A Lightweight Approach to End-to-End Encrypted Messaging & File Sharing

SHAH SUYAEIB, RYAN KOEPKE, and YEHUDA ROTHENBERG, University of Maryland, College Park

In this project, we will develop a chat system designed for Command Line Interface (CLI) environments. While graphical interfaces are more dominant nowadays, there are niche environments, particularly in low-resource settings, where CLI tools are prevalent and essential. However, these tools often compromise on security. Our solution presents a centralized server model that doubles as a Key Distribution Center (KDC), ensuring encrypted communication between clients. Through this model, we ensure end-to-end encryption of messages using AES in Cipher Block Chaining (CBC) mode, coupled with RSA for signature verification and non-repudiation. Furthermore, the chat system emphasizes message and file transfer integrity, utilizing SHA-256 hashing mechanisms. With these tools and techniques, we aim to offer secure communication in CLI settings and combine simplicity with top-tier security protocols.

1 OBJECTIVE

In today's digital age, there are many forms of communication such as chat rooms and messaging often requiring advanced technology and power to carry its messages. Chat rooms serve as dynamic spaces for individuals and groups to exchange information, share resources, and engage in conversations. They are necessary for both personal and professional communication. However, with this convenience comes the challenge of ensuring the files being shared are safeguarded against unauthorized access, data corruption, and breaches of confidentiality. These challenges primarily revolve around three key aspects: data integrity, confidentiality, and authentication. This works well for a developed society where power and resources are not an issue; however, for places that are such as third-world countries or during wars, these communication devices can be unfeasible to distribute to large populations. We need a way to send secure data over a lightweight system. A command line interface (CLI) is a good starting point for all of this. Since CLIs are all text-based interactions, we don't need the resource-heavy GUI and can rather focus on the core task which is to send secure messages.

The challenge with this new idea is not to only create a lightweight communication method but to also ensure it's secure and reliable while maintaining its lightweight nature. Current CLI chat tools, while being resource-efficient, often lack robust security measures. This shortfall is dangerous, especially when considering the potential deployment of such tools in critical scenarios such as emergency response, military communication, or in remote areas with limited resources. The absence of security features such as end-to-end encryption, cryptographic hashing, and digital signatures exposes these tools to numerous cyber threats including data tampering, man-in-the-middle attacks, and replay attacks.

Addressing these security challenges is needed because in third-world countries or war-torn regions, where resource constraints are a daily issue, using a lightweight Arduino-based system can help the masses stay informed and communicate with each other. This not only adheres to the resource constraints but opens a reliable channel of communication in adverse circumstances, facilitating information exchange, coordination, and even potentially saving lives during

Authors' address: Shah Suyaeib; Ryan Koepke; Yehuda Rothenberg, University of Maryland, College Park.

critical situations.

The first challenge is to develop a lightweight CLI chat tool that implements security features. The tool will utilize end-to-end encryption to ensure confidentiality, cryptographic hashing, and digital signatures to ensure the integrity and authenticity of messages and files transferred. Secondly, the project aims to demonstrate non-repudiation and authentication of messages, ensuring that the sender of a message can be identified and cannot deny sending the message. Thirdly, the secure channels for file transfers and the creation of compartmentalized room-based chats are envisioned to foster organized and secure communication. Lastly, while the Python implementation may exhibit slower performance, it serves as a crucial proof of concept, showcasing the core security features essential for reliable and secure communication in a lightweight setup. In the future which is out of scope due to the time frame, we would need to implement this setup inside an Arduino system to facilitate communication between two separate devices.

2 BACKGROUND & MOTIVATION

Most chat applications do not emphasize on secure file transfer and Instead are designed to offer a lightweight and user-friendly chat experience. They are more concerned with appealing to a broad range of consumers and designed to be used by anyone regardless of technical expertise. Secure file transfer requires robust security encryption methods and might introduce complexity and slow down the user experience. Users in a chat room can exchange a wide range of information that might include very sensitive and confidential data. Users place their trust in these communication platforms to keep their exchanges private and secure. Recently there has been an increase in data breaches and privacy violations across almost all online platforms that really underscores the importance of security when exchanging information. The entire success of the chat program relies on the users feeling confident that their communication is safe and secure. Without a secure platform users would lose trust as their transfers could be intercepted and tampered with. To maintain trust, multiple security measures will be implemented, using digital signatures, cryptographic hash functions, and file encryption.

One key aspect to providing user trust will be to ensure integrity of the files and safeguard them from any form of alteration or unauthorized manipulation while they are in transit from one user to another. This will be done using SHA-256 function and digital signatures. With SHA-256 function, both sender and receiver of a file can calculate the unique hash value output. With the output value of this function users compare their hash value, and if they match, the file is intact and complete. However, even if one bit change in the file content is changed, it will produce distinct hash value output and will be an indicator of potential tampering of the file.

Also, digital signatures will enhance the security of the file transfers. A KDC will issue keys to the users. When a user wants to send a file, they will be digitally signed using the sender's private key. The recipient will use the sender's public key to validate the signatures. If the signature is valid then it will be proof that the file integrity is ensured. This prevents tampering since the digital signature will be invalidated if the file has been modified in any way.

Encryption is another key aspect of user trust that the chat program will implement. By employing strong encryption mechanisms, the program ensures that files transferred within the chat room remain confidential, private, and safeguarded against unauthorized access. It's important that the files are encrypted so that only the intended user can decrypt and access the file. This confidentiality will be achieved using AES encryption. Both sender and receiver will generate the same AES key independently. When one user wants to send a file securely, the AES algorithm will be applied to the file contents using the shared AES key. When the file is transmitted over the chat room network the encrypted file will remain secure since an attacker won't have the AES key for decryption. Only the recipient will be able to decrypt the file, ensuring its confidentiality. Authentication will help users trust the program since every user will be verified and they will know exactly who they are sending data to. Ensuring that both parties are who they claim to be is vital for security. username/password authentication will prevent unauthorized users from sending or receiving files. Users will be required to create a unique username and password that will be stored in a database using hashing techniques. Strong password requirements will be implemented including minimum length, complexity requirements will provide added security.

In addition to the security measures discussed, the chat program will also implement data logging. Data logs are a crucial tool for accountability and tracking user activities within the platform. Logs will record user interactions, file transfers, and authentication attempts. These logs are instrumental in identifying and addressing any suspicious or unauthorized activities, ensuring that the chat room maintains a secure and trustworthy environment. Data logs will be carefully managed to strike a balance between accountability and user privacy.

3 PROPOSED TECHNIQUE

As previously mentioned, our implementation will be in Python, using the relevant libraries and a CLI user experience. An ideal final product would in a low-level language for speed and minimal size, but we'd like to show a proof of concept in Python given the time constraints. Naturally, our technique may also change as we face challenges in implementation.

3.1 Basic Chat Outline

The application will consist of a central server program and the user client programs. Users can create password-protected rooms in which all room members will received room messages, or directly message another user. These messages can also consist of files, at which point the file transfer protocol will be initiated. Logging will be done on the server to track attempts and seek out authentication or message traffic anomalies, but no chat records will be stored. Any failures in key generation or message or file sending/verification will result in error messages being sent to the appropriate parties. Progress bars for file sending and receiving are being considered as an additional feature.

3.2 User Authentication & Key Generation

User logins will be protected by username and password, with the salted hashes of the login info being stored on the server and compared to attempts. At registration, each user will locally generate an RSA public/private key pair. The user will store the private key locally and send the public key to the server. For each chat session, whether via room generation or direct message, a shared AES key will be generated. Whenever a user logs off, their keys are deleted, and must be regenerated.

3.3 Chat Session Setup & Message Transmission

To initiate a direct message chat, User A will generate an AES session key, store this key locally, and send User B the AES key encrypted with User B's public RSA key. User B will decrypt the session key using their private RSA key. Chat rooms will work similarly, with authenticated users being sent the encrypted room AES session key after a successful join.

Every chat message will be encrypted using the chat session AES key. For message integrity and non-repudiation, each message is also signed using the sender's RSA private key. This signature can be verified by others using the sender's public key. Messages will be rate limited to an as-of-yet undecided limit.

3.4 File Transmission Protocol

When a user decides to send a file, the client checks the file's size. If the file's too large, it's split into chunks. The chunks are encrypted using CBC mode with the AES encryption key. Then the client sends a "File Transmission Request" to the server which includes the filename, the total number of chunks, the overall file size, and an encrypted hash of the complete file for integrity checks. The AES key, encrypted with the receiver's public RSA key, is also sent, with an RSA-signed hash of the encrypted file too to ensure non-repudiation.

Upon receiving this transmission request, the server forwards it to each recipient in the room. Each recipient then can choose to accept or decline the incoming file, with their decision being sent back to the server. Once the server gathers all decisions from the intended recipients, it pings the sender with a "Start Transmission" message. Acting on this, the sender's client begins sequentially dispatching the encrypted file chunks to the server. As each chunk arrives at the server, it is immediately relayed to all accepting recipients. To efficiently manage this, the server upholds a tracking table that logs the status of chunk receipt for each participant.

If a recipient doesn't acknowledge the receipt of a chunk within a certain timeframe, the server marks it as a "timeout" for that specific chunk and recipient. It'll still attempt to resend that chunk to the lagging recipient a pre-defined number of times before resorting to an error message.

The sender's client waits after sending a chunk until the server signals readiness to receive the next chunk. This strategy ensures that the transmission remains synchronized and the server isn't swamped with data. Once all chunks are securely transmitted and acknowledged, the sender's client sends out a "Transmission Complete" signal to the server. This is relayed to all recipients who reassemble the file, decrypt it and compute the hash to check file integrity. Non-repudiation is confirmed by verifying the RSA signature on the hash using the sender's public key. Feedback is sent to the server to let it know the status of the receiving clients.

4 EVALUATION PLAN

To evaluate our solution, we'll focus on functionality and usability. We'll stress test our client and server programs by sending various files and messages of varying length and complexity to multiple clients, ensuring that all components interact seamlessly. In addition, the integrity and security of our protocols will be tested by simulating attacks like message alterations and login spamming. We'll also monitor the rates of failed authentications and the efficiency of our encryption mechanisms.

Speed is an important metric in any program. So even though our prototype is developed in Python (which might not offer the performance of lower-level languages) it's crucial for us to measure the speed of our protocols and processes to gauge their efficiency. As such, we will time the various components involved in the messaging and file sharing tasks, as well as the overall processes.

To conclude, our evaluation approach aims to test the reliability and security of our chat system. We've left some flexibility for addressing obstacles as they arise. The goal is to illustrate the feasibility of an efficient and secure chat and file-transfer system that stands up to real-world challenges.