CS 4389 Data and Applications Security
Final Project Deliverable 2

Encrypt-o-File

Robbie Castillo
Wesley Bales
Nooreen Ahmad
Zain Husain
Aaron Stone
Brad Mickow

Section 0: Description

- **Robbie Castillo, Nooreen Ahmad** **GUI Team**

Robbie Castillo:

I worked alongside Nooreen to implement the frontend application. I was co-responsible for the selection of the messaging UI template and the implementation of networking protocols. Also, I helped with refactoring of the messenger UI's React code to follow the standard of having classes, or "components" (as opposed to just having big functions). Lastly, I helped to integrate the backend's encryption/decryption scheme to the frontend code.

Nooreen Ahmad:

As a part of the frontend team, I primarily assisted Robbie in the implementation of our React-based interface. We worked together to create a functional messaging interface similar to popular messaging apps to showcase the symmetric encryption of a sent message. Robbie and I spent a majority of our time familiarizing ourselves with React and learning how to add the appropriate functionalities to our interface in order to properly address the task at hand.

- Wesley Bales, Brad Mickow, Zain Hussain, Aaron Stone **Backend Team**

Wesley Bales:

I worked on the symmetric encryption/decryption portion of the code. I had originally planned on replicating one of the common symmetric encryption algorithms like DES or AES, but I quickly realized that I wasn't going to be able to. The scope of the industry standard algorithms was far beyond what I was going to be able to replicate. I began looking at alternatives and settled on a simple symmetric algorithm that would encrypt/decrypt with some shared key. From here I decided on a block cipher for each 16 characters instead of a stream cipher.

I tried a couple different algorithms for encryption, but for most of them I was unable to decrypt the message due to congruent numbers and modulus. I ended up moving away from those algorithms, but difficulty decrypting is the point of encryption anyway. I implemented a simple use-case of my code to be used as an example for what it did, and how it worked, and we were able to submit the initial version with our first deliverable. After I had a stable product, I was able to work with Zain to implement the hashing that he wrote. We worked out some integration bugs, and once that was ready, we integrated with the frontend team's messaging app to send encrypted messages from client to client, with each message passing through the server. Each message is encrypted while passing through the network, and each time it arrives at either the server or a client the message is decrypted, and the hash verified to ensure the message hasn't changed.

Brad Mickow:

I was tasked with implementing asymmetric encryption. My goal started with writing my own advanced asymmetric encryption algorithm from scratch, but I quickly learned that it was much more challenging than I expected. I soon moved to RSA encryption and once again ran into issues with getting functional algorithms to work as public/private keys. After discussing the issues with the team, we concluded that my time would be best used contributing to this document and the presentation. I now realize that I should have started with a very simple version of asymmetric encryption and built on that, as opposed to starting with the most complicated implementation and working backwards. I will utilize this strategy in the future.

Zain Husain:
I was in charge of implementing a hashing algorithm. We decided to encrypt the hash along with the message and this placed some restrictions on the output. The result needed to be ascii characters with a total length that was a factor of 16. This would allow Wes' encryption algorithm to handle the hash without issue. I first looked into the various secure hashing algorithms to see if one would be a good fit for our project. I began researching implementation for SHA-256. I believed that this would be the ideal solution as it was very secure. The length of the output was not ideal so I began to look at other options. MD5 was the next best algorithm that I found. It had an output of 16 characters and was reasonably secure. I began to worry that I would not be able to implement it without copying the source code directly. I decided to create my own version of a hashing algorithm used on strings. This allowed me to control the output to a greater extent than other options.

Once written we had to integrate into Wes' encryption algorithm. One of the issues that we kept running into was in dealing with the large number being generated by the algorithm. The output would quickly grow to 20 or 30 digits long. The number by itself wasn't the problem, just that it was being converted to scientific notation. We were getting output like: 2.53528824267525e33. We decided to alter the algorithm so that it grew at a slower rate, staying under 16 digits and avoiding being changed into scientific notation.

Aaron Stone:
Initially I was in a support role, but then the group decided to include a Diffie-Hellman Key exchange. Which is a basic key exchange that keeps previously agreed on private key values from being transmitted over the internet. Given the proper implementation one of the parties would sign with a sender's private and receiver's public key before sending the shared key value over the network. The receiver would then receive the message and decipher it the receiver's own private key and the sender's public key value. Authentication of the sender is verified if the end result on both sides are equal to each other. In effect, this means the sender's private and the receiver's keys, and the receiver's private and the sender's public key values should render the same value. JavaScript's learning curve had a direct impact on the ability to complete this portion of the project. The diffie-hellman key exchange program, key.js, encompassed a hard coded coprime algorithm for producing a secret and public key for a sender and receiver. During the process of creating a random number for these algorithms and learning javascript on the go; I was unable to complete a function that would produce a two random numbers. However the key.js file was able to create two co-primes with hard coded values, convert them into a shared key, a public key for Alice and Bob, and private key for Alice and Bob. Key.js was not compiling, but it was error free. So, the code block to compare the two values and a push to integrate this with main was never made. The plan was to initially use this key exchange for symmetric and asymmetric encryption for added security in encryption methods. As a support role I aided in the creation of both deliverables, comparisons of similar work, and the presentation.

Section 1: Introduction

We initially chose this project because, as a group, we were interested in pursuing something related to encryption. We had some varying skill sets with frontend and backend experience among the group members and wanted something for everyone. Choosing something that has tasks for seven people isn't an easy goal, but we picked one out that was interesting and allowed for us to explore different parts of encryption. Our original plan was to encrypt and

decrypt different files, but we pivoted before the first deliverable and began to transition to a message application with encrypted messages.

We didn't have the expectation of creating a ground-breaking messaging application. It's not a revolutionary project, and encryption has been around for awhile. Our goal was to create something that would be an educational experience for those involved. We envisioned an opportunity to learn the specifics of some encryption algorithms, work on a larger than usual team, and integrate distinct frontend and backend code into a single working product.

Unfortunately, this means our project doesn't really fill any gaps in security applications. Messaging applications are a common assignment in academia right now, but most are done as command line applications. These applications use mainstream academia languages, such as C/C++ or Java. In our implementation, we were unique by using two frameworks of JavaScript, React.JS, and Node.JS to implement the frontend and backend systems, respectively.

Section 2: Background and Related Work

There are a number of similar messaging applications that encrypt a message with symmetric encryption. For example, Kissapp is a messaging app currently implemented on iOS devices that is similar in many ways to our Encrypt-o-File application. [1] In its current implementation, Kissapp uses AES 256 key symmetric encryption to generate keys for creating signatures with private keys. Public keys in Kissapp are created as RSA 2048 keys and are sent via SSL for more an added layer of security encryption over the network. Another similarity is that Kissapp uses a hash function for encrypting your phone number. However Kissapp is incredibly sophisticated as it is hashed with SHA256 encryption, and stores your phone number in the keychain, the most protected part of your phone that even Apple can not access. [1] This hashed value is also stored on the server. Your private key is kept on your device, protected by a personal private code that should be memorized in order to keep access to your private key. Given the resources available Kissapp is also able to maintain servers that encrypt messages back to you with an AES 256 key and your public key.
Apple CryptoKit is another example of a popular application that implements cryptographic operations. This application uses Secure Enclave to store 256-bit elliptic curve private keys in the keychain, symmetric encryption for message authentication codes, SHA512, SHA384, and SHA256 hashing algorithms. [2] Unlike our application the CryptoKit also implements an AES (Advanced Encryption Standard) for the implementation of ChaCha20-Poly1305 cipher, and Public-Key Cryptography.

Three other popular messaging applications that have a similar design to our initial idea of Encrypt-o-File include Messenger, Riot, and Signal. [3] These applications are all very similar in that they implement Diffie-Hellman, symmetric, and hashing to encrypt messages. In these applications Curve25519 is implemented as the Diffie-Hellman key exchange, AES-256 is the specific symmetric encryption, and HMAC-SHA256 is the hashing algorithm. Other common themes to our application, Encrypt-o-File, include not allowing a secondary factor of authentication, absence of logging timestamps and user IP addresses, and only Signal actually encrypts metadata. [3]

Section 3: Implementation and Results

3.1: Implementation

Frontend

The frontend is implemented using an open source React JS template that is specifically designed for messenger environments. Therefore, we used that template as the basis for our frontend application. From there, we created a dataflow for the Compose component, which is where the user types and submits their message. On message submission, the Compose component takes whatever was typed, and passes the message “up” to the MessageList component. The user’s message is then encrypted and sent over the network to the backend server. There, the message is processed for decryption and storage.

When a message is received from the backend server, it is first decrypted, and then hashes are then compared to ensure that the message wasn’t modified in transit. If the message integrity remains, then that message is displayed by adding the new message to the existing messages list on the client side.

Backend

The symmetric encryption, decryption, and hashing are all written in Node.js. We began by designing a simple scenario that represented the interaction, then expanded on the individual parts of the system until it was sufficiently implemented out. When a new message is sent, the message is first separated into blocks. If there aren’t enough characters for the last block, the message is padded for consistency of length. Then the message is hashed to create an additional hash-block at the end to provide an integrity check as the message is passed. After being hashed, the message is encrypted with our simple algorithm, and then stored in a data structure. From here, the newly encrypted message is sent out every time the frontend accesses the endpoint our backend creates. The frontend then receives the encrypted message and it is handled client side.

Organization

We used GitHub to maintain strict control over our working code and what was in development. Each person worked from their own branches and we merged code as it become ready for integration. Some of us have worked on projects before where people would force push to master and overwrite code. By implementing strict version control over the project we were able to keep the disparate portions of the project protected until ready.

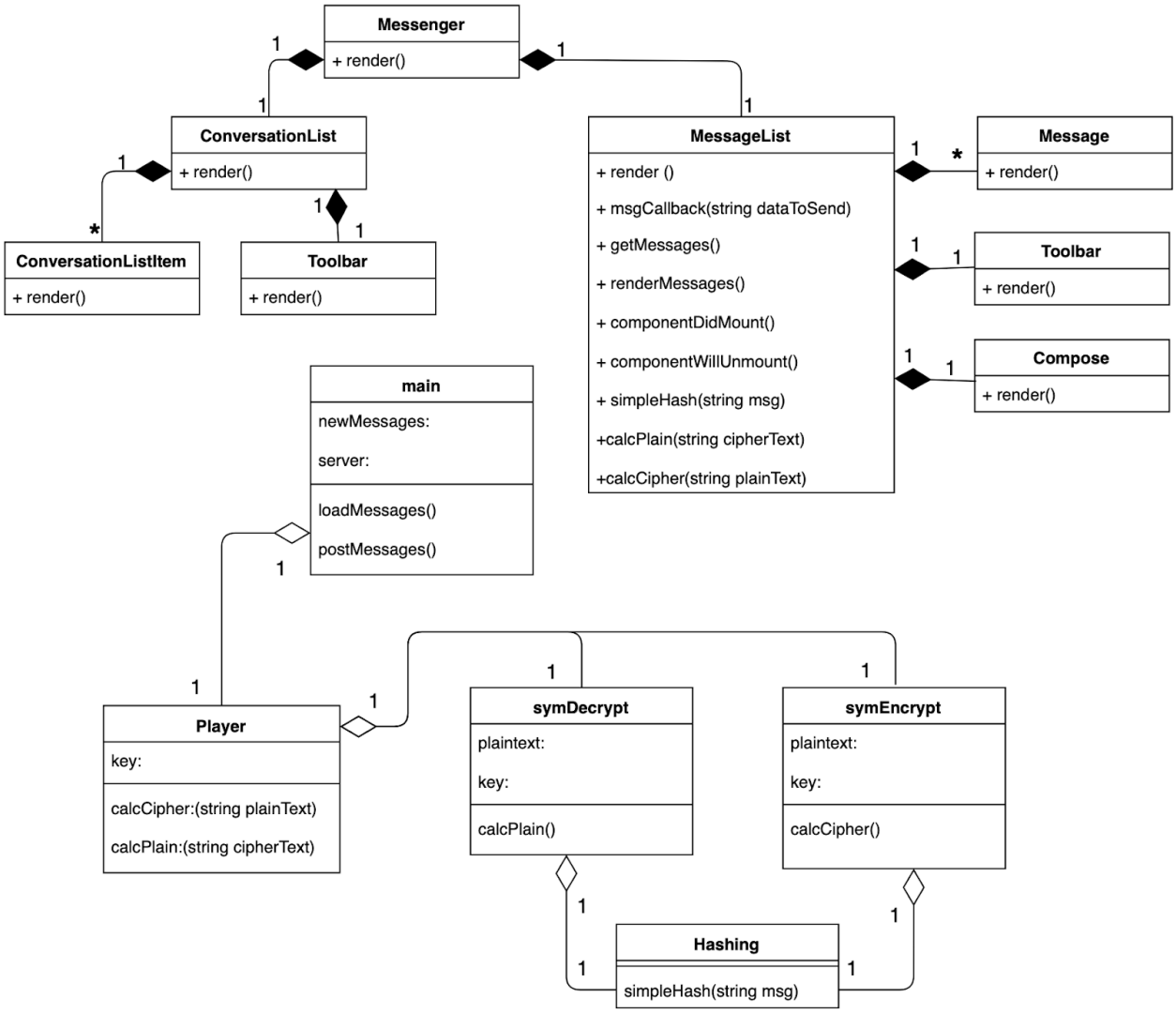
3.2: Security Principles Addressed

Our project supports the confidentiality, integrity, and availability security principles. Each message is encrypted in transit keeping the contents confidential from anyone who doesn’t have the key. Each message is hashed prior to encryption and the hash is verified at decryption, supporting integrity. The data is accessible to the users via the client where they can send and receive messages any time they want.

3.3: Results

The project resulted in a fully functional messaging app that includes symmetric encryption and the capability of displaying plain-text or encrypted version of the data. We were unsuccessful in asymmetric encryption and the Diffie-Hellman key exchange implementation but succeeded in our main goal of creating a messaging app with encryption.

When comparing our encryption. hashing and decryption methods to industry implementations, one quickly realizes that ours is a simplified version of them. We did attempt to replicate industry standards such as AES, DES, RSA, DH, but we were not successful. This leads us into whether or not we met our initial expectations for the system. Although these professional implementations were too advanced for us to replicate from scratch, we were able to learn a great deal from completing a simplified version. This simplified version works and meets our expectations set out at the beginning of the project.



This is the UML class diagram for our entire system, comprised of two different applications.

Section 4: Conclusion and Future Work

Our implementation results were a success. We successfully implemented a basic messaging app that could protect confidentiality and integrity. We were able to work as a team to create two separate systems that network together to create one cohesive and secure experience. Our application adds value by using only React and Node JS to create this secure messenger app. Most web-based messenger apps use a different backend programming language, as opposed to using Node JS. This project could be extended further by using open-source cryptography libraries to enhance the confidentiality and integrity aspects of the application. Features such as a more complex and robust encryption/decryption library, along with a complex hashing library, would be a tremendous boost to the overall security of the app.

Section 5: References

[1] *Kisapp, world's first smart ephemeral and encrypted instant messenger, available on iPhone and iPad.* [Online]. Available: https://www.kisapp.co/en/blog.html. [Accessed: 15-Nov-2019].

[2] “Apple CryptoKit,” *Apple CryptoKit* | *Apple Developer Documentation*. [Online]. Available: https://developer.apple.com/documentation/cryptokit. [Accessed: 15-Nov-2019].

[3] *Postfix TLS Support*. [Online]. Available: http://ftp.uma.es/mirror/postfix/doc/TLS_README.html. [Accessed: 16-Nov-2019].