Design Phase Documentation: Chat Application

Corey Rau & Anthony Rojas

CECS 478

May 4, 2017

TABLE OF CONTENTS

Design Phase

1 Overview

1.1 Introduction

The application being designed is a chat application that will allow two users to send messages to each other. Each message sent in the conversations will utilize end-to-end encryption to protect the privacy and data integrity of each user. The purpose of this document is to give a detailed overview of how such security methods will work, how they will be implemented, the attack surfaces from which an adversary may attempt breaching our security, and what measures would theoretically have to be taken given a successful attack.

1.2 Implementation Overview

This implantation of this application is completed using JavaScript for the backend, which means the server side actions, such as the API's, and security methods are done with JavaScript. The front end of the application will be using Java for such things as the GUI and app deployment onto desktop machines. This application will be a desktop application.

2 Security Methods

2.1 Message Encryption

Passing the message via plaintext without encrypting will always make it susceptible to eavesdropping or tampering of the contents with relative ease. To preserve the users' privacy, every message sent will be encrypted using RSA encryption, AES encryption, and an HMAC tag for message integrity (which will be discussed later on). The first step to encryption is getting the input, which will be the message and the file path to the RSA public key. Once the input has been received, the process to encrypting the message can begin and will end with a JSON format object being returned. First, an AES object will be created and generate an AES key. Once that is completed, the system will have an IV added to it by using an AES key to generate said IV. The message will then be encrypted using the AES object that is prepended with the IV, meaning the message is run through AES encryption and AES ciphertext is generated. An RSA object will then be created to load the public key into it, along with the AES and HMAC keys. OAEP padding will be used in the RSA object when concatenating the 3 attributes that make up the object and RSA ciphertext will be output. An HMAC tag is created using the HMAC key and generated with SHA-256. After the AES ciphertext, RSA ciphertext, and HMAC tag have been generated, they are packaged in a JSON object, ready for use in the decryption method.

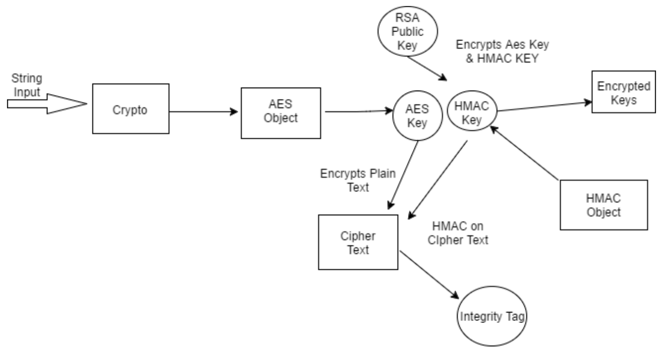

Figure 1: Message Encryption

2.2 MESSAGE DECRYPTION

To decrypt the cipher text, first an RSA object with OAEP padding and a key size of 2048 bits is created. The private key is loaded into the RSA object. Then the RSA ciphertext is decrypted and a pair of 256-bit keys, an AES key with CBC and an HMAC key, are recovered. HMAC with SHA-256 is ran on the recovered HMAC key to re-generate the HMAC tag. The tag is then compared with the input tag from the JSON object. If the tags do not match then failure is returned. If the tags do match, then the AES ciphertext is decrypted and the plain text is returned.

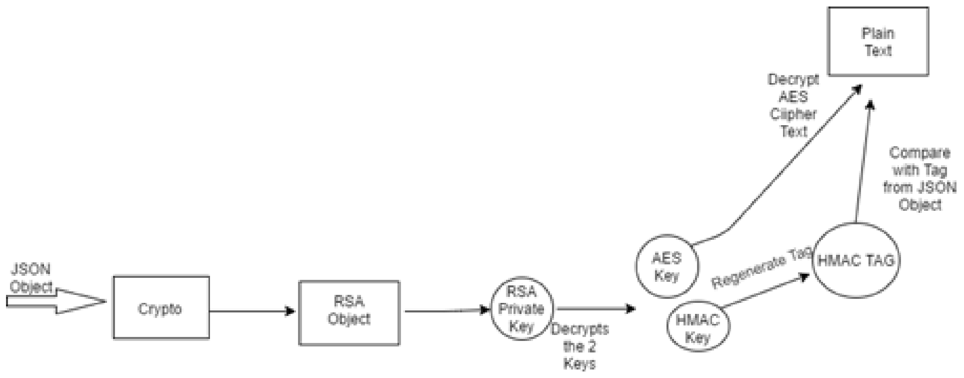

Figure 2: Decryption

2.3 MESSAGE INTEGRITY CHECKS

The message integrity checks being used for this application will involve the validation of HMAC tags. What this means, is that every message sent will have an HMAC assigned to it. The HMAC tag is created at the time of encryption where a 256-bit HMAC key is used with the RSA ciphertext of the message to generate a tag. As stated in section II-B, the key used for the HMAC tag generation will be stored within the RSA ciphertext next to the IV and AES key. This HMAC key will be crucial for the receiver of the message.

Once the message has been sent, the receiver will then extract the key from the RSA ciphertext, as stated previously in section II-C, and will run a HMAC generator to create an HMAC tag from that HMAC key. Once that is completed, the system will pull the HMAC tag from the JSON object passed in from the encryption method. This HMAC tag from the JSON will then be compared to the HMAC tag generated in the decryption function. If the two tags match, then the system will confirm that the message has not been tampered with, and message integrity has in fact been retained. This message integrity check will be performed on every message in the application.

2.4 Authentication Methods (JWT)

JWT or JSON Web Tokens are used for authentication. When the user is logged in, API requests include the JWT to allow user authentication for routes, services, and resources. The JWT consists of a header, payload, and signature. The header contains the token and a hashing algorithm, HMAC SHA256. The payload contains the claims, which are statement about an entity. The signature is the last part, which contains the encoded header, encoded payload, a secret, the HMAC algorithm, and sign that. JWT is a stateless authentication mechanism, is compact, and is very secure.

2.5 Key Exchange

Key exchange is a crucial part of this project because if keys are compromised and obtained by an adversary, then a majority of the system will be compromised, specifically the privacy and integrity of the users. The public key must be exchanged in order to allow for the sender to be able to encrypt a message in a form that can be decrypted by the receiver. To simplify it, let's create a scenario where user A is trying to send message to user B. In order for this to occur, user A must first obtain the public key of user B in order to encrypt the message in a way that can be decrypted by user B when he receives the message. Without this key exchange, user B will not be able to understand what user A is saying because the system will not be able to accept the message due to the HMAC key in the RSA ciphertext being wrong, which will flag it as a failure in message authentication.

3 Implementation

As stated previously, the code for the backend side of this application is implemented using Node.js JavaScript.

3.1 Message Encryption

The encryption method will be included in the same file as the decryption method for simplicity's sake and to use up less memory and resources when running the application. Due to Node.js syntax and common practice, the module here will export the output from the encryption method. To even use this method, the crypto and file reader libraries have to be included into the program. An IV, AES key, and HMAC key will be initialized using the randomBytes function to have randomness when encrypting the message. The public key is read using readFileSync() in utf8 encoding. The IV is then crated using createCipherIv() where the encryption algorithm is specified as AES in CBC mode. Then, the message is encrypted using AES encryption via the update() function to concatenate the ciphertext with the HMAC array. The HMAC tag is then generated using createhmac() and digest() with SHA 256. Finally, the RSA ciphertext concatenates the keys into itself with OAEP padding with the publicEncrypt() function. The return value for this function is a JSON object with RSA ciphertext, AES ciphertext, and an HMAC tag.

3.2 Message Decryption

As explained earlier, for simplicity's sake the decryption method has been included in the same file as the encryption method. The input parameters for this are a JSON object and a key path to the private key of the receiver; the key path is entered is entered as a string. The first thing to do is unzip the JSON object by creating variables and initializing them with the value of each element of the JSON object. This means the RSA cipher text, AES cipher text, and HMAC tag are all stored in variables. The private key is then read using the readFileSync() method to read it in synchronous mode. A buffer then reads the content of the RSA cipher text in hex encoding. The privateDecrypt() method is used to decrypt the RSA cipher text using the buffer that was initialized before. This results in a plaintext that has an IV key, AES key, and HMAC key. Due to the way the keys were concatenated in the emcryption method, the plaintext must be sliced using the slice() function as follows: the IV is sliced from 0 to 16, the AES key is located from index 16 to 48, and the HMAC key is located from index 48 to 80 all within the RSA plaintext. The HMAC tag is then created using the HMAC key and the createHmac() function with SHA-256. The HMAC tag is then read from the variable created at the beginning of the decryption function with a buffer and compared to the HMAC tag created with the HMAC key after putting it through the digest() function. They are compared in an "if" statement and an error message will be output if there is not a match. If there is a match then the AES ciphertext is decrypted. An IV is generated using createDecipherIv() with 256-bit AES in CBC mode, the AES key, and the IV key. The IV is then used to decrypt the AES ciphertext using the update() function and a buffer to parse the AES ciphertext with hex encoding. The AES plaintext is generated from this and will then be encoded into utf8 to make it readable to users on the client side with the final() function. The AES plaintext is returned from this decryption function.

3.3 Message Integrity Checks

The message integrity checks have been implemented by generating the HMAC tag with the createHMAC() function as explained in section 3.2 and is then compared to the HMAC tag passed in from the HMAC tag. The comparison statement in the "if" statement is the only integrity check needed in this code. /api/authenticate route checks the name and password against the database and provides a token if successful. /api/users lists all users and is protected by a token. ./api/getMessage returns the requested message if the tokens match.

3.4 Authentication Methods

JSON Web tokens are used for authentication in this project. Passport is the middleware used for Node.js and is easy to use with express-based web applications. Users are authenticated by their username and password, and when successful, a token is passed back. All of the API routes need to be authenticated.

3.5 Key Exchange

For this project, the key exchange will occur through email. Users that would like to message each other will send each other their public keys. The application will then instruct them store this public key in the filepath "C:\chatapp\keys" and will be asked to preserve the .pem file extension without tampering it as it will cause errors with such an action. Once that is completed, the users may begin messaging each other with end-to-end encryption.

3.6 Client Implementation

So far, only the server side and backend implementation has been discussed. However, client side implementation is also crucial to this for the application to be fully functional. For the client side, Java will be used in conjunction with the Node.js backend. The chat application will utilize node-java to design a portable exe file that will allow the application to run from any desktop machine. With this client implementation, scripts will be run and API requests will be made using Node.js and mongodb. All the user will see, however, is the frontend design in which it will be a GUI that will have an appearance similar to that of a terminal for simplicity. Should time permit it, the GUI will be designed to look closer to a browser for better user enjoyment. However, client implementation will be left to the last thing in the roadmap of the development of this application because security goals must be met first along with server side implementation and functionality.

4 Attack Surfaces

4.1 Access to the database

If a user has access to the database, they can steal other users login information and be able to log in as those users. User authentication is implemented to control access to the database. Messages and passwords are encrypted before being sent to the database, so any attacker would only be able to see encrypted data.

4.2 Man-In-the-Middle Attack

Man-in-the-middle attack is when the attacker secretly relays and alters the communication between two parties who believe they are directly communicating with each other. Implementing the encryption, decryption, and user authentication mention in section 3 combat against such attacks. using HTTPS and SSL also protects against this attack.

4.3 Authenticity

Users cannot interact with another user if they were not authenticated. Using JWT authentication ensures that a client can only interact with another authenticated client.

4.4 DDos Attacks

A distributed denial of service attack is when an attacker overloads the server with traffic from multiple sources in order to crash the server. Amazon web services has standard DDOS protection, with its AWS Shield service.

4.5 Hardcoded Keys or Tokens

If there are any hardcoded keys or tokens, an attacker may be able to gain access to these and decrypt messages or pose as another user. The server is stateless, which means it does not store any of the sensitive information like keys, and nothing is hardcoded that an attacker may be able to use.

4.6 Brute Force Algorithms

Some of the cryptographic algorithms are known to have weaknesses to brute force algorithms if they are not implemented the correct way. Using RSA with OAEP and a 256 bit key ensures that it cannot be easily brute force attacked. AES with CBC block mode is used to prevent this attack, along with using SHA256 for the HMAC hashing algorithm.

4.7 Replay Attack

A replay attack is when an attacker intercepts a request on the network and attempts to re-send the request with malicious intent and the user thinks that they completed the request successfully. JWT authentication protects against this by using an expiration time and a one-time generated pseudorandom number or nonce.

4.8 Other Vulnerabilities

4.8.1 User Behavior

As explained in section 3.5, the user must be trusted enough not to tamper with the public key. Otherwise, the messaging application will fail when the 2 users attempt to talk to each other because message integrity check will always fail which will result in constant error messages.

5 FUNCTIONALITY

5.1 Use Cases

Use Case	Description
Register	The user can create an account and be added to the database as a user. The user will have to enter their email, create a username, and a password.
Login	The user can log in using their username and password and be authenticated as a real user.
Reset Password	After 10 unsuccessful logins, the user will be sent a password rest link to their email allowing them to reset their password. This will only work when the user knows their username.
Send Message	The user can send a text message to another user.
Delete A Message	The user can delete a message they have received.
Delete a Conversation	The user can delete a conversation with another user.
Receive A Message	The user makes a pull request to receive new messages.
Logout	The user can log out of their account.