Chit Chat

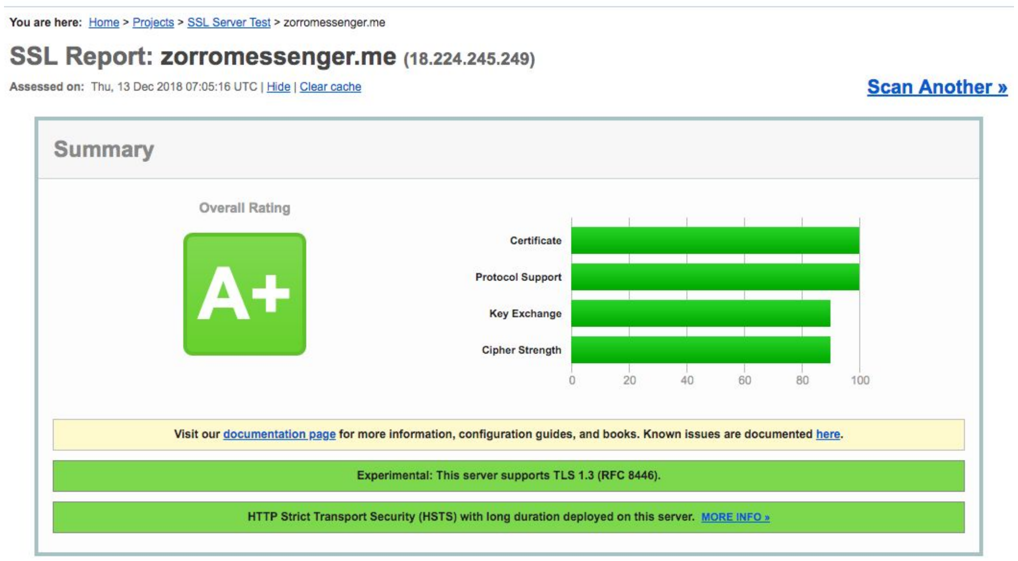
Name: Dayana Rios
ID: 010852359
Team: Honey Potter
Partner: Sogand Kavianpour

Abstract

This paper illustrates the steps and challenges in creating a secure end to end encrypted messaging application for one to one communication. Chit Chat uses the following tools: AES-256, HMAC, SHA-256, RSA, RESTful API, Node server with Express, Json web tokens, MongoDB, Nginx, and AWS. These are the most up to date and secure tools at the time of writing. Chit Chat can be broken into four major components: user interface, encryption/decryption class, server, and database. This paper details how each component interacts to form Chit Chat.

Table of Contents

List of Figures



You are here: Home > Projects > SSL Server Test > zorromessenger.me

SSL Report: zorromessenger.me (18.224.245.249)

Assessed on: Thu, 13 Dec 2018 07:05:16 UTC | Hide | Clear cache

Scan Another »

Summary

Overall Rating: A+

Certificate	100
Protocol Support	100
Key Exchange	90
Cipher Strength	90

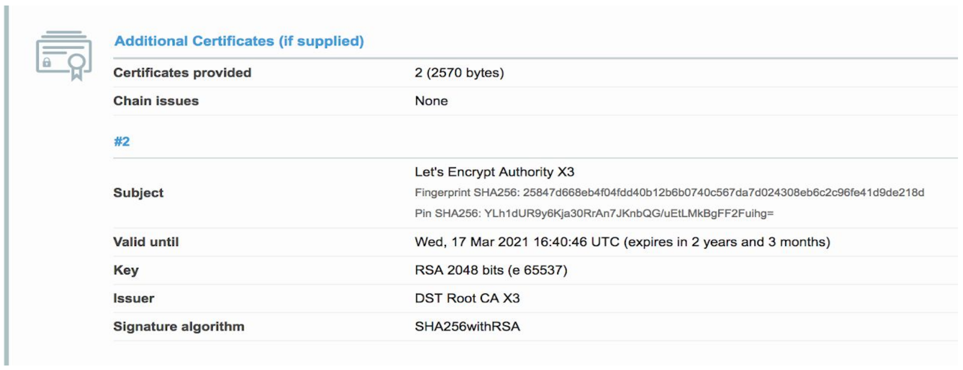
Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

Experimental: This server supports TLS 1.3 (RFC 8446).

HTTP Strict Transport Security (HSTS) with long duration deployed on this server. MORE INFO »

Figure 1[1] - SSL Lab Rating A+

A+ SSL rating for our domain zorromessenger.me



Additional Certificates (if supplied)

Certificates provided	2 (2570 bytes)
Chain issues	None
#2	
Subject	Let's Encrypt Authority X3 Fingerprint SHA256: 25847d668eb4f04fdd40b12b6b0740c567da7d024308eb6c2c96fe41d9de218d Pin SHA256: YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=
Valid until	Wed, 17 Mar 2021 16:40:46 UTC (expires in 2 years and 3 months)
Key	RSA 2048 bits (e 65537)
Issuer	DST Root CA X3
Signature algorithm	SHA256withRSA

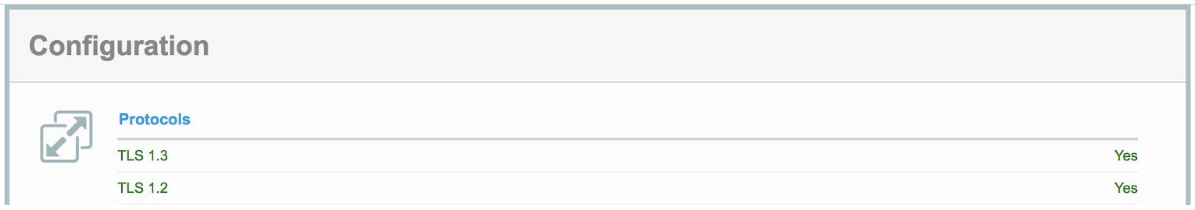
Figure 2[2] - Let's Encrypt Certificate

Let's Encrypt certificate authority (CA) verification, needed to enable HTTPS (SLL/TLS) for our domain.

---

[1] "SSL Report: zorromessenger.me" Qualys, SSL Labs, n.d., https://www.ssllabs.com/ssltest/analyze.html?d=zorromessenger.me (accessed 12 December 2018)

[2] "SSL Report: zorromessenger.me" Qualys, SSL Labs, n.d., https://www.ssllabs.com/ssltest/analyze.html?d=zorromessenger.me (accessed 12 December 2018)

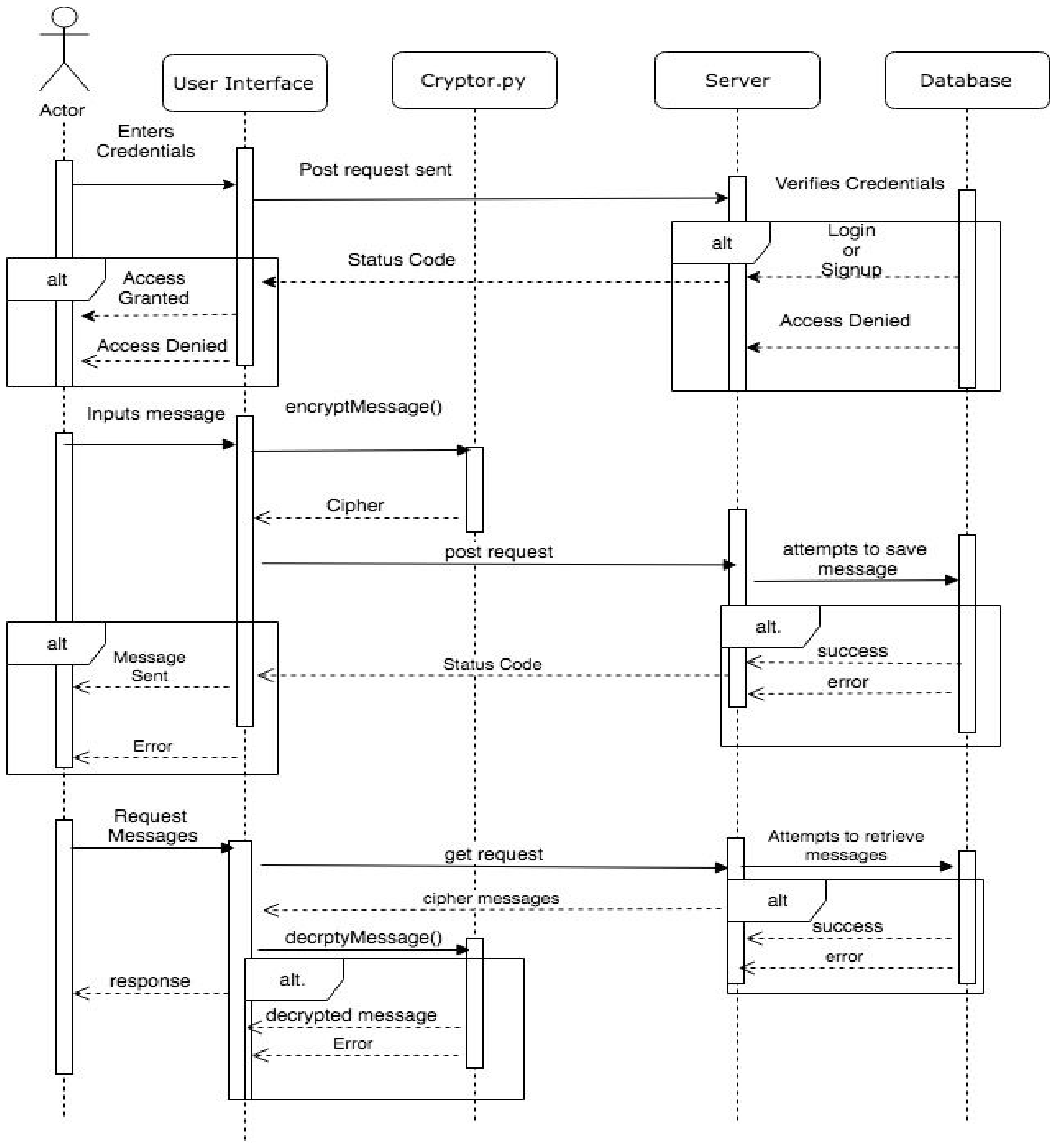

Configuration

Protocols

TLS 1.3	Yes
TLS 1.2	Yes

Figure 3[3] - Configuration Protocols

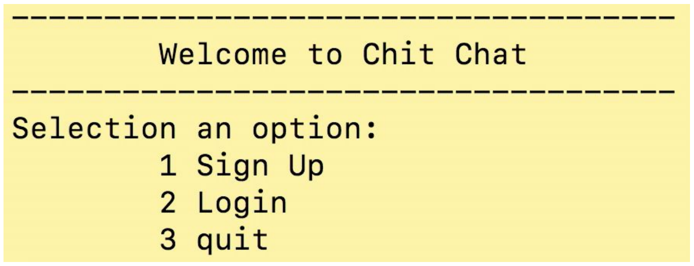
Domain had TLS 1.2 and the recently released 1.3 protocols enabled.



3 "SSL Report: zorromessenger.me" Qualys, SSL Labs, n.d., https://www.ssllabs.com/ssltest/analyze.html?d=zorromessenger.me (accessed 12 December 2018)

Figure 4[4] - UML Sequence Diagram

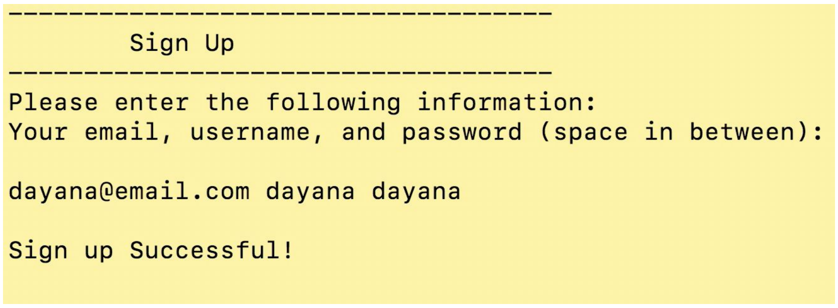
Depicts how the four main system components: user interface, encryption/decryption functions (Cryptor.py), server, and mongoDB interact with one another to form Chit Chat.



```
-----------------------------------
        Welcome to Chit Chat
-----------------------------------
Selection an option:
        1 Sign Up
        2 Login
        3 quit
```

Figure 5 - Launching Chit Chat

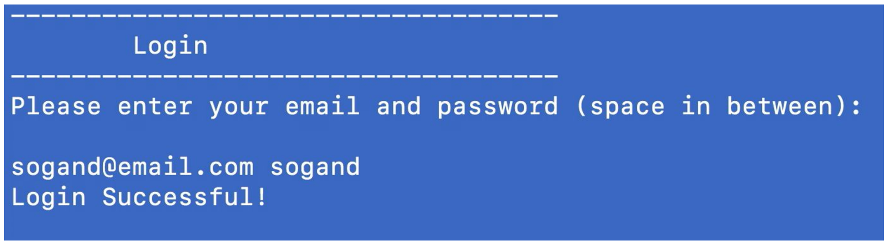
When the application is launched, users are given three options for proceeding.



```
------------------------------------
        Sign Up
------------------------------------
Please enter the following information:
Your email, username, and password (space in between):

dayana@email.com dayana dayana

Sign up Successful!
```

Figure 6 - Chit Chat Sign On

Users are prompted for an email, username, and password to be used as credentials.

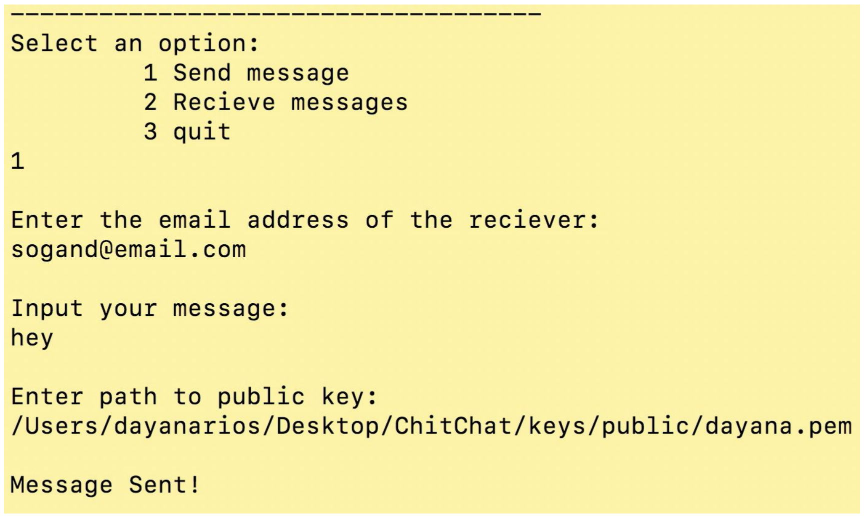


```
------------------------------------
        Login
------------------------------------
Please enter your email and password (space in between):

sogand@email.com sogand
Login Successful!
```

[4] "SSL Report: zorromessenger.me" Qualys, SSL Labs, n.d., https://www.ssllabs.com/ssltest/analyze.html?d=zorromessenger.me (accessed 12 December 2018)

Figure 7 - Logging into Chit Chat

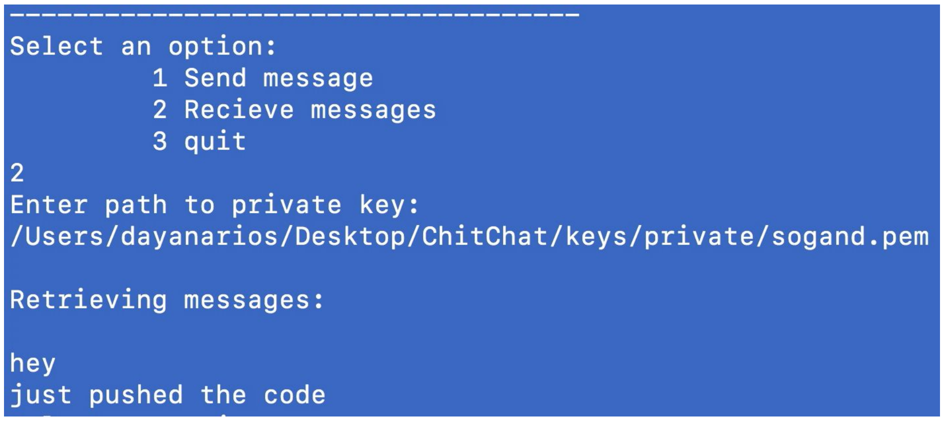
Users must log in with valid email and password credentials. Our application verifies that the credentials are valid.



```
------------------------------------
Select an option:
         1 Send message
         2 Recieve messages
         3 quit
1

Enter the email address of the reciever:
sogand@email.com

Input your message:
hey

Enter path to public key:
/Users/dayanarios/Desktop/ChitChat/keys/public/dayana.pem

Message Sent!
```

Figure 8 - Sending Messages with Chit Chat

Users are given the option to send, receive or quit the program. The figure shows the process of sending a message. Users must input their message and enter the path to the public key. Proper functions are called to encrypt and sign the message. A json object is sent as a post request to our server and stored in our database. A success message is displayed when the message is sent.



```
------------------------------------
Select an option:
         1 Send message
         2 Recieve messages
         3 quit
2
Enter path to private key:
/Users/dayanarios/Desktop/ChitChat/keys/private/sogand.pem

Retrieving messages:

hey
just pushed the code
```

Figure 9 - Receiving Messages on Chit Chat

Users are prompted for the path to their private key and a get request is sent to our server. Json objects are retrieved from our database and appropriate decryption functions are called to decrypt and verify the contents of the json object. Messages are displayed in plaintext to the user.

1. Introduction

This project undertakes the task of developing a secure end to end encrypted messaging application for one to one communication. Such a project is crucial in today's society where users are constantly having their data breached by outsiders while using applications that claim to keep their data secure. Our project strives to ensure the users of our application can rest easy with the knowledge that their messages are securely encrypted and only visible by themself and those intended to receive the messages. Integrity and confidentiality of data is achieved by using the most up to date encryption tools such as AES, HMAC, and RSA. Availability is ensured by using a secure channel of communication. This channel was established by hosting our server on AWS and using Let's Encrypt to obtain a Certificate of Authority. Messages are sent and received as json object by making post and get request to our server. The major difference between our project and others is the form of key exchange. Keys are stored locally on user's machines and exchanged through an external hard drive. We rely heavily on users ability to provide adequate security of keys once they are generated.

WhatsApp is an application that also guarantees secure end to end encryption of messages between users. Chit Chat differs from WhatsApp because the latter uses Extensible Messaging and Presence Protocol (XMPP), an open XML technology for real-time communication[5] between users. This is a very different approach since Chit Chat implemented its own API for sign up, log in and exchanging messages. Additionally, Chit Chat relies heavily on users exchanging keys while WhatsApp handles this issue independently of the user. Despite these shortcomings, Chit Chat provides a secure channel of communication between two users, just like WhatsApp.

2. Literature Search

As stated above, WhatsApp is a popular app used for messaging the guarantees end to end encryption of messages. Extensible Messaging and Presence Protocol (XMPP), an open XML technology for real-time communication[6] between users. XMPP is chosen because it is highly reliable and substantial even under peak traffic[7]. Chit Chat was not tested to see if it could handle a DDOS attack, a vulnerability that WhatsApp easily handles with XMPP. WhatsApp was programmed in Erlang an aigle language that enables quick bug fixes and updates[8]. Mnesia,

[5] "About XMPP", XMPP, n.d. https://xmpp.org/about/ (accessed 12 December 2018)
[6] "About XMPP", XMPP, n.d. https://xmpp.org/about/ (accessed 12 December 2018)
[7] "Build a WhatsApp like Chat App", Contus, 1 February, https://blog.contus.com/how-whatsapp-works-technically-and-how-to-build-an-app-similar-to-it/
[8] "The Technology Behind WhatsApp", Agriya, 19 December 2016, www.agriya.com/blog/technology-behind-whatsapp-developing-instant-messaging-app/

a multi-user distributed database management enables quicker request responses[9], is used for message storage by WhatsApp. It is strikingly more efficient than the mongoDB we used. Overall WhatsApp uses more agile methods to provide the same services as Chit Chat, with the most significant difference being the manner in which Chit Chat distributes keys.

3. Body of Paper

3.1 Problem Statement

The purpose of this project is to provide a secure end to end encrypted messaging application for one to one communication. This type of application will ease communication between individuals that want to safely and confidentiality message others without interference from adversaries. The most up to date and secure tools, AES-256, HMAC, SHA-256, RSA, RESTful API, Node server with Express, Json web tokens, MongoDB, Nginx, and AWS that are used in today's industry are incorporated into our application to provide message integrity and confidentiality and user authentication. Keys are exchanged between users using an external hard drive and stored locally. Chit Chat relies on users to ensure the security of their keys, a major vulnerability that will be further explore in future research.

3.2 Challenges

3.2.1 Individual Challenges

This project was challenging because of the learning curve between the project requirements and the lack of knowledge I had of the concepts. Despite reading about RSA, AES, and HMAC, phase II of the project was very confusing and was implemented without completely understanding the concepts. It wasn't until later in the semester when the concepts were taught, that I was able to connect how each individual component related to the other. Additionally, understanding the individual roles of the AWS instance, Nginx Server, and RESTful server and how they connected with each other was very unclear for the first IV phases of the project. It wasn't until we had to put our client and server code together that it all began to make sense. Setting up our server on AWS and achieving an A+ on SSL Labs with TLS 1.3 enabled and Nginx as a reverse proxy, took four separate attempts to achieve. There is extensive documentation online about these topics, but none provided an explanation specific to our situation. At most we found tutorials that answered questions relative to other servers and operatinging systems but in the same vicinity of problems we were experiencing.

[9] "The Technology Behind WhatsApp", Agriya, 19 December 2016, www.agriya.com/blog/technology-behind-whatsapp-developing-instant-messaging-app/

Despite these shortcomings, this project improved my intuition and allowed me to critically analyze a foreign situation and find solutions to challenging problems.

3.2.2 Group Challenges

Our group challenges parallel my individual challenges. Both Sogand and I had difficulty understanding the concepts of the project since it was both of our first time taking a course on computer security. Our lack of concept understanding made the first three phases of the project difficult to implement. It wasn't until the last phase that the concepts began to make since since they were taught in lecture and we had to find a way to connect all individual components together. Lack of understanding of concepts also meant there was a lot of frustration between us since at every phase there was an issue with our project that prevented us from progressing to the next requirements of the phase.

3.3 Solutions

3.3.1 Design Overview

Figure 5 - Sequence Diagram gives a high level design overview of the four main components of our project and how they function with one another. These four components can further be split into server and client functionality.

For our server design, we have an Ubuntu AWS instance using Nginx as a reverse proxy. We've set up a RESTful node server for our messaging API, which handles all the functionality of our application except for decrypting and encrypting messages, this includes registering new users, signing into the application, storing and receiving messages.

Our client design contains the most up to date and secure functionality for encrypting, decrypting, signing and verifying messages. Additionally our user interface links our client and server functionality together to provide a secure end to end encrypted chat application for one to one communication. Further details of implementation can be found in section 4.

3.3.2 Analysis

Choosing to use a Ubuntu AWS instance with Nginx, meant that little attention had to be paid to the security of our server. We relied heavily on AWS and Nginx's default security protocols to protect our server as well. This provides a single point of failure for our server if either AWS or Nginx are attacked.

Our user interface is not user friendly and relies heavily on user input. Our application instantly sends and receives messages, but depending on the user's typing speed and familiarity with the application, it can take users up to 30 seconds to send a single message because users are prompted for multiple inputs. This makes the application seem complex to the normal user.

We’ve ensured to use the most up to date and secure encryption, decryption, signing and user authentication tools available at the time of publication. These include AES-256, RSA, HMAC, SHA-256, and json web tokens. Our choice of encryption and signing is proven to preserve message integrity and confidentiality from adversaries at the time of publication, but can easily be breached with the use of quantum computers. Json web tokens are the most up to date tool for user authenticity and accountability.

3.4 Individual Work

3.4.1 Phase I and II

Since we were unfamiliar with the concepts required for these phases, we deployed our AWS instance, obtained a domain name, set up Nginx as a reverse proxy, and obtained our Let’s Encrypt Certificate Authority together. Additionally, these steps were required to be done only once, thus working on them together made the most sense.

3.4.2 Phase III

I wrote all the functions relating to encryption in Cryptor.py for phase III. This includes taking a message in plaintext and an RSA public key for the purposes of encryption. The message is passed as an argument to the encryptAES() function which creates a 256 bit key and a 128 bit IV using a secure PRNG. The message is concatenated with the IV and padded appropriately. The final padded result is encrypted using the CBC mode of encryption and created IV and key. The ciphertext is then sent to the HMACgenerator() function which signs the message using HMAC with SHA-256 as its hashing function. This generates a HMAC key and tag. The HMAC key is concatenated with the AES key and encrypted using the RSA public key obtained at the beginning of the program. The message ciphertext, rsa ciphertext, and hmac tag are then stored in a json object. This json object is what is sent to our server and stored in our database.

3.4.3 Phase IV

I set up the mongoDB that was to be used for storing and retrieving messages. I set up a RESTful API with Node.js and Express Framework to communicate with our database and client application. Additionally I provided the “skeleton” for the rest of the API, this includes setting up appropriate routes and controllers. I implemented the initial set up for sign up post requests and retrieving all users in the database (for testing purposes).

3.4.4 Phase Client

For phases I-IV our application had a GUI, but for this final phase we decided to go with a command line interface. I implemented the general flow of the new interface which first allows the user to have 3 options when starting the application: 1. Login, 2. Sign Up, and 3. quit. Both Login and Sign up redirect users to the messaging portion of the application.

To send a message our client code calls the function messaging() which is split into sending and receiving messages. When sending a message, users are prompted for the receiver's email, we chose to identify users by email since our API ensures that emails are unique relative to the users that are stored in the database. Users then input their message and are prompted for a public key. Our application takes the message and path to public key and calls proper the proper encryption functions from phase III. The resulting json object is used to make a post request to our server, which then stores the json object in our mongoDB.

The same messaging() function has an option for receiving messages. In this section users are prompted for the path to their private key and our application makes a get request to our server which retrieves all json objects pertaining to the user. The json object is spent to our decryptMessage() function of our Cryptor class which extracts the individual components of the json file and call appropriate decryption methods. Our Cryptor class returns the plaintext version of the message and outputs it to the user.

3.5 Work Comparison

Work was fairly divided between myself and my partner with equal participation in all phases of the project. We both strived to implement our application with the most secure and up to date tools. On my end, a could have implemented a better manner of key exchange since our current implementation leaves us vulnerable to attacks. Another issue is our user interface is not user friendly and relies heavily on the user knowing how to properly use RSA keys. On my partners end, more functionality could have been included in our API to enable users to manually delete messages. When making a get request for messages, my partner didn't implement a way of displaying who the message was coming from. Additionally, this project was frustrating for us both and caused tension because we didn't fully understand the concepts and had difficulty meeting all the expected requirements on time for deadlines.

3.6 Shortcomings

The biggest shortcoming of our project was our method used to exchange keys between users. We rely heavily on user's ability to provide adequate security for their keys. Keys are exchanged via external hard drive, meaning users must be in close proximity to each other. Keys can be kept on the hard drive or kept locally on the machine. We choose this method since exchanging keys over the web enables them to be vulnerable to web based attacks and attacks mentioned previously. Additionally our project is subject to Man in the middle attack.

3.7 Lessons Learned

3.7.1 Individual

Despite having a steep learning curve for this project, by the final phase of the project it became clear how the different components involved functioned with one another. Implementing this project provided a deeper understanding of the concepts presented in lecture. Additionally, it gave me a strong understanding of how popular end to end encryption applications work.

3.7.2 Group

Lessons learned as a group parallel individual lessons learned. We both had limited knowledge of the concepts required for this project, but by the final phase and after concepts were learned in lecture the project as a whole made more sense.

4. Implementation and Results

Refer to Figure 5 - UML Sequence Diagram for a highlever overview of the main components of the project. This section is taken directly from the Chit Chat - Design Documentation[10], written by myself and Sogand, with modifications and added explanation of individual components. It should be noted that little error handling was implementing. Our application only alerts users of an error if the user's credentials are invalid or already in use by another user (email and username) and errors with get and post request. We assume users have their own pair of RSA keys, know how to properly use them and can supply a valid key path to our application.

4.1 Phase I

We deployed an Ubuntu Amazon Web Services instance and linked a free DNS registered on Namecheap (https://zorromessenger.me) to our server. Next, we set up Nginx on our Ubuntu instance. It should be noted that our AWS server is no longer running, thus our website is no longer accessible.

4.2 Phase II

We imported a config file enabling our Nginx to act as a reverse proxy. We used Certbot and Let's Encrypt to obtain a Certificate Authority[11] to enable our site to accept HTTPS request. At this stage we tested our domain name on SSL Labs and obtained a rating of A+ and enabled TLS 1.3.

[10] Kavianpour, Sogand. Rios, Dayana. "Chit Chat - Design Documentation", 12 December 2018
[11] "Certbot", n.d. https://certbot.eff.org/ (accessed 12 December 2018)

4.3 Phase III

We generated 2048-bit RSA public/private key pairs[12] through the command line, necessary for encryption purposes. At this stage we created a python program to handle the encryption, decryption, signing, and verification of messages.

4.3.1 Encryption Functions

encryptMessage():

This function is in charge of calling all subfunctions of encryption. It calls the functions in the following order: encryptAES(), HMACgenerator(), and encryptRSA(). The AES and HMAC keys are concatenated and passed as an argument to encryptRSA(), which in turn returns ciphertext. A json object containing the AES ciphertext, RSA ciphertext, and the HMAC tag is returned.

encryptAES():

The message is passed as an argument which creates a 256 bit key and a 128 bit IV using a secure PRNG. The message is concatenated with the IV and padded appropriately and encrypted with AES[13]. The resulting ciphertext is returned.

HMACgenerator():

The ciphertext is then sent to this function which signs the message using HMAC with SHA-256 as its hashing function. The resulting tag and generate HMAC key are returneed.

encryptRSA():

Takes the concatenation of the AES key and HMAC key and encrypts it using a RSA[14] public key. Ciphertext is returned.

4.3.2 Decryption Functions

decryptMessage():

[12] "OpenSSL: Generating an RSA Key From the Command Line" Rietta / Security, 17 January 2018, https://rietta.com/blog/2012/01/27/openssl-generating-rsa-key-from-command (accessed 12 December 2018)
[13] "Using AES for Encryption and Decryption", Novixys Software Dev Blog, 8 February 2018, https://www.novixys.com/blog/using-aes-encryption-decryption-python-pycrypto/
[14] "Black Hat Python — Encrypt and Decrypt with RSA Cryptography", Medium, 21 December 2017, https://medium.com/@ismailakkila/black-hat-python-encrypt-and-decrypt-with-rsa-cryptogaphy-bd6df84d65bc (accessed 12 December 2018)

Takes the JSON file from encryption, which contains the AES cipher text, RSA cipher text, and HMAC tag, and feeds it to the RSA decryption method. We then authenticate the HMAC tag to make sure it matches with the RSA resulting tag value. Once this is verified the AES cipher text is then decrypted and shows the original message

decryptAES():

Takes in the AES key, AES cipher text, and an IV to output an AES object

decryptRSA():

Takes in the user's private key and RSA cipher text and and outputs a decrypted RSA object

4.4 Phase IV

Here we deployed a node server with Express Framework and set up a MongoDB database. We also created a RESTful API for our application. Our server consisted of routes, users, messages, and server JavaScript files. POSTMAN played a large role in testing our POST and GET requests. API routes included: signup, login, send message, receive message, and show all users (for testing purposes). The sign up route is used for creating new users in the database (email, username, password are the required credentials). The login route is used for verifying a user is registered and thus their credentials are in the database. For added security, we integrated json web tokens[15] in our routes for authentication. Routes were also created for posting and getting messages.

4.5 Phase Client

We created a separate user interface that integrated the API with our encryption/decryption class. The new user interface is split into four sections a main which controls the flow of the program, a sign up function, a login function and a messaging function. Our sign up function registers a new user for our application and stores their credentials simultaneously validating whether the given credentials are unique. Our login function validates the credentials supplied with the user to ensure that the user logging in is a valid user. Please refer to section 3.4.4 Phase Client for message implementation details.

4.6 Results

The Phase Client concluded our project and resulted in a secure end to end encrypted messaging application for one to one communication. Our application preserves confidentiality of messages by encrypting them using AES-256 and signing them with HMAC. AES and

[15] "Introduction to JSON Web Token" JWT, n.d. https://jwt.io/introduction/ (accessed 12 December 2018)

HMAC keys are encrypted using RSA keys. The most up to date and secure encryption tools were used at the time of writing. Message integrity is preserved by ensuring the HMAC tag was not tampered with. HMAC uses SHA-256 as its hash function and a 128 bit key. Messages can only be sent and received from registered users. Our application ensures that users are who they claim to by validating their credentials during log in by issuing Json web tokens. All usernames and emails must be unique to each user. Figures 5-9 display snapshots of our running application.

5. Conclusion and Further Work

This project shows how we can use the most up to date and secure tools available at the time of writing to make a secure end to end encrypted messaging application for one to one communication. Some of the most prevalent tools used are the following: AES-256, HMAC, SHA-256, RSA, RESTful API, Json web token, MongoDB, Nginx, and AWS.

If more time was allotted we would make the following improvements, we would implement a different method for key exchange. Our current implementation leaves users vulnerable to several attacks. We would also like to implement a GUI that only request users to login or sign up, select who they would like to message, send messages, and receive messages. I would like to revisit this project in the future and make these improvements.

6. References

"SSL Report: zorromessenger.me" *Qualys, SSL Labs,* 12 December 2018, https://www.ssllabs.com/ssltest/analyze.html?d=zorromessenger.me

"About XMPP", *XMPP,* n.d. https://xmpp.org/about/

"Build a WhatsApp like Chat App", *Contus,* 1 February, https://blog.contus.com/how-whatsapp-works-technically-and-how-to-build-an-app-similar-to-it/

"The Technology Behind WhatsApp", *Agriya,* 19 December 2016, www.agriya.com/blog/technology-behind-whatsapp-developing-instant-messaging-app/

Kavianpour, Sogand. Rios, Dayana. "Chit Chat - Design Documentation", 12 December 2018

"OpenSSL: Generating an RSA Key From the Command Line" *Rietta / Security*, 27 January 2012, https://rietta.com/blog/2012/01/27/openssl-generating-rsa-key-from-command/

"Certbot", 12 December 2018, https://certbot.eff.org/

"Using AES for Encryption and Decryption", *Novixys Software Dev Blog,* 8 February 2018, https://www.novixys.com/blog/using-aes-encryption-decryption-python-pycrypto/

"Black Hat Python — Encrypt and Decrypt with RSA Cryptography", *Medium*, 21 December 2017, https://medium.com/@ismailakkila/black-hat-python-encrypt-and-decrypt-with-rsa-cryptogaphy-bd6df84d65bc

"Introduction to JSON Web Token" *JWT,* n.d. https://jwt.io/introduction/