CSC4026Z

Encrypto

Network & Internetwork Security Practical

Created by

David Court

Bradley Culligan

Michael Scott

Introduction

Encrypto is a secure messaging application that facilitates end-to-end encryption of user messages sent in a private room for group communication. Messages are exchanged using a client-server architecture where the server also acts as the Certification Authority for connected users and ensures that messages are sent to the correct room. Each user holds a certificate that can be used to ensure authentication, and a corresponding private/public key pair that is used to provide end-to-end encryption.

A logged-in user can start a room with a custom room ID and password, and subsequent users can then join that room by supplying the correct details. When users join a room, users exchange certificates and authenticate each other. A PGP security model is then used to provide end-to-end encrypted messaging between the users in a room.

This report proceeds to document Encrypto's **Communications Implementation**, **Overall Design and Security Implementation**, and **Evidence of Testing**.

1. Communications Implementation

Communications process

Encrypto uses a client-server architecture for sending and receiving messages. The communication system is based on the TCP model where the server waits for a connection from a user, after which a dedicated socket and server thread are created to manage further communication. Concurrently, the user application starts two threads - UserRead which listens for messages from the server, and UserWrite which sends messages from the user to the server. The user initiates sending messages to the server via controls supplied on the GUI.

All messages have headers (annotated as ':X:') that are tokenized and used to control the receiver in some way - e.g. logout a user from the server, or update the user's GUI to display a text message. Receivers can then understand the structure of the message contents and react appropriately.

Users interact with a GUI and connect to the server under a chosen username when logging in. Once connected, users can start a chat room of their own or join an existing room and begin chatting. When starting or joining a room, both a room name (ID) and password must be supplied. The server will check whether the given room already exists and, in the case of joining, whether the password is correct. If there is any conflict in this process (e.g. trying to start a room with a name that already exists), the server will inform the user via a message with header :INVALID: and the user will be updated on the problem via its GUI. If not, the GUI will be updated to reflect the new room.

Evidently, Encrypto makes use of the MVC architecture. Our GUI is the view which represents the state of the user and room models. User actions are captured by the buttons available on the GUI which in turn send control messages to the server model and subsequently all user and room models where necessary. Once the models have been updated, they then update the user's view (e.g. new user in room is displayed on the GUI on the left panel).

A user may leave a room by joining or starting a new room, using the logout button, or closing the application. In such a case, a control message is broadcast to the current room, informing present users of the event and removing them from the room's broadcast list.

Key exchange process

We now focus our discussion on the key exchange processes that occur. Note before beginning our discussion that users have their own public/private key pairs and the server also has its own public/private key pair when the objects are initially created (via the Makefile). The asymmetric

encryption specification used is the RSA algorithm in ECB mode with PKCS1Padding. When a user first logs in, they will send a message to the server to initiate their own server thread for subsequent communications. The server, which also acts as the Certification Authority (CA), will generate a certificate for the user and sign the certificate with its own private key. The server sends the signed certificate and the CA's public key back to the initiating user. The CA's public key is important to authenticate signed certificates (as discussed below) and to allow encryption of a room password.

A logged in user can then initiate a group chat (by starting or joining a room as discussed above). When a user joins a room, all users in the room exchange certificates and use the CA's public key to authenticate the certificates received. The user stores the (updated) list of certificates for all connected users in the room and uses this to extract user public keys when necessary. When a user wants to send a message to the room, it uses the PGP model for end-to-end encryption of the message sent individually to each user. The sender will use the AES algorithm in CBC mode with PKCS5 padding to generate a shared key for encrypting the text message. The message will be encrypted with this shared key and the shared key will be encrypted with the public key of the receiver. The message will thus be received as ciphertext and the key to decrypt this ciphertext can be obtained by the receiver by using their private key to decrypt the encrypted session key. The, now, decrypted session key can be used to decrypt the ciphertext message and displayed to the receiver on their GUI. We can be certain that the keys are valid by the authentication mechanism of PGP, as discussed in the next section.

2. Overall Design and Security Implementation

This section details the central message sending process together with some additional design choices, with a focus on explaining the security implementation. To ensure message integrity, authentication, and confidentiality, messages in Encrypto are exchanged according to the PGP security protocol. The process of sending a message from one user to another, whilst ensuring the above security concerns, is shown in the diagram below and explained in detail thereafter.

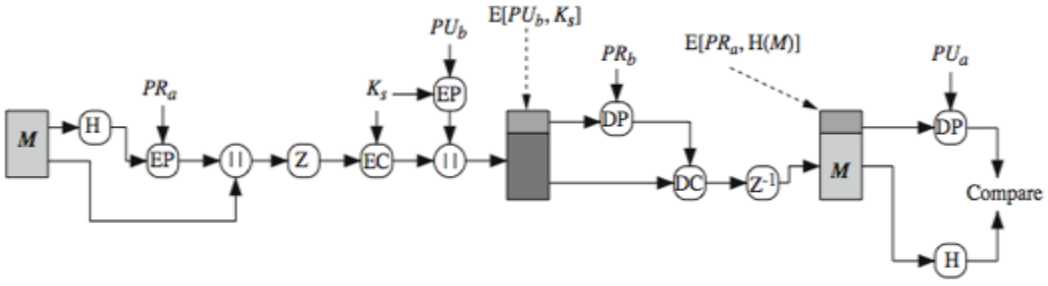

Figure 1: PGP security protocol

First, the plain text of the message is hashed and signed (encrypted) with the private key of the sending user. Our implementation uses the SHA-512 hashing algorithm of the MessageDigest class in the java security library. This one way function is reused later in the protocol to hash the (decrypted) plain text received and compare it to the hash of the plain text sent. Because the function is deterministic, any inconsistencies in the hashes calculated will signal that message **integrity** has not been upheld. Such messages are discarded and are not allowed to update the GUI. The integrity failure is reported in the debug statements.

Signing the hash with the private key of the sender ensures that when a receiver decrypts the hash (using the sender's public key) and compares it to their own hash (as above), they can **authenticate** that the message did indeed come from the sender. The signing and decrypting of hashes are done using the Cipher class functions from the javax crypto library.

Next, the signed hash and the message plain text are concatenated and **compressed** using a gzip function. This reduces overall message size and potentially the number of TCP packets required. This is a performance optimization design choice.

To further improve performance, **shared keys** are used instead of private/public keys for overall message encryption given that their encryption/decryption speed is faster. A one-time-use shared key is generated by the sender per message and used to encrypt the compressed payload. Our implementation uses the AES algorithm in CBC mode with PKCS5 padding. This encryption ensures the payload is kept **confidential** between holders of the shared key. The shared key is encrypted with the public key of the receiver and concatenated with the ciphertext that was generated with this shared key. This ensures that only the intended receiver with the corresponding private key can decrypt the concatenated message to obtain the shared key and the ciphertext generated from the shared key. This ensures **confidentiality** and that anyone couldn't have access to modify the internal message, thus providing **integrity**.

The message is now ready to be sent across any network securely. Using their public key ring and a list of present users in a room, a user then follows the process above for each public key and sends these messages to the server. The server identifies to whom each message is addressed using the unencrypted header fields and forwards the message appropriately. When the receiver receives the message, they immediately decrypt the shared key with their private key, decrypt the payload with the shared key, decompress the message and generate and compare hashes as described above which checks **authentication** and **integrity**. The message has been securely delivered and is displayed on the user's GUI.

It should be noted that our implementation does expose header information (metadata) to potential 3rd party listeners. However, for the scope of the project, this is not considered a critical security concern because only control commands, room IDs, and public keys are included in this metadata.

A separate security concern that is addressed is the visibility of passwords. Passwords are required when joining or starting a room and should never be communicated in plain text over a network. Our implementation reuses the SHA-512 hashing algorithm to store and compare user provided passwords, as well as asymmetric encryption using the server's public key for sending encrypted room passwords to the server, and its private key for decrypting password attempts to compare and verify password hashes. Additionally, passwords are masked on the GUI using password text fields.

3. Evidence of testing

The GUI for Encrypto allowed us to use the terminal for outputting messages that were happening in the background. This space was used for printing debug statements and testing that the code was running correctly. Further, the use of a GUI allowed us to control the user workflow and limit functionality to when it was required. Thus, with the assistance of some control restrictions, we are able to make the assumption that our system is robust and functions cannot be called incorrectly by the user to break the application.

The terminal space outputted messages that updated the underlying models. In particular, the server model would use its terminal space to output messages received from users (such as if a user successfully logs in, then the server outputs that a new user has connected). The server could also update its own model because it is the CA and, thus, the terminal space would also output messages indicating the certificates that were generated by itself.

The user model would output messages that it received from the server. This would include the user's certificate generated by the CA, messages indicating if a room operation (join/start) was valid, and text messages received from users in the room. Of particular importance is the password and text messages received which would be encrypted. These are end-to-end encrypted (user-to-server for room passwords and user-to-user for messages). Thus, messages were also outputted to the user's terminal space as they followed along the PGP model path - showing the encrypted message, session key, and hashed message, along with the decrypted message and session key.

All terminal output for debug statements are implemented via 'inform' methods, which the server and user both have access to. Finally, Encrypto was manually tested under various use cases to ensure the working order of the application. Please see the README for application usage details.