, the receiver must make a request to the server for any new messages. After the request, they may go into the "Received Messages" section of the application to view any messages they have received. If they have received a message, it will display as the Base 64 encoded string that was sent by the sending device and is shown in a list view. To decrypt this message, they tap on the message that is displayed. The application then pulls the string from the specific message they tapped, and then it pulls the key from the local database that was saved by the device.

Similar to the encryption process, the receiving device converts the key into a byte array and instantiates it to an AES key. It takes the byte array of the key and the message it pulled as input into the decryption algorithm of the AES cipher, and if the key is able to decrypt the message, it displays the decrypted message where the encrypted message was moments ago. If the user exits the received messages view, the message is re-encrypted. If the key is unable to decrypt the message, nothing happens to the message being displayed. If a user has changed the key that they used for those previous

messages, the messages are lost forever. There is no way to get the previous key back, and no way to decrypt them.

Possible Future Implementations

The manner in which the key is sent, namely as a plaintext string, is clearly not a secure method of sharing keys. We attempted to implement some mitigation steps to help reduce the chance that an attacker would be able to recover the key, but if they are able to see the code and determine where to remove the padded strings, they will easily recover the key. The desired course would be to use a Diffie-Hellman approach, and with more time to implement it, we would use this method over all others. In addition, a user is able to see all previous messages.

Ideally, after exiting the "Received Messages" view, the device should delete all messages it previously received. This would safeguard against any attempts to use an old key with older messages. In addition, all messages and keys are saved on the server. Once the recipient receives the key, the server should delete the key so there is no record of it anywhere except on the receiver and sender's device. Since the key is not used on the internet, there is no way to recover the secret key, meaning the scheme would be extremely secure if the keys are shared in a more secure manner.

6. Conclusion

We learned a lot through this experience of designing and implementing an encrypted messaging app. We encountered a lot of obstacles and difficulties to get a final working product, and we experienced just how tough both ends of designing a message encryption protocol, and the implementation of the application can be.

The biggest difficulty we faced was the problem of key distribution, because we intended in the messaging protocol that only the two devices would store the secret key. To create such a protocol on our own was beyond the scope of our abilities and time provided for the project. We opted for our own obfuscated encryption format for sending keys between devices, which needed to be stored on the server if we wanted to enable the receiving device to get the key at a later time

On the implementation side, we had to deal with all the little details that got in the way of a fully integrated Android messaging app with a server database backend. We found it was best to keep all the details and code related to encryption/decryption on the Android client, where the server was just a vehicle for sending messages of a format the server was blind to. This allowed the team member(s) working on the server to stay focused on a simpler task, and the team member(s) working on the Android App could use the database server as a low-level service with which to build an encrypted messaging application on top of that.

7. Acknowledgements

Ted primarily handled the server side of things, setting up the PHP server, and instantiating the database. He wrote the PHP scripts that facilitated the JSON communication protocol between phone and server. Andrew and Nebras setup the Android app code, and were the primary authors for the Android app. Nebras helped to establish GET and POST communication, as well as parsing of JSON data from the Android end. Andrew handled the encryption and decryption of messages and key distribution on the Android end. Ted formatted the final report. We would like to thank Dr. Yinqian Zhang for teaching the network security course, and providing us guidance on the project.