Information Security Institute

CAPSTONE FINAL REPORT

Social Media with End-to-End One-Time Encryption

STUDENT TEAM
Ziwei Li, Wang Qian, Zeyin Zhang

Advisor
Dr. Xiangyang Li

TABLE OF CONTENTS

1 ABSTRACT

This project presents the development of an innovative social media platform enhanced with end-to-end one-time encryption to ensure secure communication among users. In the digital age where data breaches and privacy concerns are rampant, our app addresses these challenges by integrating robust encryption techniques into a user-friendly social media interface. We employ a method where all group users share a common secret key and a one-time password key, generated every 30 seconds, for encrypting messages. The front-end development enables secure message transmission via HTTPS requests, while the back end, developed in Python, interacts with a MySQL database for storing chat records, ensuring data persistence and security. Our approach includes operational maintenance features like automatic deletion of chat records older than seven days, and a user interface that prioritizes ease of use while maintaining high-security standards. Significant advancements include stress testing, threat modeling and analysis, and exploring shared key methods. The system's architecture, designed to operate in a high-demand environment, ensures scalability, performance under load, and resistance to a wide array of cyber threats.

2 INTRODUCTIONS

In today's digital age, social media software plays a pivotal role in facilitating communication, enabling individuals to connect with friends, share images or documents, and coordinate team activities. However, beyond the convenience it offers, users are increasingly concerned about the potential privacy risks and the possibility of message leaks. Previous research has highlighted the various potential threats users face when using social media software [1]. Therefore, addressing security issues within these platforms is of paramount importance for their continued development.

Furthermore, an array of messaging applications, such as WhatsApp and Discord, have emerged to cater to different user preferences and needs. Each of these platforms boasts different features and, notably, varying levels of message confidentiality. Moreover, they are tailored for specific contexts and purposes. In this project, our aim is to explore the technologies employed by these software solutions to safeguard confidentiality and integrate them seamlessly into our own software.

Our primary focus will be on developing secure chat software. To enhance communication security, our approach will be to implement end-to-end encryption using symmetric encryption. Firstly, the sender and receiver should find a way to share a single secret key. Then for every 30 seconds, we will generate a one-time key, which is used to encrypt the plaintext based on AES. Based on the time and the secret key, the receiver can recover the one-time key and then decrypt the ciphertext. In this procedure, even if a package is inadvertently intercepted from the secure channel or the database server is compromised, attackers will be unable to decipher the ciphertext and access the original messages. On the other hand, even

though the attacker pretends to be the sender, the message still needs to be encrypted with the secret key held by the real sender so that the receiver can get the readable message by decrypting it with the same secret key.

3 LITERATURE REVIEW

In modern society, understanding the challenges of end-to-end encryption (E2EE) is critical. This literature review synthesizes various scholarly contributions, presenting a comprehensive analysis of E2EE in the context of secure messaging applications.

Husin addresses the critical need for securing chat sessions against data leakage, underscoring the importance of robust session management in maintaining confidentiality [1]. This concern forms the foundation for understanding the importance of E2EE as discussed in subsequent studies.

Alatawi provide an understanding of the requirements for a secure messaging system, emphasizing the importance of confidentiality, integrity, authentication, and forward secrecy among other aspects [2]. They underscore how E2EE ensures the privacy of messages during transmission, protecting against potential access by external entities such as governments, service providers, or hackers. Moreover, they highlight a vulnerability: the risk of Man-in-the-Middle (MitM) attacks due to the server-based distribution of encryption keys in most chat software. To mitigate this risk, the addition of authentication is suggested, ensuring secure communication.

Tyagi, Miers, and Ristenpart discuss the issue of untraceable content, a byproduct of enhanced confidentiality [3]. They present two methods for source attribution in messaging systems: path traceback and tree traceback. The path traceback is particularly efficient, creating an encrypted label to trace forwarded messages back to their origin. This feature is significant for identifying and mitigating the spread of harmful content such as fake news or malicious messages.

The debate over E2EE extends into the realm of national security, as discussed by Endeley [4]. While E2EE provides robust privacy protections, it also poses challenges for law enforcement in combating criminal activities. Endeley cites the example of WhatsApp to illustrate how governments see E2EE as an obstacle to national security. The paper also addresses the controversies surrounding "backdoors" in encryption software, using the case of Skype to exemplify the potential loss of trust and security.

Candra, Kurniawan, and Rhee analyze the security of Telegram, a popular instant messaging app that employs its proprietary MTProto encryption protocol [5]. Their study, which includes static and dynamic analysis, highlights Telegram's unique approach to encryption, particularly in its secret chat feature. This includes a two-step encryption process and self-destructing messages.

Agarwal explores a different aspect of messaging security: the proliferation of spam in encrypted messaging environments [6]. They study WhatsApp's approach to mitigating spam through message labeling, which marks messages containing URLs, email addresses, and phone numbers. While this method helps identify potential spam, it has limitations in capturing more sophisticated messages.

Neogi's exploration of WhatsApp's encryption protocol provides deeper insights into the technical workings of E2EE [7]. The paper explains the combination of cryptographic techniques employed by WhatsApp, highlighting the significance of the Signal Protocol and its components. This study explains the balance WhatsApp strikes between maintaining message confidentiality and analyzing metadata to understand user communication patterns.

Lastly, Schliep and Hopper introduce a novel approach to secure mobile group messaging [8]. Their work on the CoWPI system demonstrates how integrity, deniability, and conversation consistency can be achieved in group chats, even allowing for asynchronous participation by users. This study is particularly relevant in demonstrating the application of E2EE in a group messaging context.

4 PROBLEM DEFINITION

4.1 Current Expectation

With the proliferation of digital communication platforms, ensuring the security of chat applications has become paramount. As cyber threats continue to evolve, robust evaluation methods are essential to safeguard users' data. The next step is to test our End-to-End One-time encryption social media. Comprehensive testing and analysis are needed to ascertain its resilience against potential cyber threats. We will focus on a multi-faceted approach to assess the security of the chat application. The methods employed will include:

1) Enhanced Stress Testing to understand the system's limits and potential points of breakdown.
2) Threat Modeling and Analysis to identify potential threats and assess risks.
3) Exploration of Shared Key Methods to ensure secure data exchange.
4) Security Audit to review the system's security policies, and procedures.
5) Penetration Testing to simulate cyberattacks and assess the application's vulnerability.

To comprehensively evaluate the chat application's security framework and identify potential vulnerabilities. The aim is to fortify its defenses against cyber threats, ensuring a safe environment for its users.

5 TECHNICAL SOLUTION, DESIGN, AND ANALYSIS

5.1 Solution

5.1.1 Overview

Our project entails developing a secure communication application where encrypted messages are sent through a dedicated server API, utilizing a shared secret key for encryption/decryption, and incorporating a rolling One Time Password (OTP) mechanism for enhanced security. The technical solution pivots around a client-server architecture.

Our encryption communication method is all group users now share a common secret key (e.g., "thisisasecretkey") and a key nickname (hash1(secret key)). Each group is assigned a unique channel name known only to its members. An OTP key of 6 digits is generated every 30 seconds based on the secret key.

On the front-end, messages are encrypted with AES-256, using an OTP (which is generated by hashing a shared secret key), and then Base64 encoded, and transmitted via HTTPS POST/GET requests to the server. The back end, developed in Python, interacts with a MySQL database to store/retrieve messages, ensuring data persistence while adhering to a 7-day retention policy managed via Crontab. Both message transmission and retrieval endpoints are engineered to be secure and efficient, utilizing middleware for rate limiting and ensuring robust error handling. On the User Interface side, we developed the UI for displaying chat history with appropriate colors based on user nicknames. The main page now features Edit and Settings buttons, providing additional functionality. And on the Settings page, we allow users to input the server URL.

On the back end, to ensure the availability of the server in various conditions. We decided to separate the database and the backend server into independent parts so that even though the backend server came up with some accident, we can still deploy another node as the substitute. Moreover, the database input is only bound to our server. And on the Settings page, we allow users to input the server URL. Based on the stress testing, we implemented rate-limiting middleware to handle up to 100 requests per second, this can be increased by upgrading the server and optimizing program code. We developed two APIs: /postChat and /getChat as described in the design draft. And we implemented a Crontab job to automatically delete chat records older than 7 days.

Meanwhile, we deploy both the server and the database with the AWS cloud. Since typically we don’t have the ability to handle the physical threat and the physical server environment, it would be more secure to utilize the AWS cloud environment.

5.1.2 Front-End Design

5.1.2.1 Overview

This APP designed using SwiftUI, an innovative UI toolkit by Apple, allows for the creation of declarative UIs across all Apple platforms. In our design, we leverage this toolkit to provide a responsive experience for the users.

5.1.2.2 Security Implementation

Encryption and Decryption:
Security is paramount. We have implemented AES encryption and decryption to safeguard user messages. The Encrypt.swift file is the key component here.

```swift
func AESEncryptWithString(input: String, key: String) -> String? {
//    let symmetricKey = SymmetricKey(data: key.data(using: .utf8)!)
    let symmetricKey = AESKey(from: key)!

    return AESEncrypt(input: input, symmetricKey: symmetricKey)
}

func AESEncrypt(input: String, symmetricKey: SymmetricKey) -> String? {
    guard let inputData = input.data(using: .utf8) else {
        return nil
    }

    do {
        let sealedBox = try AES.GCM.seal(inputData, using: symmetricKey)
        let combinedData = sealedBox.combined
        return combinedData?.base64EncodedString()
    } catch {
        print("Encryption failed: \(error.localizedDescription)")
        return nil
    }
}

func AESDecryptWithString(input: String, key: String) -> String? {
//    let symmetricKey = SymmetricKey(data: key.data(using: .utf8)!)
    let symmetricKey = AESKey(from: key)!

    return AESDecrypt(input: input, symmetricKey: symmetricKey)
}

func AESDecrypt(input: String, symmetricKey: SymmetricKey) -> String? {
    guard let inputData = Data(base64Encoded: input) else {
        return nil
    }

    do {
        let sealedBox = try AES.GCM.SealedBox(combined: inputData)
        let decryptedData = try AES.GCM.open(sealedBox, using: symmetricKey)
        return String(data: decryptedData, encoding: .utf8)
    } catch {
        print("Decryption failed: \(error.localizedDescription)")
        return nil
    }
}

func AESKey(from string: String) -> SymmetricKey? {
    // Ensure the input string is not empty
    guard !string.isEmpty else {
        print("KeyGen failed: Empty String")
        return nil
    }

    // Hash the string using SHA256, resulting in 32 bytes data, which is suitable for AES256
    let hash = Hash(from: string)

    // Convert the hash to a symmetric key
    let key = SymmetricKey(data: hash)

    return key
}
```

Figure1: Code snippet for AES

Time-Based One-Time Password (TOTP) Algorithm:
Our TOTP implementation in TOTP.swift enhances security. It generates temporary codes based on a shared secret and the current time, thereby reducing the risk of unauthorized access.

```swift
func GetHashedString(_ s: String, _ n: Int64) -> String {
    // Combine s and n into a single string
    let input = s + String(n)

    // Compute the SHA-1 hash value
    let h = Insecure.SHA1.hash(data: input.data(using: .utf8)!)

    // Convert the digest to an array of bytes
    var bytes = [UInt8](repeating: 0, count: 20) // Use 20 instead of
        h.count
    h.withUnsafeBytes {
        bytes = Array($0)
    }

    // Parse the first 6 bytes of the hash value as an int64
    var result: Int64 = 0
    for i in 0..<6 {
        result |= Int64(bytes[i]) << (40 - 8 * i)
    }

    // Take the result modulo 10^6 and format it with leading zeros
    return String(format: "%06d", result % 1_000_000)
}

func Now() -> Int64 {
    return Int64(Date().timeIntervalSince1970)
}
```

Figure2: Code snippet for TOTP

5.1.2.3 Data Management

SQLite Integration:
We chose SQLite for local data storage due to its lightweight and efficient nature. The Client.swift file demonstrates our approach to managing chat sessions and messages in the local database.

```swift
    // execute raw sql
    func ExecRaw(_ sql: String) -> Int32 {
        var errorMessage: UnsafeMutablePointer<Int8>?
        if sqlite3_exec(db, sql, nil, nil, &errorMessage) != SQLITE_OK {
            if let error = errorMessage {
                print("Error executing SQL: \(String(cString: error))")
            }
            return SQLITE_ERROR
        }
        return SQLITE_OK
    }
```

Figure3: Code snippet for SQLite

Data Synchronization:

The application synchronizes data between the local database and the server. This ensures that the user has access to their messages across different devices.

```swift
extension ChatClient {
    // Setup timer
    func startPolling() {
        print("start polling")
        Timer.scheduledTimer(withTimeInterval: 5.0, repeats: true) { [weak self] _ in
            self?.handleFetchAndUpdateChats()
        }
    }

    func handleFetchAndUpdateChats() {
        Task {
            do {
                try await self.fetchAndUpdateChats()
            } catch {
                print("Error: \(error)")
            }
        }
    }

    // FIXED: weak self -> Solving circular reference issues

    // Fetching chat and update database
    func fetchAndUpdateChats() async throws {
        let sessions = self.GetSessions()
        print("sessions:")
        print(sessions)
        for session in sessions {
            fetchAndUpdateChat(chatSession: session)
        }
    }

    func fetchAndUpdateChat(chatSession: ChatSession) {
        let lastTime = GetLatestChatTime(sessionID: chatSession.id)
        let contentsDict = GetChatAPI(secretKey: chatSession.secretKey, lastTime: lastTime)
        // Insert into database
        for contentDict in contentsDict {
            if let time = contentDict["time"] as? Int64 {
                InsertMessage(sessionID: chatSession.id, isUser: false,  nickname:
                    contentDict["user_nickname"] as? String ?? "",  message: contentDict["body"]
                    as? String ?? "", dbTime: time)
            }
        }
    }
```

Figure4: Code snippet for Polling

5.1.2.4 User Interface and Interaction

Navigation and Views:
The user interface is structured into various views such as GeneralView, UserView, and ServerView. These views are designed to be intuitive and easy to navigate. Users can customize settings like nickname and server address in the Settings.swift file, allowing for a personalized chat experience.

```swift

import SwiftUI

struct GeneralView: View {
    var body: some View {
        VStack{
            NavigationView {
                Form{
                    Section(header: Text("User")) {
                        NavigationLink("Nickname") {
                            UserView()
                        }
                    }
                    Section(header: Text("Server")) {
                        NavigationLink("Server Address") {
                            ServerView()
                        }
                    }
                }
            }
        }.navigationBarTitle("General")
    }
}

struct GeneralView_Previews: PreviewProvider {
    static var previews: some View {
        GeneralView()
    }
}

```

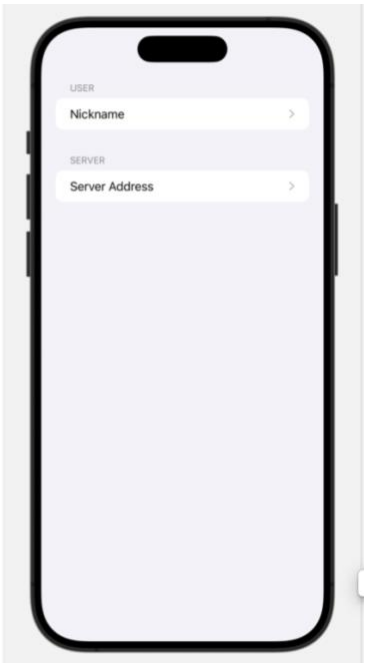

Figure5: Code snippet for Settings

Session Management:
In NewChatSessionView, users can create new chat sessions, highlighting the flexibility of the application.

```swift
struct NewChatSessionView: View {
    @State private var secretKey: String = ""
    @State private var nickname: String = ""
    var addSession: (String, String) -> Void

    var body: some View {
        VStack {
            TextField("Secret Key", text: $secretKey)
                .padding()
            TextField("Nickname", text: $nickname)
                .padding()
            Button(action: {
                addSession(secretKey, nickname)
            }) {
                Text("Add Chat Session")
            }
            .padding()
        }
        .padding()
    }
}

struct NewChatSessionView_Previews: PreviewProvider {
    static var previews: some View {
        NewChatSessionView { secretKey, nickname in
            print("Secret Key: \(secretKey), Nickname:
                \(nickname)")
        }
    }
}

// FIXED: Pop-up window addSession closure call
```

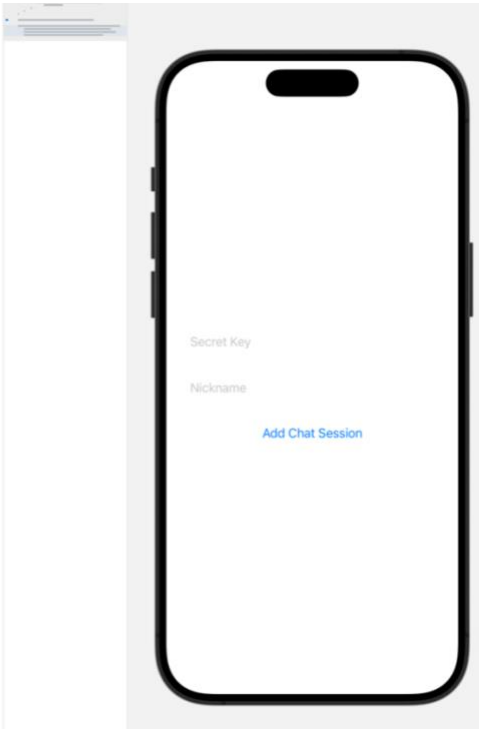

Figure6: Code snippet for Add Session

Chat Functionality:
The core of iChat lies in its chat functionality. The ContentView.swift and Chat.swift files provide the layout and logic for displaying chat messages and handling user inputs.

```
43
44              HStack {
45                  TextField("Enter message...", text:
                        $currentMessage)
46                      .padding(10)
47                      .background(Color.gray.opacity(0.2))
48                      .cornerRadius(8)
49
50                  Button("Send") {
51                      if !currentMessage.isEmpty {
52                          let now = Now()
53
54                          // 1 Insert into local database
55                          client.InsertMessage(sessionID:
                                chatSession.id, isUser: true,
                                nickname: conf.Nickname, message:
                                currentMessage, dbTime: now)
56
57                          // 2 Send to server
58                          _ = SendChatAPI(secretKey:
                                chatSession.secretKey,
                                currentMessage: currentMessage,
                                time: now)
59                          currentMessage = ""
60                      }
61                  }
62              }
63          }
64          .padding()
65      }
66  }
67
```

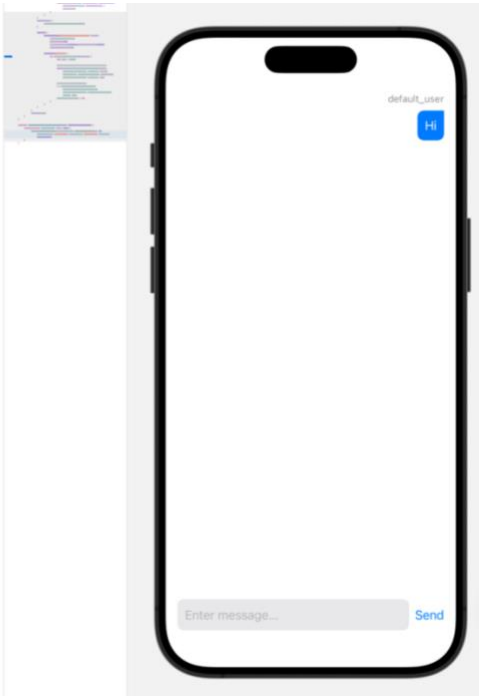

Figure7: Code snippet for Chatting

5.1.2.5 Configuration and Setup

App Configuration:
The Conf.swift file contains configurations like server IP, API routes, and user nickname, making it easy to modify the app settings.

```
10  class Conf: ObservableObject {
11
12      @Published var ServerIPPort: String
13      @Published var GetChatRouterFormat: String
14      @Published var PostChatRouterFormat: String
15
16      @Published var Nickname: String
17
18  //    @Published var dbPath = ""
19  //    @Published var db: OpaquePointer?
20
21      init() {
22          ServerIPPort = "https://127.0.0.1:8000"
23          GetChatRouterFormat = "/getChat?time=%@&key_nickname=%@"
24          PostChatRouterFormat = "/postChat"
25          Nickname = "default_user"
26      }
27  }
28
29  var conf: Conf = Conf()
```

Figure8: Code snippet for Conf

5.1.3 Back-End Design

Basically, we utilize the Django framework to implement the overall system for the backend server systems. It primarily handles two tasks: 1. Receive the post request from the client side, which indicates a new message is sent through a specific channel. 2. Receive the get request from the client side, which typically requests for an updating of the sent messages in one specific channel. I will mainly talk about the workflow design and security concerns of our server system in the following paragraph.

As you can see, when the users send out a post request or a get request to the server, we need to accept the data from the user, which can be potentially dangerous if there is malicious data in it. Plus, for example, when the server is going to process a GET request, it needs to compare the username and timestamp. There are some chances for potential SQL Injection attacks if the username is specifically designed to contain tokens like “and True”. If that is the case, the attacker may be able to capture the chatting history from other users, which can violate the guarantee of privacy.

As a result, instead of constructing a raw SQL query, we utilize the built-in ORM QuerySet API to avoid those potential problems. The ORM QuerySet API and other frameworks protect the server from potential string formatting SQL injection attacks by converting or transforming potential unsafe user input into a harmless form. So basically, the comparison of username and timestamps required in the GET request are processed through a more secure way, which will conduct checks and transformations on the input to protect the server systems from attacks such as SQL injection attacks. As you can see in the following code block, the user input key_nickname will be used as a whole parameter to compare, so there is no room for bypass the check through “and True”.

```
key_nickname = request.GET.get("key_nickname")
ctime = request.GET.get("time")
rs = tb_user_chat.objects.filter(chat_time__gt = ctime, user_nickname = key_nickname)
data = []
for r in rs:
    d = dict()
    d["time"] = r.chat_time      EC2 Default User, last month • [auto-cancel] update i
    d["content"] = r.chat_body
    data.append(d)

returnContent = dict()
returnContent["data"] = data
returnContent["errno"] = 0
returnContent["errmsg"] = "Success"
return Response(returnContent)
```

Figure9: Code snippet for Get Request

On the other hand, even though in the real case we may have a larger disk space and better storage device, we still need to be careful with the potential Denial-of-Service attacks. For example, the user may keep sending useless information which can potentially occupy the database system storage in a relatively fast space.

To avoid such kinds of issues, we also restrict the number of messages that can be sent by a user in one minute. The default setting of that limitation is 300 requests per second. We utilized the rest framework in our project to identify the IP as an identification and provide a brief frequency control on a certain client. In all, if a user frequently sends the message to our server systems and surpasses the limitation of the single client, the request will be rejected and the server systems will send back an error message to inform the user that he may send the messages too frequently and should wait for some time to initiate a new request, which is shown in the following code block.

Figure10: Code snippet for Rest

The error code is mainly divided into several types of error to notify the user how to deal with it.

6 EXPERIMENTATIONS, EVALUATION, AND RESULT ANALYSIS

6.1 Enhanced Stress Testing

t2.micro			
vCPU	1	Model name	Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz

Architecture	x86_64	Stepping	2
CPU op-mode(s)	32-bit, 64-bit	CPU MHz	2400.226
Byte Order	Little Endian	BogoMIPS	4800.04
CPU	1	Hypervisor vendor	Xen
On-line CPU(s) list	0	Virtualization type	full
Thread(s) per core	1	L1d cache	32K
Core(s) per socket	1	L1i cache	32K
Socket	1	L2 cache	256K
NUMA node	1	L3 cache	30720K
Vendor ID	GenuineIntel	NUMA node0 CPU	0
CPU family	6	Flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca……
Model	63	MemTotal	975592 KB
Storage	8GB		

Table1: Server setting

The stress testing was conducted to evaluate the performance of our system under extreme conditions. For this, we utilized the ab command, a tool for benchmarking our server. The specific command used was:
ab -n 1000 -c 100 '3.133.149.139:8000/getChat?time=0&key_nickname=key'

This command simulates a thousand requests to our server with a concurrency of 100 users.

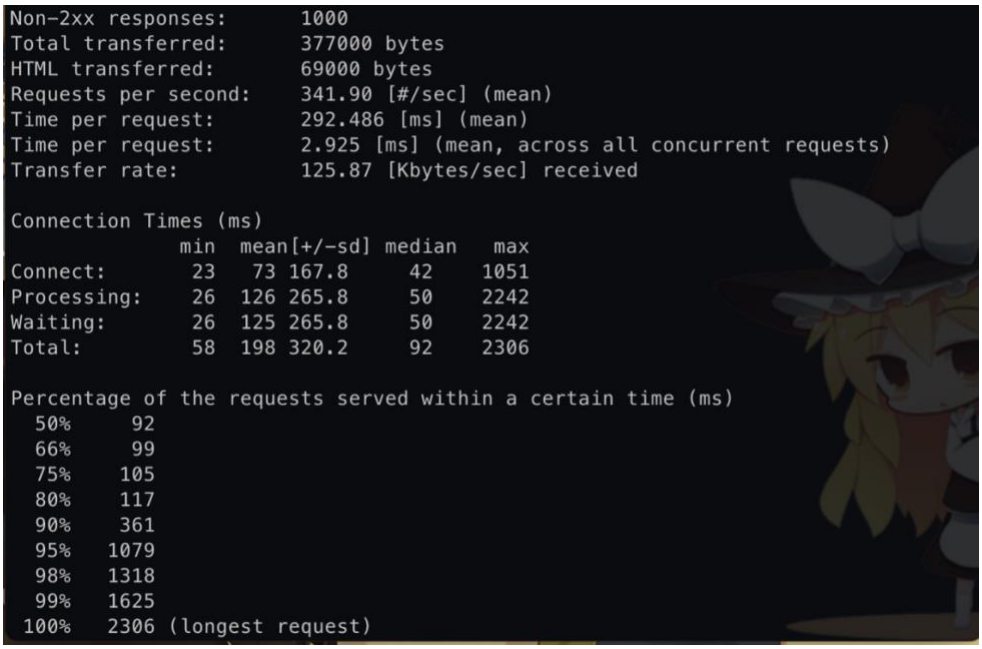

Figure11: Result for 100 concurrent requests

During the stress test with 100 concurrent requests, we observed that approximately 5% of the requests experienced a delay exceeding one second, with the maximum delay recorded at 2300 milliseconds. This result indicates that the current capacity of our leased server has reached its threshold. Because 2 seconds is obvious to users.

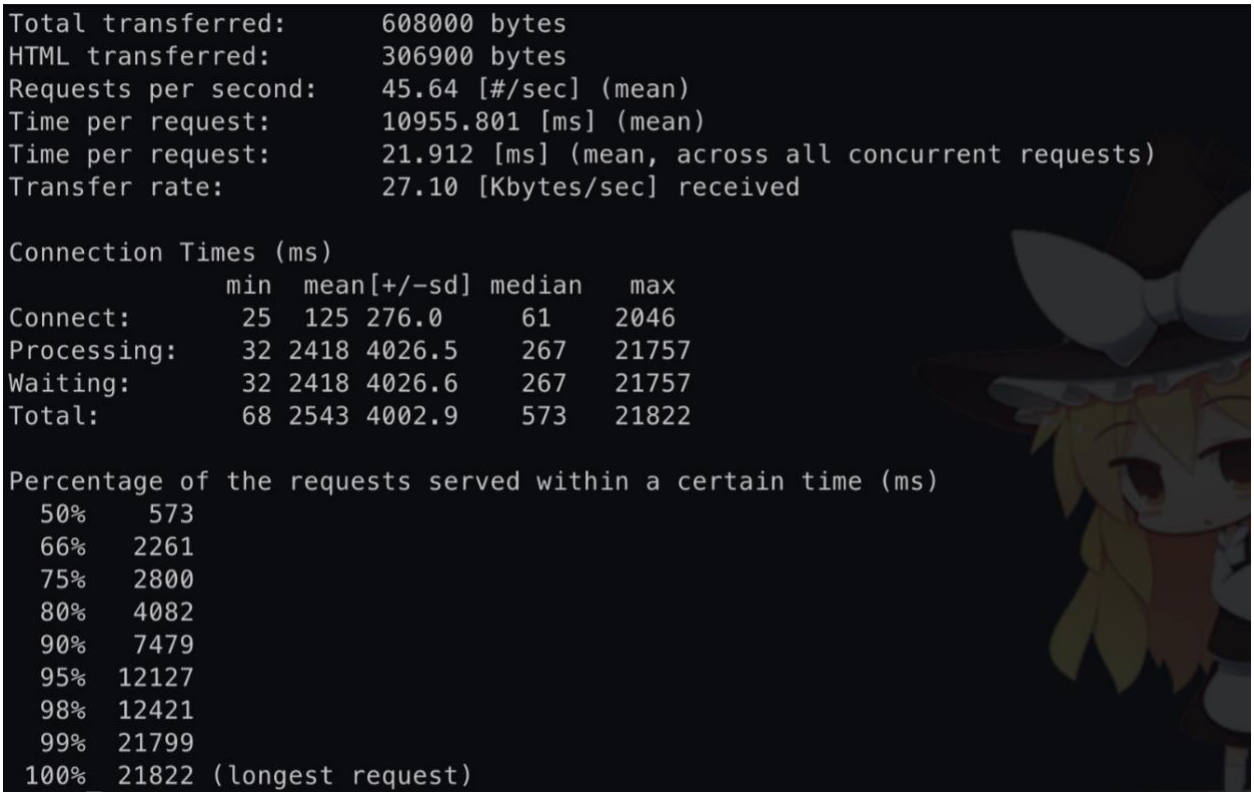

Figure12: Result for 500 Concurrent Requests

When the test was scaled to 500 concurrent users, the results were more concerning. About 50% of the requests had delays over one second. 10% of these requests experienced significant delays ranging from 7 to 21 seconds. Such extensive delays can severely impair user experience, potentially leading to user attrition.

6.2 Future and Optimizations

In the initial phase, our primary focus will be on maintaining the number of concurrent users around 100. This strategy is not only about managing server load but also about ensuring that each user has a smooth and responsive experience. During this period, we will be gathering user feedback to make incremental improvements to our application.

Our user base expands, we will prioritize these improvements and adapt our approach. The feedback will guide us in refining the user experience and identifying the most critical areas for optimization. This could involve enhancing specific features. In this part, we have identified two main avenues for optimization: refining our codebase or upgrading to a more robust server.

Our goal is to ensure that as the demand increases, our system is robust and capable enough to accommodate growth seamlessly, thereby sustaining and enhancing user satisfaction.

6.3 THREAT MODELING AND VULNERABILITY ANALYSIS

6.3.1 Situation 1

In our chat application, a critical aspect of ensuring security involves analyzing potential vulnerabilities. A regular user, User A, shares a piece of SQL code with another user, User B, in a chat. This situation, although seemingly innocuous, could inadvertently lead to SQL injection attacks. Such attacks can compromise the integrity and confidentiality of the database by allowing attackers to manipulate SQL queries.

To reduce this risk, we have used Python's Django framework, it has MySQL Object-Relational Mapping feature. The Object-Relational Mapping provides a layer of abstraction between our application code and the underlying SQL database. It allows us to interact with the database using Python objects, thereby avoiding the direct use of SQL statements. By doing this, it can reduce the risk of SQL injection, as the Object-Relational Mapping automatically handles the necessary query parameterization and escaping, ensuring that any SQL code shared within the chat is not executed as part of the database query.

This approach not only enhances the security of our application against SQL injection attacks but also simplifies the database interaction process. This leads to a more secure application.

6.3.2 Situation 2

Another situation in our application involves the potential threat from malicious users, for example, User C, attempting a man-in-the-middle attack through DNS spoofing. In this kind of attack, User C could intercept and manipulate the communication between two legitimate users, such as User A and User B, potentially leading to the theft of their private chat records. This kind of attack is insidious as it can occur without the knowledge of the users involved, making it a significant threat to the confidentiality and integrity of user's data.

To combat this threat, our application employs HTTPS, a secure communication protocol over the internet. HTTPS uses Transport Layer Security to encrypt data transmitted between the user’s device and our servers. This encryption ensures that even if a malicious actor like User C were able to intercept the data, they would not be able to decrypt the contents of the communication.

By using HTTPS, we ensure that all data, including chat messages, are securely encrypted in transit. This approach not only protects against DNS spoofing and man-in-the-middle attacks but also strengthens user trust in our application's ability to safeguard sensitive information.

6.3.3 Situation 3

In this situation, we address a potential threat posed by a malicious user, for example, User A. User A attempts to overwhelm our server by sending a barrage of invalid or nonsensical messages. This kind of flood attack can lead to server resource exhaustion, resulting in the

service being slow or entirely unresponsive for users. It's a form of Denial-of-Service attack, where the attacker's goal is to disrupt the normal functioning of the application by overloading the server with excessive requests.

To reduce the probability of this risk, we have implemented a rate-limiting strategy on our server. This way involves setting a cap on the number of requests that can be sent from a single IP address per minute. By using this limit, we can prevent an individual user from sending an overwhelming number of messages in a short period, thereby protecting our server resources from being down by flood attacks.

This rate-limiting can safeguard our server from unnecessary strain and maintain the quality of service for other users. It ensures that our server can efficiently handle legitimate requests without being shut down by malicious traffic. This measure limits the potential attackers. Rate-limiting can ensure our user's experience. It also can ensure our application's availability.

6.3.4 Situation 4

In this situation, we address the risk posed by a malicious user, for example, User A, this user attempts to exploit vulnerabilities in our system by connecting to various ports and probing for weaknesses. This type of attack involves the attacker scanning a range of ports on our server, seeking to find and exploit open ports that might provide unauthorized access. The potential consequences of such an attack are severe, ranging from unauthorized access to sensitive data to the server or database being compromised.

To combat this threat, we have adopted a port management strategy. Our approach involves closing all unnecessary ports and services such as FTP, SMTP, SMB, and others that are not essential to our application's functionality. For example, the FTP protocol itself is not necessarily secure. We maintain only the essential ports - 8080 port, POST, and GET to ensure secure and focused communication channels.

The port management strategy greatly reduces the attack surface of our server, limiting the opportunities for an attacker to gain unauthorized access. By restricting the number of open ports, we minimize potential vulnerabilities and strengthen the security of our system. This measure is important to protect our server and database from being hacked. It ensures the integrity and confidentiality of our application.

6.3.5 Situation 5

In this situation, we confront a threat from a malicious user, for example, User A, who attempts to orchestrate a Distributed Denial of Service attack. This is executed by infecting multiple client devices with a virus, enabling the attacker to control these devices to simultaneously send an overwhelming amount of traffic to our server. This kind of attack can quickly deplete server resources, leading to service disruptions and potentially causing the server to crash, denying service to legitimate users.

To counter this sophisticated form of attack, our response involves a two-step approach. Firstly, we deploy automated monitoring systems to detect unusual traffic patterns indicative of a Distributed Denial of Service attack. This system alerts us to potential threats in real time, allowing us to do a quick response. Then we implement a manual review process. This involves scrutinizing the source of the suspicious traffic to identify and confirm if specific client devices have been compromised and are part of the attack. Once identified, we will block the IP addresses of these compromised devices. This action prevents them from further participating in the attack and mitigates the impact on our server.

While automated systems are crucial for initial detection, the manual review is essential to accurately identify and deal with compromised devices. This approach ensures that we don't inadvertently block legitimate users while effectively neutralizing the threat.

6.3.6 Situation 6

In this situation, we address a potential vulnerability related to file transfers within our chat application. The concern here is the risk of malicious file uploads or the embedding of harmful code in image files, this is known as steganography. This type of threat involves users unknowingly or maliciously uploading files that contain hidden malicious code. Once these files are uploaded and accessed by other users, they can execute harmful actions or compromise the security of the application and its users.

To combat this risk, we will implement multiple layers of defense. We can enforce strict limitations on the types of files that can be uploaded. By restricting file uploads to certain safe formats, we reduce the likelihood of malicious files being introduced into our system. This will be a proactive measure to filter out many common types of harmful files. Besides preventing file type restrictions, we will employ a comprehensive scanning process for all uploaded files. This process involves the use of advanced malware detection software that scans each file for any signs of malicious code or hidden payloads. The scanning process is thorough and is designed to detect even sophisticated forms of malware that may be embedded in seemingly innocuous files.

Through these measures, we will ensure that the files shared within our chat application are safe from malicious content. This not only protects the application and its infrastructure but also safeguards the users from inadvertently downloading or sharing harmful files. This is important for the trust and safety of our user community.

6.3.7 Situation 7

In this situation, we face the risk of a buffer overflow attack, a common and potentially devastating security threat. The buffer overflow occurs when a program attempts to write more data to a fixed-length block of memory, or buffer, than it can hold. This can lead to overwriting adjacent memory, which may contain other data or control information. In our chat application,

such an attack could be executed by inputting large amounts of data, aiming to exploit vulnerabilities within the system, potentially leading to unauthorized access or control.

To reduce the risk of buffer overflow attacks, we will use a double strategy approach. The first defense is the use of Django's Object-Relational Mapping system for dynamically generating and inserting data. Django Object-Relational Mapping acts as a protective layer that abstracts the database interactions, preventing the direct insertion of raw data into the database. This method ensures that data is properly handled and sanitized before being inserted, thereby reducing the risk of buffer overflow attacks due to unchecked data input. Over more, we will implement data size restrictions within our MySQL database. This involves setting explicit limits on the size of data that can be stored in each field of the database. By enforcing these limits, we ensure that any attempt to insert data beyond the permissible size is automatically rejected by the database system. This not only prevents buffer overflows but also reinforces the integrity of our database by maintaining control over the data storage.

These measures will form a strong defense against buffer overflow attacks. This proactive security posture is crucial in maintaining the safety and reliability of our chat application, ensuring that it remains resilient against such threats.

6.3.8 Situation 8

In this situation, we explore the implications of a regular user, for example, User A, having their device stolen. Such an incident can pose a security threat, particularly if the stolen device contains sensitive keys used for encrypting chat communications. The attacker, having access to these keys, could potentially use them to send false messages to another user, User B, under the guise of being User A. This not only breaches the confidentiality of the communication but also erodes trust within the user community.

To address this challenge, our solution is to encourage User A to promptly notify our support team and User B about the stolen device. This notification can be made through alternative secure channels, such as email. Once notified, our team takes immediate action to disable the compromised keys and initiate the process of key replacement. At the same time, user B is advised to stop all communication with user A and await the issuance of new encryption keys. The new keys are securely shared with User B, ensuring that future communications are safeguarded.

Our application for handling stolen devices and key theft is designed to reduce risks while maintaining the integrity and confidentiality of user communications. By empowering users to be part of the response process and providing them with the necessary support and tools, we ensure that the impact of such incidents is minimized.

6.3.9 Situation 9

This situation is the risk from a malicious user, for example, User B, who attempts to compromise the security of our chat application through a brute force attack aimed at obtaining the keys of a regular user, User A. In a brute force attack, the attacker systematically checks all possible keys until the correct one is found. If successful, such an attack would lead to a complete breach of User A's communications, exposing sensitive information and breaking confidentiality, and it also has the potential impact on integrity and availability.

To thwart such attempts, our application employs a dynamic key management strategy that involves the regular rotation of encryption keys. Our encryption keys are automatically changed every 30 seconds. This frequent key rotation significantly increases the complexity and difficulty of a brute-force attack. By the time an attacker might be close to cracking a key, it would have already been replaced with a new one, rendering the effort futile.

This strategy ensures that even if an attacker were to employ powerful computing resources in their brute force attempt, the constantly changing keys would make it nearly impossible to gain unauthorized access to a user's communications. This approach protects against brute force attacks and strengthens security, ensuring that our users' communications remain confidential and secure.

6.3.10 Situation 10

In this situation, we face a non-technical but equally dangerous threat: social engineering. A malicious user, User A, employs deceptive social tactics to trick other users into disclosing their encryption keys. This type of threat exploits human psychology rather than system vulnerabilities. If successful, the attacker, User A, could gain unauthorized access to private communications by using the disclosed keys, leading to a significant breach of privacy and security.

To combat this threat, our application includes proactive measures to educate and remind users about the importance of key security. We regularly disseminate information and alerts within the app, highlighting the dangers of sharing encryption keys and advising users on how to recognize and avoid phishing emails and other forms of deceptive communication. This educational approach is important, as it empowers users to be the first line of defense against such social engineering attacks. We will implement warning systems within the chat application. These systems will detect and alert users when they are about to share sensitive information, such as encryption keys. The alerts prompt users to reconsider their actions and provide guidance on maintaining key confidentiality.

This strategy can help prevent key disclosure through social engineering but also strengthen the overall security culture within our user community.

6.3.11 Situation 11

This situation faces a common security risk: the same passwords are used on multiple platforms. A regular user of our chat application, User A, has their encryption key compromised on another platform. This vulnerability is due to User A's practice of using the same password for multiple services, including our chat application. This password reuse poses a significant threat, as the breach on one platform could potentially lead to unauthorized access to the user's chat records in our application, resulting in a privacy breach.

To counter this risk, our strategy involves educating users about the dangers of password reuse. We encourage our users to adopt unique passwords for different websites and applications. This guidance is provided through various channels, including in-app notifications, and email communications. The goal is to raise awareness about the importance of using different passwords for each online service to enhance their security. And we will provide features within our application to facilitate secure password management. This includes password strength and suggestions for strong.

Educating users on unique passwords is important to user security and privacy.

6.3.12 Situation 12

In this situation, we consider the risks associated with a regular user, using our chat application while connected to public Wi-Fi. Public Wi-Fi networks, often found in places like cafes, airports, and hotels, are notoriously insecure due to their typically lax security protocols. This lack of security makes communications over these networks susceptible to being monitored or intercepted by malicious actors. For User, this means that their private messages and potentially sensitive information could be at risk of being accessed by unauthorized third parties.

To reduce this risk, our application will educate and alert users about the dangers of using public Wi-Fi for sensitive communications. We provide notifications and reminders that advise users to avoid or be cautious when using public Wi-Fi networks for accessing the chat application. Our application employs end-to-end encryptions for all communications. This encryption ensures that, even in the event of data interception on an insecure network, the content of the messages remains protected and unintelligible to anyone other than the intended recipient. While encryption provides a significant layer of security, we emphasize to our users that the best practice is to avoid public Wi-Fi for sensitive activities altogether.

By combining user education, in-app alerts, and robust encryption, our goal is to safeguard our users’ communications against the vulnerabilities inherent in public Wi-Fi networks. Our approach protects the data integrity, especially in less secure online environments.

6.3.13 Situation 13

In this situation, we have the potential security risks arising from regular user A's device operating system containing unpatched vulnerabilities. If the operating systems are not

regularly updated, can have security flaws that malicious actors can exploit. For users of our chat application, such vulnerabilities pose a significant risk as they could lead to unauthorized access and potential leakage of messages stored locally on the device.

To address this risk, our application regularly informs and reminds users about the importance of keeping their device's operating system up to date. Through notifications and periodic reminders, we educate users on the critical role that software updates play in maintaining device security. These updates often contain patches for known vulnerabilities, making them essential for protecting against potential security breaches. This proactive measure greatly reduces the window of opportunity for attackers to exploit outdated systems. And we will continuously update our application to align with the latest security standards and practices.

By actively educating users about the importance of regular operating system updates and designing our application to complement these updates, our goal is to reduce the risk of message leakage due to unpatched security vulnerabilities in user devices.

6.3.14 Situation 14

In this situation, we address the issue of device failure experienced by a regular user. Device malfunctions can occur unexpectedly and may lead to the loss of important data, including chat records that are not backed up.

To reduce the impact of such incidents, our application will have a server-side backup strategy. We will maintain a backup of all chat records on our server for a period of seven days. This policy provides users with a window of opportunity to recover their chat records in the event of a device failure. When a device malfunctions, user can get back their lost chat records by logging into their account from a different device. After logging in, they can access and download their chat history from the last seven days. This feature can ensure continuity for user communications.

Through these measures, we aim to protect our users from the adverse effects of device failures.

6.3.15 Situation 15

In this situation, we address the threat posed by a malicious user, User C, who attempts to deploy keylogging malware to monitor the keystrokes of another user, User A. Keylogging is a form of cyberattack where malicious software records every keystroke made by a user, with the intent of capturing sensitive information such as passwords or encryption keys. If User C successfully installs keylogging software on User A’s device, they could potentially gain access to User A’s encryption keys and, consequently, their encrypted communications.

To counter this threat, we recommend users to use trusted virtual keyboards. Virtual keyboards are software-based keyboards that appear on the device's screen, allowing users to input

information without using the physical keyboard. Virtual keyboards can help to reduce the risk of keylogging, as the keystrokes made on a physical keyboard are not recorded by the keylogging malware. We will provide users with information and guidance on how to identify and avoid potential malware threats. This will educate users on avoiding the download of applications from untrusted sources.

On our side, we will constantly update our application to enhance our users' overall security posture. By combining user education and recommendations for virtual keyboard usage, we aim to protect our users from keylogging threats, to protect their encryption keys and the confidentiality of their communications.

6.3.16 Situation 16

In this situation, we face the risk associated with a regular user, accidentally clicking on a malicious link provided by a dangerous user. Clicking on such links can be especially dangerous as it may lead to the user's device being infected with malware. This malware can range from spyware, which monitors and steals sensitive information.

To combat this threat, our application will implement multiple ways to approach it. We will integrate real-time link analysis and warning systems within the chat interface. When a user receives a link, our system automatically assesses it for known patterns of malicious content. If a link is flagged as potentially dangerous, the user is immediately warned through an in-app notification, advising them to exercise caution. This notification will include information on why the link was flagged and the potential risks of proceeding to the website. We educate our users on the importance of being alert when clicking on links, especially those from unknown or untrusted sources. This education is provided through regular security tips and reminders within the application and through our user support.

If we combine technology with user education, we believe we can create a secure environment where users can communicate without fear of potential security problems.

6.3.17 Situation 17

In this situation, we face the issue of high energy waste caused by our chat application when it is kept running in the background for extended periods by a regular user. The reason for the energy usage is the application's reliance on polling, rather than push notifications or long-lived connections. Polling involves the app repeatedly checking the server for new messages, this is effective for message delivery but can be more power-intensive.

To lighten this energy waste issue, we will implement a feature that intelligently manages the app's background activity. After the application has been running in the background for a predetermined period, it automatically transitions into a low-energy state by ceasing its polling activity. This approach significantly reduces the app’s energy usage when not actively in use, preserving the device's battery life. At the same time, we need to ensure that this reduction in

activity does not compromise the user's ability to receive timely notifications. When a new message is sent to the user, the app should resume its polling activity or initiate a push notification to alert the user, depending on their settings. This balance allows the app to remain responsive and efficient without unnecessarily using the device's battery.

Through these measures, we aim to minimize the energy waste in our application.

6.4 Key Exchange

6.4.1 Secure Key Sharing for User Communication

To start secure conversations within our application, users are required to exchange encryption keys in a unique and personal way. This is done to establish a secure line of communication between the two users. Our current system leverages external, trusted communication channels - phone calls or emails, for this key exchange process.

6.4.2 Process Overview

6.4.2.1 Key Generation

When a user downloads our app and create an account, our application generates a unique encryption key for each user. This key is crucial for encrypting and decrypting messages within the app.

6.4.2.2 Key Exchange

Users share their keys with each other using external communication channels.

Phone Calls: Users can verbally share their encryption keys during a phone call. This method is beneficial for its immediacy and the difficulty in intercepting voice communication.

Emails: Users can also choose to share their keys via email. While convenient, it is required to ensure that the email account is secure and that the email is sent to the correct recipient to prevent unauthorized access.

6.4.2.3 Adding as Friends

Once both users have shared their keys, they can add each other as contacts within the app. The shared keys are used to encrypt and decrypt messages exchanged between these users, ensuring that their conversations remain private and secure.

6.4.2.4 Key Verification

After exchanging keys, users should verify that the keys match, ensuring that there has been no error or interception during the exchange process.

6.4.3 Security Considerations

While using phone calls and emails for key exchange is effective, there are still things that need attention:

1) Verify the identity of the person on the other end of the call or email.
2) Ensure the security of the email account used for key exchange.
3) Be aware of the surrounding environment when sharing keys over the phone to avoid eavesdropping.

6.4.4 Benefits

Personalized Security: Involving users directly in the key exchange process, increases the sense of security and participation by doing it themselves.
Flexibility: Users can choose the method of key exchange that they are most comfortable with, phone calls or emails.
Independence from the App: This method does not rely on the app for key exchange, reducing the risk of key compromise through the app itself.

6.4.5 Summary

Our key exchange method prioritizes security and user involvement. By using phone calls and emails, we provide our users with a straightforward and secure way to initiate encrypted conversations.

6.4.6 Future： Security and Convenience

We want to improve users' security and experience, and we are exploring advanced methods for key exchange. These plans are designed to enhance security.

6.4.6.1 End-to-end Encrypted Email Integration

We plan to utilize end-to-end encrypted email in our key exchange. This approach allows users to securely exchange encryption keys via emails that are encrypted from sender to receiver. This method ensures that even if the email is intercepted, the contents remain indecipherable to unauthorized parties. If we can add this functionality directly within our chat application or use it as an external function. Users can securely do key exchanges.

6.4.6.2 Near Field Communication (NFC) Key Exchange

We plan to add NFC technology for key exchange. It's a fast and convenient method for users to share encryption keys securely. A secure NFC connection can be established by simply bringing

their devices into close distance, enabling the transfer of encryption keys. This method is advantageous for in-person key exchanges, as it minimizes the risk of interception by third parties. This will make the key exchange a straightforward and seamless process.

6.4.6.3 Biometrically Authenticated Key Exchange

We plan to add Biometric authentication as an extra layer of security to the key exchange process. By requiring users to verify their identity using unique biological attributes such as fingerprints, the risk of unauthorized individuals gaining access to encryption keys is significantly reduced. This method ensures that key exchanges occur only between verified users, providing a high level of trust and security.

6.4.6.4 Encrypted Chatbot for Key Exchange

We are planning to add an AI-powered, encrypted chatbot to our application. This chatbot guides users through a secure and automated key exchange process, simplifying the user experience. The AI chatbot can manage key validations, offer instructions, and assist with troubleshooting, making the key exchange process more accessible, especially for less tech-skill users. This way simplifies the key exchange process, making it more efficient and user-friendly.

7 CONCLUSIONS

Our project, dedicated to creating an encrypted communication system for group users, has made significant strides, achieving numerous milestones. We successfully implemented a sophisticated encryption communication method, ensuring secure and private group interactions. Each group uses a shared secret key and a unique channel name, coupled with a dynamically generated key, enhancing message security.

The front-end development has been robust, facilitating encrypted chat message transmissions and allowing users to access chat histories securely. We focused on user experience, developing a UI that is both functional and visually appealing, with features like Edit and Settings buttons for added utility.

On the back end, we established a resilient infrastructure using Python, capable of handling high request rates and interfacing with a MySQL database for secure chat record storage. We chose AWS for hosting, considering its reliability and security standards.

Moreover, we took significant steps in operational maintenance, implementing a Crontab job for regular data management, and laying down a solid database structure for efficient data handling.

However, the journey doesn't end here. We've identified key areas for further development, such as enhanced stress testing, threat modeling, and exploring alternative key sharing

methods. These initiatives aim to bolster our system's robustness and security, ensuring it remains reliable and secure in a constantly evolving digital landscape.

This project stands as a testament to our commitment to secure communication in the digital age. Through innovative technical solutions, rigorous testing, and continuous improvement, we strive to provide users with a safe and seamless communication experience, addressing the growing concerns around data privacy and cyber threats.

8 REFERENCES

[1] H. S. Husin, “Preventing data leakage by securing chat session with randomized session ID”, Int. j. commun. netw. inf. secur., vol. 13, no. 3, Apr. 2022.

[2] M. Alatawi and N. Saxena, “Sok: An analysis of end-to-end encryption and authentication ceremonies in secure messaging systems,” *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 187–201, Jun. 2023. doi:10.1145/3558482.3581773

[3] N. Tyagi, I. Miers, and T. Ristenpart, “Traceback for end-to-end encrypted messaging,” *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 413–430, Nov. 2019. doi:10.1145/3319535.3354243

[4] R. E. Endeley, “End-to-end encryption in messaging services and national security—case of WhatsApp messenger,” *Journal of Information Security*, vol. 09, no. 01, pp. 95–99, 2018. doi:10.4236/jis.2018.91008

[5] A. Candra, Y. Kurniawan, and K.-H. Rhee, “Security analysis testing for secure instant messaging in Android with study case: Telegram,” *2016 6th International Conference on System Engineering and Technology (ICSET)*, pp. 92–96, 2016. doi:10.1109/icsengt.2016.78496

[6] P. Agarwal *et al.*, “Jettisoning junk messaging in the era of end-to-end encryption: A case study of whatsapp,” *Proceedings of the ACM Web Conference 2022*, pp. 2582–2951, Apr. 2022. doi:10.1145/3485447.351213030

[7] P. P. G. Neogi, “A dive into WhatsApp’s end-to-end encryption,” arXiv.org, Sep 2022, arXiv preprint arXiv:2209.11198.

[8] M. Schliep and N. Hopper, “End-to-end secure mobile group messaging with conversation integrity and deniability,” *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, 2019. doi:10.1145/3338498.3358644

9 APPENDICES

Front End: https://github.com/liziwei01/capstone
Back End: https://github.com/catbeat/capstone-backend