

MIT: 6.857

FINAL PROJECT REPORT

---

**Messngr: End-to-End Encrypted Messaging Application With  
Server-Side Search Capabilities**

---

*Authors:*

Marcos PERTIERRA (marcosp@mit.edu)  
Dennis GARCIA (dagarcia@mit.com)  
Anthony ERB LUGO (aerblugo@mit.edu)

*Professor:*

Ronald RIVEST

*Teaching Assistant:*

Cheng CHEN

May 24, 2016

## **Abstract**

This project implements a proof-of-concept app that provides strong end-to-end encryption for chats and allows users to search through their encrypted messages without the server learning the contents any message nor what keyword the user searched for.

Messngr App: <https://searchable-encrypted-messaging.herokuapp.com>.

Messngr source code is available here: <https://github.com/Messngr/Messngr>.

# Contents

<b>1</b>	<b>Background</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	Previous Work . . . . .	2
1.2.1	End-to-End Encryption . . . . .	2
1.2.2	Searching Over Encrypted Data . . . . .	3
1.3	Threat Model . . . . .	3
1.3.1	Compromised Server . . . . .	3
1.3.2	Malicious Users Analyzing Network Traffic . . . . .	3
1.3.3	Attackers Well-Versed in Launching OWASP Attacks . . . . .	4
<b>2</b>	<b>Design &amp; Implementation</b>	<b>5</b>
2.1	System Design . . . . .	5
2.1.1	Account Creation and Login . . . . .	5
2.1.2	Chat Creation . . . . .	6
2.1.3	Messaging . . . . .	7
2.1.4	Keyword Searching . . . . .	7
2.2	Implementation . . . . .	10
2.2.1	Database Storage . . . . .	10
2.2.2	Authentication . . . . .	10
2.2.3	Client-Side Storage . . . . .	10
2.2.4	End-to-End Encryption . . . . .	11
2.2.5	Searchable Encryption . . . . .	12
<b>3</b>	<b>Analysis &amp; Discussion</b>	<b>14</b>
3.1	Issues . . . . .	14
3.1.1	Possible Vulnerabilities . . . . .	14
3.1.2	Performance Issues . . . . .	15
3.2	Future Work . . . . .	16
3.3	Conclusion . . . . .	16
3.4	Acknowledgements . . . . .	16

# Chapter 1

## Background

### 1.1 Introduction

Recently, WhatsApp, a popular messaging app, introduced end-to-end encryption capabilities for messages sent on its service [1]. End-to-end encryption is powerful because it makes it impossible for the provider to learn the contents of the messages sent. However, while this improves privacy it forces the service to limit some potential functionality such as allowing the user to search through their messages on the server. If the user can't store all of the messages sent locally then the user loses the ability to search through those messages. Ideally, there would be a web app that allows users to send messages to each other, encrypted end-to-end, but with the added ability of searching through their messages. Thus, we present the design and implementation of Messengr, a simplified version of the searchable encryption scheme in “Practical Dynamic Searchable Encryption with Small Leakage” [2], that provides searching capability over end-to-end encrypted messaging through the form of a web application.

The rest of the paper is organized as follows. Section 1.2 presents previous work that has been done in the field of end-to-end encryption with regards to messaging as well as work done in implementing search functionality over encrypted data. Section 1.3 introduces our threat model and what kind of attacks we are defending against. Next, Chapter 2 presents an overview of the system and its implementation. This chapter includes the implementation of the app as well as the end-to-end encryption and searching protocols. Finally, Chapter 3 talks about potential issues in our app such as potential attack vectors and app performance. This chapter then concludes the paper.

### 1.2 Previous Work

#### 1.2.1 End-to-End Encryption

As previously mentioned, end-to-end encryption applications exist and have recently been made widely available through services such as WhatsApp's messaging service and Apple's iMessage service. These applications come as an answer to a consumer want for more privacy in the modern social world. While providing a much needed increase in privacy, these services lack functionality which would otherwise be available. For example, if a device cannot store all of the decrypted messages for a conversation locally, then the device cannot ask the provider to search through the encrypted messages. This is intentional, of course, since the messages are encrypted with

the goal of limiting the amount of information that the provider could possibly extract from the messages. Unfortunately, this can sometimes limit the usability of an application. Our messaging app fills this gap by implementing a protocol that allows the user to search through their encrypted messages securely with the help of the remote application server.

### 1.2.2 Searching Over Encrypted Data

Searching over encrypted data is an idea that has been around for at least 15 years [6] but which has yet to be applied in popular secure communication applications. Some applications bypass this issue by offloading the search functionality onto the device, which is the case with both iMessage and WhatsApp, and this has proven useful. This, however, is not always ideal as certain computations may be too hard to be done locally on devices. To consider this case, we investigate the use of searchable encryption on end-to-end encrypted messaging schemes.

With regards to searchable encryption, we can consider our case such that messages sent by users are considered as documents, encrypted using symmetric encryption, and stored on a remote server. This case lends itself to [2], which provides a sophisticated dynamic searchable encryption scheme over documents. The work introduces a scheme that can handle searches over the documents even after documents have been deleted from the collection or inserted into the collection. The protocol, while useful, can be simplified to handle just insertion cases and run with reasonable algorithmic running time overhead.

Pmail [7], a project which dealt with providing search functionality over encrypted emails on Gmail, implemented a similar searchable encryption protocol. Messengr is inspired by Pmail's design and builds off of it in order to provide this service to more modern forms of communication such as instant web messaging. This project also differs from Pmail in that it implements the necessary underlying end-to-end encryption protocol with ease for the user in mind. Moreover, this project handles the key sharing aspect for the user in a secure and reliable manner.

## 1.3 Threat Model

Before we dive into an overview of the design and implementation of Messengr, we first discuss our main adversaries and threats:

### 1.3.1 Compromised Server

A compromised server in which the attacker who was able to compromise it can read user information stored in the database. This includes data such as usernames and more sensitive data such as passwords and messages between users. We decided on this semi-compromised server state as we reasoned that Messengr would be open-source and thus users and developers would notice if anything was amiss with the implementation of the server itself.

### 1.3.2 Malicious Users Analyzing Network Traffic

Malicious users analyzing network traffic via a packet sniffing tool such as Wireshark. As mentioned in the previous section, this form of adversary is particularly detrimental to a messaging app as the privacy of users in communication with each other is ruined if messages were to be unencrypted.

### **1.3.3 Attackers Well-Versed in Launching OWASP Attacks**

Attackers that can exploit common web vulnerabilities such as cross-site scripting, cross-site request forgery, or bypassing authentication and session management. Because Messengr is a web application, we acknowledge that an attacker with enough time and resources can find non-trivial bugs and exploit some of these common vulnerabilities.

In order to mitigate threats from these adversaries, we designed Messengr to provide end-to-end encryption between users, and provide a wide variety of other security features. The exact details of how we achieved these features are made clear in the next two sections, where we expand on the design of the application as well as the details of its implementation.

# Chapter 2

## Design & Implementation

### 2.1 System Design

In this section, we describe Messengr’s design at a high level. In Section 2.2, we go into details of how Messengr was implemented.

#### 2.1.1 Account Creation and Login

A user creates an account by simply entering a username and password in the login page after which a secret key and a public key are created for that user. The public key is sent along with the username and password-hash (see 2.2.2 for details, but essentially, the password-hash acts as the password) to the server. Additionally, the key-pair is encrypted using the password and this encrypted key-pair is stored in the server.

If a user already has an account, he can login using his username and password. The username and password-hash is sent to the server, and the server returns the user’s encrypted key-pair if the login is successful. The encrypted key-pair is then decrypted and stored in client-side storage. The user is then redirected to Messengr’s home page.

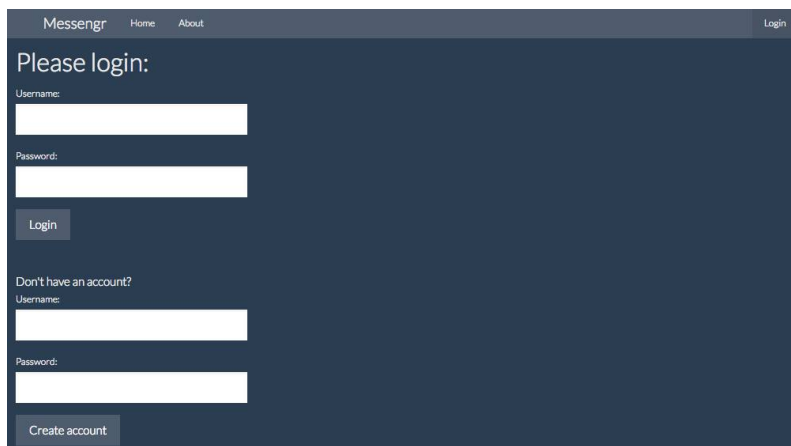


Figure 2.1: Login and account creation page.

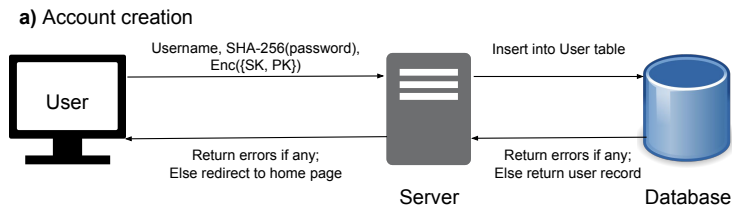


Figure 2.2: Account creation requests and responses.

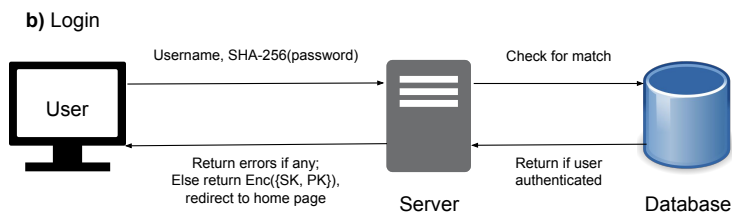


Figure 2.3: User login requests and responses.

## 2.1.2 Chat Creation

After logging in, a user is redirected to Messengr’s home page, where all the chats that the user is participating in are displayed in a table. Each table row links to the corresponding chat’s page. A user can create a new chat with any other user by simply specifying the other user’s username. Currently, Messengr only supports chats between two users.

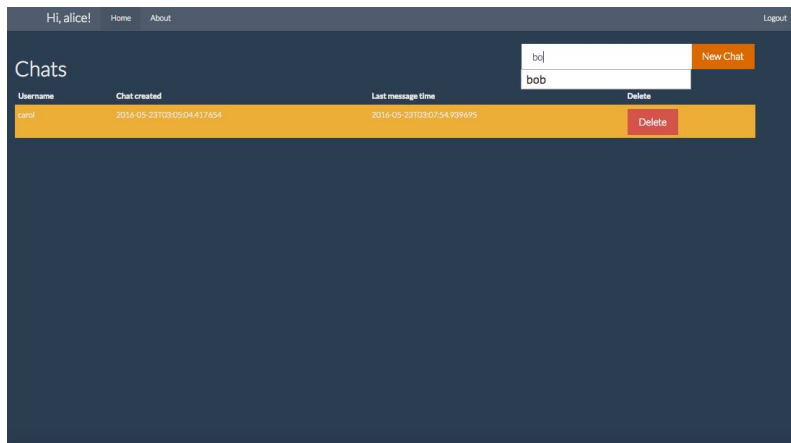


Figure 2.4: Home page. All existing chats are displayed here. To create a new chat, simply specify the other user’s username. The autocomplete feature makes this easy for users.

When a user attempts to create a new chat, the client asks the server for the other user’s public



key. Then, the client generates a symmetric key that will be used by both users to encrypt and decrypt messages in that chat. The client encrypts the symmetric key using the current user's public key and the other user's public key, and sends both of these encrypted keys to the server. If the chat is successfully created, the user is redirected to the chat's page.

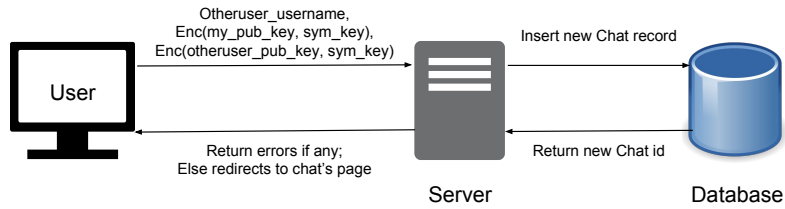


Figure 2.5: Creating a new chat. Encrypted symmetric key is stored on servers.

Whenever one of the two users in the chat visits the chat's page, the server sends the corresponding encrypted key to each user, so that each user can decrypt using his secret key. Thus, each user will have the shared symmetric key, but the server will not.

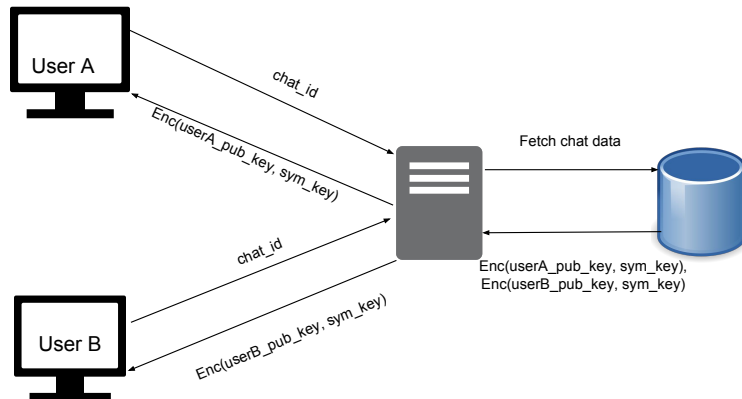


Figure 2.6: User A and user B visiting their chat page. Server sends them encrypted symmetric key, which is then decrypted by each, using their secret keys.

### 2.1.3 Messaging

Once a user is at a chat's page, the user can send messages. The plaintext message entered by the user is encrypted using the chat's symmetric key on the client before being sent to the server. The server stores the encrypted message with a reference to the chat. When the other user in the chat accesses the page, all encrypted messages are fetched from the server and can be decrypted using the chat's symmetric key.

### 2.1.4 Keyword Searching

In order to perform keyword searches over the encrypted messages, our application needs to do preprocessing on the messages on the client side so as to construct an inverted index which will allow the client, with the help of the server, to then perform searches. The steps to process each message are complicated and include constructing encoded pairs which are then stored on the

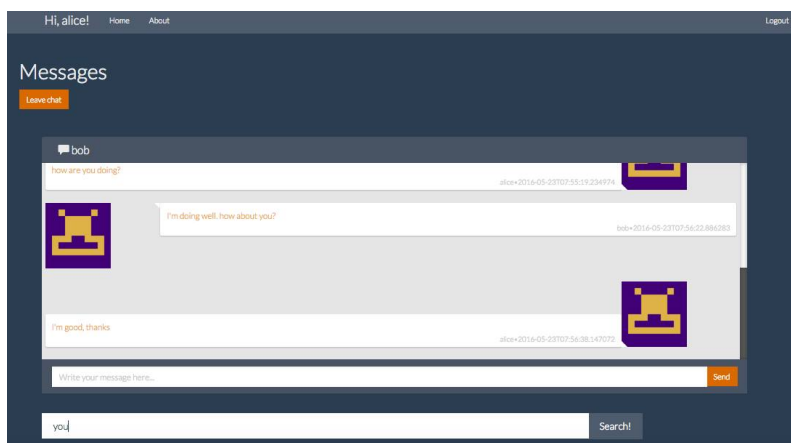


Figure 2.7: Chat page showing decrypted messages between users alice and bob.

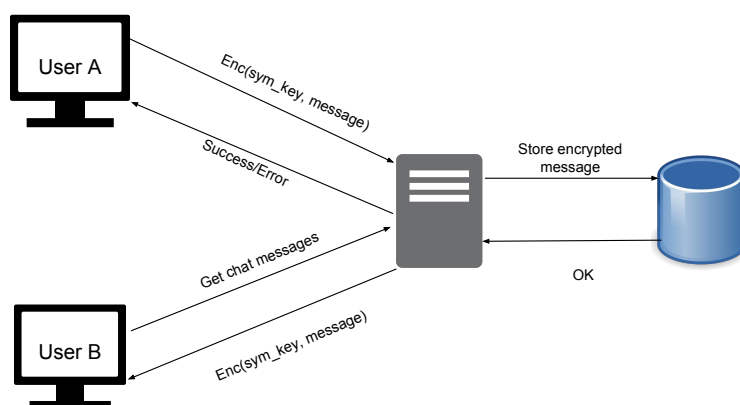


Figure 2.8: Messages are encrypted by the sender using the shared symmetric key and then sent to the server to be stored. The receiver fetches the encrypted messages and decrypts them using the same key.

server. This extra construction is explained in the implementation section. For now, in Figure 2.9 we show an overview of the communication between the client and the server when performing the preprocessing.

Message preprocessing is handled for each message as it is received by the client. This allows the database to be up to date with all the necessary information for any given search request. Next, the process for a search request is as follows: the client enters a keyword that he or she wishes to search for and then a token is produced from the keyword as well as the symmetric key used for encryption in the chat conversation being searched over. Figure 2.10 shows the process flow of a search request in our app.

Given this overall view of the system, we can now examine the implementation of our app.

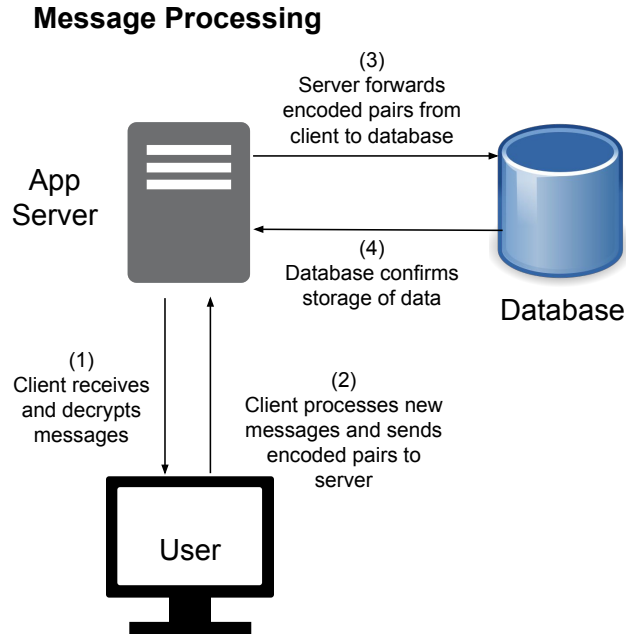


Figure 2.9: Message processing for later use in performing keyword search.

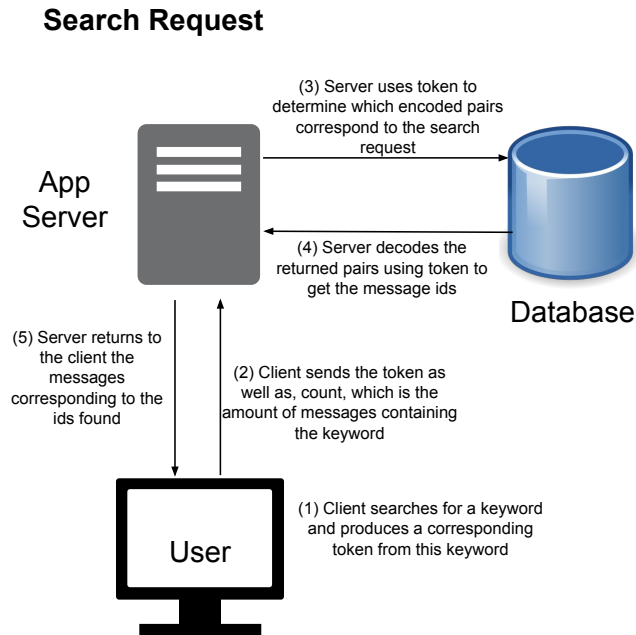


Figure 2.10: The process of handling a search request with our encrypted search protocol.

## 2.2 Implementation

We coded our web app in Python using the web framework Flask<sup>1</sup>. This allowed us to use many useful extensions to rapidly develop a prototype, such as Flask-SQLAlchemy<sup>2</sup> to construct our data models, Flask-SocketIO<sup>3</sup> to enable real-time messaging, and Flask-SSLify<sup>4</sup> to redirect all incoming requests to HTTPS. For client-side cryptography, we used Stanford Javascript Crypto Library (SJCL) [3] to generate public/private key pair, encrypting and decrypting chat messages, and message-processing for encrypted searching that is further described in Section 2.2.5.

### 2.2.1 Database Storage

We use Postgres for our database. We have a ‘users’ table that stores each user’s id, date-time of creation, username, password-hash, public key, and “encrypted user-data” (which contains the user’s secret key and other data necessary for searchable encryption further described in Section 2.2.5); this user data is encrypted using the user’s password on the client-side. We keep a ‘chats’ table that stores each chat’s id, date-time of creation, date-time of last message, and the ids, usernames, and “encrypted-symmetric keys” of both users in chat; by “encrypted-symmetric keys” we mean the symmetric key that is shared by both users, encrypted with each user’s public key. We have a ‘messages’ table that stores each message’s id, date-time, encrypted text, sender (id and username), receiver (id and username), and chat id. There is also an ‘encoded\_pairs’ table that stores a pair id, a hash key, and a hash value; these values are described in detail in Section 2.2.5 .

### 2.2.2 Authentication

Each user logs in with a username and password. The server never has access to a user’s password; instead, we compute the SHA-256 hash of the user’s password on the client-side and send that to the server. Then, on the server, this hash is hashed with a randomly generated salt using the bcrypt package<sup>5</sup>. The password-hash (that is, the bcrypt hash of the SHA-256 hash of the password) and the corresponding salt are stored in the database. The reason we send the SHA-256 hash of the password is so that the server never has access to the user’s plaintext password, which would be used to decrypt the “encrypted user-data” and attain the user’s secret key.

Every route in our app ensures that a user is authenticated (using a Flask session) and can only learn information about chats he/she is participating in and can only have access to messages from chats he/she is participating in.

### 2.2.3 Client-Side Storage

For convenience, each user keeps his keys in sessionStorage (as well as the other message-processing data described in Section 2.2.5). This sessionStorage only lasts as long as the browser is open [5], and we also make sure to clear it when the user logs out. The alternative would be to keep these keys encrypted in sessionStorage and decrypt whenever the application needs to encrypt or decrypt

---

<sup>1</sup><https://github.com/pallets/flask>

<sup>2</sup><https://github.com/mitsuhiko/flask-sqlalchemy>

<sup>3</sup><https://github.com/miguelgrinberg/Flask-SocketIO>

<sup>4</sup><https://github.com/kennethreitz/flask-sslify>

<sup>5</sup><https://github.com/pyca/bcrypt/>

a message, or process new messages. However, assuming the user's unencrypted password is not being cached, this would require prompting the user for his password fairly frequently, which would make the app less user-friendly and possibly, unintentionally encourage users to choose weaker passwords. We discuss the possible vulnerabilities caused by this implementation decision in Section 3.1.1.

## 2.2.4 End-to-End Encryption

Messengr uses the elliptic curve NIST P-256 [4] to implement end-to-end encryption of messages. Each chat consists of two users. Each message sent in the chat is encrypted by the sender and decrypted by the receiver, using a shared symmetric key.

### Key Generation

SJCL lets us generate a public/private key pair with a simple line of code:

```
var pair = sjcl.ecc.elGamal.generateKeys(256);
```

We then serialize the keys and store them in sessionStorage, client-side storage [5].

### Symmetric Key Sharing

When a user, wants to create a new chat, he/she specifies the other user and asks the server for this other user's public key. The server responds with the other user's public key. The first user randomly generates a 256-bit, secret, symmetric key and encrypts it using his public key to get  $sk_{sym\_1}$  and also encrypts it with the other user's public key  $sk_{sym\_2}$  and sends both of these to the server. Now, whenever the two users want to enter their chatroom, they are handed their corresponding encrypted-symmetric keys, which they can then decrypt using their secret key to get their shared symmetric key. The code below shows how this is implemented in the client-side Javascript using the SJCL library.

```
// Generate symmetric key (8 4-byte words = 256 bits)
// 10 is maximum paranoia level
var symmetric_key = sjcl.codec.base64.fromBits(sjcl.random.randomWords(8, 10));
// Encrypt it using my public key
var sk_sym_1 = sjcl.encrypt(my_public_key, symmetric_key);
// Encrypt it using receiver's public key
var sk_sym_2 = sjcl.encrypt(receiver_public_key, symmetric_key);
```

When a user enters one of his chatrooms, he receives the secret symmetric key that has been encrypted using his public key,  $ENCRYPTED\_SYM\_KEY \in \{sk_{sym\_1}, sk_{sym\_2}\}$ . Thus, he can easily decrypt using his secret key.

```
symmetric_key = sjcl.decrypt(secret_key, ENCRYPTED_SYM_KEY);
```

## Message Encryption and Decryption

Now that each user in the chat holds their shared symmetric key, each message is encrypted using this key before being sent to the server.

```
var encrypted_message = sjcl.encrypt(symmetrical_key, message);
```

The server forwards the encrypted message to the other user in the chat, who can then decrypt using the same key.

```
var decrypted_message = sjcl.decrypt(symmetrical_key, encrypted_message);
```

### 2.2.5 Searchable Encryption

The protocol for providing the search functionality has two main parts: message processing and handling of search requests. We process each message locally on a user's device where it can be decrypted and handled in its plaintext form. From this processing we can produce an encoded inverted index which we store on the database such that the server is unable to decipher the inverted index. Next, when a search request is made, we can send a search request along with a token that allows the server to return the parts of the encoded inverted index which correspond to the search request. Below we describe these steps in detail.

#### Message Processing

As a user opens a chat or reads new messages on the browser, our client-side Javascript detects which messages have already been processed through the session storage functionality and then processes the new messages. A new message is processed by extracting the unique words from the message along with a message identifier, the id that the server's database uses to refer to it, and encoding these pairs to send to the server. This encoding process is the key to how the search process works. After extracting the unique words and the message id, we have a list of unique words  $[w_1, w_2, \dots, w_k]$  and the message id. Next we produce  $k$  encoded pairs from the list of words and the id. The encoding process is as follows:

**Word, Message ID Pair Encoding** From the session storage on the client, we can extract the symmetric key, `symmetrical_key`, which is used for encrypting messages for this chat. Using this we can then encode each word,  $w$ , and message id pair for use later when searching. For the purposes of the search, we also keep track of how many times each word has appeared so far in a conversation. We then produce the encoding pair as a key-value pair such that the key can be used to find the pair in the database. We compute a token and then compute the key and value as follows:

Token:  $\text{HMAC}(\text{symmetrical\_key}, \text{sha256}(w))$

Key:  $\text{HMAC}(\text{token}, \text{count} + "0")$

Value:  $\text{id} \oplus \text{HMAC}(\text{token}, \text{count} + "1")$

Note that by using this token, it is possible for the server to find the encoded pairs when given this token as well as the corresponding count. This is useful because it allows us to keep the word which is searched for secret while still letting the server perform a search. All of these key-value pairs are computed for each message and are then sent to the server to be stored in the database, in the ‘encoded\_pairs’ table.

## Search Request Processing

On the client, when a search request is made for all messages containing a certain keyword, we can produce all of the information needed to perform a search on the server. Locally, we keep track of how many messages contain any given keyword and we can thus include this count with our search request. The search request is then made up of this count and a token which is produced in the same fashion as when producing the encoded pairs. Once the token and count are sent to the server, the server can then determine all of the encoded pairs which correspond to the search request. This is done by computing the key values of each encoded pair as such:

```
/models/encoded_pairs.py:
    hash_keys = []
    for i in xrange(1, count + 1):
        # Convert ith int to string and append "0"
        hash_key_message = str(i) + "0"

        # Produce key for each encoded pair from 1 to count,
        # using HMAC-256 and token
        hash_key = hash_utils.hmac256(token, hash_key_message)
        hash_keys.append(hash_key)
```

Next, we can search the database to find all encoded pairs which have these keys and extract the corresponding pair-values. We know that these pair-values will have the form:

Value:  $\text{id} \oplus \text{HMAC}(\text{token}, k + "1")$  for an integer  $1 \leq k \leq \text{count}$

Finally, computing the message ids from each of these pairs is done by applying the exclusive-or operation on the pair value with the corresponding  $\text{HMAC}(\text{token}, k + "1")$  value, which the server can compute since it has the token and  $k$  values from the request. The server then collects all of the message ids and redirects the user to a page which displays the contents of each of these (encrypted) messages which the user can decrypt locally on his or her device.

# Chapter 3

## Analysis & Discussion

### 3.1 Issues

Now that we've presented Messngr's system design and implementation, we now identify potential weaknesses in the application and which threats Messngr is not able to fully mitigate yet.

#### 3.1.1 Possible Vulnerabilities

##### Cross-Site Scripting Attacks

In Messngr's current implementation, the client maintains the current user's (unencrypted) keys in session storage while the user is logged in. Although this is convenient for the message processing and encryption/decryption of messages required for Messngr's end-to-end encryption and searchable encryption capability, this could potentially leave users vulnerable to cross-site scripting (XSS) attacks. Specifically, an injected script could attain a user's sessionStorage data and send it to an attacker; the attacker would then have access to a user's secret key. However, for an attacker to make use of this, he would have to first attain the victim's chat-data (encrypted-symmetric key) and messages from the server. There is no straightforward way to do this, as the server ensures that it will only ever send chat data to a user if the user is one of the two participants in the chat. That is, the only messages from the victim that the attacker could attain are the ones from a chat between the victim and the attacker; however, in such a chat, the attacker would obviously already be able to decrypt messages. Thus, for the attacker to gain access to the victim's messages (that are not sent directly to the attacker), the attacker would need the victim's password to authenticate as the victim. However, the attacker has not learned anything about the victim's password from the data contained in sessionStorage. Cracking a user's password is not trivial, as Messngr requires passwords to be at least 8 characters long, and must contain a lowercase letter, an uppercase letter, and a digit.

To protect against the possibility of XSS attacks, we ensure that all data sent to the server that might be displayed for users is sanitized and escaped. Due to Messngr's simplicity, we believe Messngr is reasonably protected against XSS attacks. The only data that the server accepts from a user that could be stored and displayed in other user's views is a username and a chat message. Thus, we implemented multiple server-side safety checks. A user can not inject a script by choosing a username

```
<script>stealYourSessionStorageData();</script>
```



as the server only accepts usernames that are composed of letters, numbers, dashes, and underscores. A user could attempt to send a message to another user that contains a similar malicious script. However, all messages sent and received are escaped before being displayed in a chat view. Thus, a user receiving a message containing a malicious script would only be vulnerable if the user's client-side Javascript had already been tampered with to leave the user unprotected by not escaping HTML strings.

In the case that the server is compromised by an attacker, there are two scenarios to consider. If the attacker is able to modify the source code, then decryption of messages would be trivial. The attacker could inject a malicious script into one of the views that asks the client to send not only the unencrypted sessionStorage data to the server, but also the plaintext password of each user. This would allow the attacker to then query the database for encrypted user-data and decrypt the user-data containing the secret key, which would then allow the attacker to decrypt the encrypted-symmetric key for each of the user's chats; finally, the attacker would use this key to decrypt all messages in the chat. However, if the attacker is not able to modify any of the source code, the attacker can use the victim's secret\_key to decrypt the stored encrypted symmetric key for any of the victim's chats, allowing the attacker to then decrypt all the chat's messages. This is assuming the attacker succeeded in carrying out an XSS attack through some client-side input in the app. We believe that our current implementation is reasonably protected against this type of attack, for reasons described in the previous paragraph.

## Side-Channel Attacks

One weakness of in-browser, Javascript cryptography is the lack of control over the underlying process running on the system. This makes it difficult to manage system information which could leak information about the cryptographic processes needed when handling message encryption/decryption or key generation. Given this, we must note that our app relies on a library, the Stanford Javascript Crypto Library, which could potentially be susceptible to side-channel attacks. Most importantly, it could be possible that there exist timing attacks on the library that we may not be aware of.

### 3.1.2 Performance Issues

Security features, by necessity, introduce an overhead on any system. In our case, we try to reach a balance between three main parts of our system: security, performance and usability. In doing so, we make trade-offs on performance. One such trade-off is that of encrypting user data and storing it server-side so that the user can access the site regardless of the device they are using. The trade-off here is that the user needs to load this information each time that they log in. This information, however, can grow to be large depending on the amount of conversations that the user is in. Given this, just loading the site could potentially be slow.

Another performance issue is that of decrypting and encrypting messages. When a chat page is loaded, all of the messages in the chat are decrypted using the Stanford Javascript Crypto Library (SJCL). While the system works well with a couple of messages, if there were hundreds of messages, this may be a difficult and slow task overall.

## 3.2 Future Work

Currently, the main security weakness in Messenr is unencrypted session storage in the client. One way to mitigate risk of an XSS attack is to keep the session storage data encrypted with the user's password, and decrypt the data when necessary by prompting the user for his password. Although this might decrease user-friendliness (and possibly performance), this could greatly reduce the chances of a successful XSS attack described in the previous section .

Performance, as noted before, is a concern when implementing security features. Thus, it's important to consider how we could improve the app with respect to this aspect. One such improvement would be to implement a more efficient storage scheme when storing user data on the server. Moreover, when sending any request between the client and the server, we can verify that the information being sent is just the information we need so that we are not sending extra unnecessary information. For example, when displaying a chat to a user, we can limit the chat to only load 50 messages at a time. This would limit the amount of computation required when displaying a page.

Additionally, one potential improvement to the app is to provide a group chatting feature. This is traditionally difficult due to issues with key-sharing and encryption. However, in our protocol, the actual communication is encrypted using symmetric keys. Moreover, because the symmetric keys are stored on the server and encrypted using the public keys of the parties in a conversation, adding more parties would be as simple as just encrypting the symmetric key with the added party's public key and including it in the database.

## 3.3 Conclusion

In this paper, we presented Messenr, a proof-of-concept app that provides strong end-to-end encryption for chats and allows users to execute server-side searches through their encrypted messages without the server learning the contents of any message nor what the user searched for. This web application builds on the research presented in "Practical Dynamic Searchable Encryption with Small Leakage" [2] as well as in Pmail [7] to bring users the functionality we set out to achieve in a modern instant messaging based web application. We not only hope to tackle the remaining issues and future work discussed in the trailing sections of the paper, but we also hope that the ideas we presented in the design and implementation of Messenr are useful for existing messaging companies such as WhatsApp as well as developers seeking to expand on this open source project.

## 3.4 Acknowledgements

We would like to thank Professor Ronald L. Rivest, our TA Cheng Chen, and the rest of the 6.857 staff for running an incredibly interesting class and for their patience, guidance, and support throughout the entirety of our project.

# Bibliography

- [1] “WhatsApp Encryption Overview”  
<https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>.
- [2] E. Stefanov, C. Papamanthou, E. Shi. “Practical Dynamic Searchable Encryption with Small Leakage”,  
<https://eprint.iacr.org/2013/832.pdf> .
- [3] Stanford Javascript Crypto Library (SJCL)  
<https://crypto.stanford.edu/sjcl/> .
- [4] NIST. “Digital Signature Standard (DSS)”, Appendix D.2.3,  
<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf> .
- [5] Session Storage API.  
<https://developer.mozilla.org/en/docs/Web/API/Window/sessionStorage> .
- [6] D. X. Song, D. Wagner, A. Perrig. “Practical Techniques for Searches on Encrypted Data”,  
<http://www.cs.berkeley.edu/~dawnsong/papers/se.pdf> .
- [7] A. Erb Lugo. PMail  
<https://github.com/tonypr/Pmail> .