CloseCircle
CISC 468 Final Report

Uroosa Imtiaz (20216964)
Daniel Fichtinger (20148196)

ABSTRACT

'CloseCircle' is a secure, password-authenticated peer-to-peer messaging application designed for local network use, implemented in Python and Go. It employs the multicast Domain Name Service (mDNS) for peer discovery and Transmission Control Protocol (TCP) for establishing connections. For secure message encryption, the app uses the Diffie-Hellman Ephemeral Key Exchange (DH-EKE) protocol to generate session keys. A user's private information is protected by an attack-resistant password-derived security scheme inspired by the Bitwarden Security Whitepaper.

1 INTRODUCTION

The growing need for secure peer-to-peer messaging applications is fueled by public concerns over the lack of end-to-end security in popular messaging platforms and the threat of mass surveillance by intelligence agencies. However, many new messaging apps contain preventable security vulnerabilities, rely on deceptive advertising, or fail to offer true end-to-end encryption.[15]

This report introduces our implementation of two interoperable, peer-to-peer messaging clients in Python and Go. A 'CloseCircle' client requires zero configuration for local network chatting and offers robust security features within the app and for messages exchanged between contacts.

2 NETWORKING

CloseCircle uses Zero-Configuration networking for automatic service discovery on local networks. Upon login, the user's name and service type `_closecircle._tcp` is registered with their device's local Internet Protocol (IP) address, determined by pinging Google's public DNS server (`8.8.8.8`).

TCP is a transport-layer protocol that guarantees reliable message delivery. Once the service address of a peer is discovered, the app establishes connection requests and conducts subsequent messaging using TCP. The app listens for incoming connections on port `3000`, supporting a single synchronous chat session at a time. When the app exits, all network operations are gracefully shutdown and the service is unregistered to remove the user from the network, preventing outdated entries.

3 MESSAGING

Closecircle implements a secure handshake protocol using password-authenticated Elliptic Curve Diffie-Hellman Ephemeral Key Exchange (ECDH-EKE) and AES encryption. Before sending a connection request, a user must add a contact with their shared password agreed upon through a secure, out-of-band channel, such as a face-to-face meeting. The authentication process is repeated each time a connection request is initiated, so each party can derive a session key used to encrypt further communication. To establish a chat session, both peers must be online in the same local network, using the correct password.

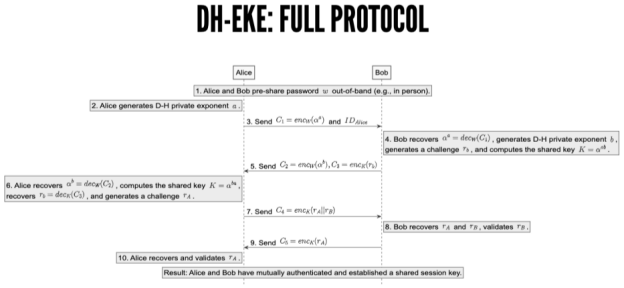

Figure 1: Figure 1: Sequence diagram illustrating the complete Diffie-Hellman Encrypted Key Exchange protocol.[5]

3.1 Key Derivation

Once a contact is selected, the system retrieves and stretches the shared password for the requested peer using a Key Derivation Function (KDF) called Scrypt to 256 bits. Scrypt is specifically chosen for its resistance against brute-force attacks. Let's denote this 256-bit key as w.

The system utilizes elliptic curve cryptography to generate a new public-private key pair on the Curve P-256. The peers exchange public keys to derive a shared session key.

3.2 Mutual Authentication

The connection process uses a DH-EKE handshake. The initiating peer encrypts their public key with w and sends it to the responding peer. The responding peer decrypts it with w and uses it to generate the session key, k. It also generates a random 8 byte challenge and encrypts it with k, and encrypts its public key with w. It sends both of these values to the initiating peer.

The initiating peer decrypts the public key and derives k. It then is able to decrypt the challenge. It generates its own challenge and appends it to the received challenge, encrypts this message with k and sends it to the other peer. The receiver decrypts the message, verifies their own challenge, and replies with the initiating peer's challenge. Finally, the initiator verifies the challenge. Now, both peers have mutually authenticated.

3.3 Integrity

We chose to use a hash function for verifying message integrity. We decided to use SHA-256, which was designed by the NSA and is known to be secure. After the message is encrypted, the ciphertext is given as input to a SHA-256 function to obtain a 32 byte message

digest. This hash is included with the message when transmission occurs. Upon receipt, the receiver hashes the ciphertext with the same hash function, and compares it with the message digest included with the message. If the hashes match, the integrity of the message has been verified.

3.4 Encryption

All messages are encrypted with AES-256-CBC. A symmetric cipher was chosen for its efficiency in operating on long strings of bytes. As such, users do not need to be limited on the length of the messages they wish to send. Furthermore, this enables the efficient encryption of files, which are typically larger than the average string message. A 256 bit (or 32 byte) key was chosen for its high security benefit at the minor cost of slightly increasing the size of message transmissions.

AES was selected due to its security being well known, with the United States government using if for all secret and top secret communications.[14] Finally, Cipher Block Chain mode was chosen due to its use of a unique, random initialization vector (IV) with each encryption operation. Identical plaintext blocks will encrypt to different ciphertext blocks, preventing an attacker from employing pattern recognition techniques to learn information about the plaintext.

3.5 Authentication Failure

The authentication immediately fails if one side hasn't added the other as a contact or if the shared password is incorrect, and timeout if no response is received. This failure occurs because they cannot decrypt each other's public keys with the wrong password-derived key, resulting in a PKCS7 padding error.

Furthermore, if both sides successfully a session key but fail to send the correct challenge, the handshake also fails, which helps prevent replay attacks. Although there is no limit on the number of contact requests from a specific peer, the immediate closure of the connection mitigates Denial of Service Attacks.

3.6 Attack Resilience

3.6.1 Perfect Forward Secrecy (PFS). In each chat session, a random public-private key pair and session key are derived using ECDH-EKE. A session key can only decrypt messages from its own session and cannot be used to derive or decrypt other keys or messages.

If w is compromised and the public keys being exchanged can be decrypted, it will also not allow a bad actor to derive the session key. Thus, while the compromise of w defeats authentication, it does not affect PFS, as it will not allow an eavesdropping attacker to learn any session keys.

3.6.2 Man-In-The-Middle Attacks. A strong password must be shared in a secure, out-of-band channel to prevent Man-In-The-Middle Attacks. An attacker would need to know w in order to execute such an attack; otherwise, the attack will be unsuccessful as the DH-EKE handshake would immediately fail.

3.6.3 Replay Attacks. These attacks are prevented by the challenge-response element of DH-EKE. Because both peers' challenges are randomly generated per session, an attacker cannot replay a successful handshake's packets to attack a future session because the target will be expecting a different challenge to be returned to them.

Furthermore, replayed messages will not be accepted in future sessions as the target will be using a new session key. Resilience could improved by including encrypted timestamps per message to guard against same-session replay attacks.

3.7 Formatting

A message is encoded in JSON with three fields: `data`, `iv`, and `hash`.

- `data`: message or file content.
- `iv`: initialization vector used for encryption.
- `hash`: hash of the ciphertext.

A packet object is a list of messages used to send either a single message or two messages simultaneously, as in the second step of key exchange or when sending a file.

4 USER DATA

Each user's data is stored in their private vault - their profile data, along with their contacts information and messages. The vault is encrypted with AES-256-CBC. The following section describes a security scheme inspired by the Bitwarden Whitepaper [6].

4.1 Registration

If there is no existing user profile, the app prompts the user to choose a username and password. It generates a random salt and feeds it into PBKDF2 to create a 256-bit Master Key. It also generates a random 256-bit key and IV, with the key encrypted using the Master Key, which is not saved.

The Master Key and password are then passed through PBKDF2 again to produce the password hash, which is stored along with the salt, IV, and encrypted key. This information is later used for password verification and encrypting or decrypting the vault.

4.2 Login

If the app detects a user profile, the user is prompted to log in. It loads the user's profile and processes the password attempt through PBKDF2 with the salt to generate the Master Key. This Master Key, along with the password attempt, is then passed through PBKDF2 again to create a candidate for the password hash. The app compares this candidate with the actual password hash, and if they match, it logs the user in.

The encrypted key is then decrypted using the Master Key, and all user files are decrypted with AES-256-CBC using the decrypted key and IV. If the hash of the encrypted files doesn't match the stored hash, or if padding errors are detected, the login permanently fails. This ensures that if the user's password hash or data is tampered with, the data cannot be decrypted.

4.3 Logout

In the event of a keyboard interrupt or the user's exit from the menu, the user's messages and contacts are once again encrypted using AES-256-CBC with the key and IV. The key is then encrypted with the Master Key and stored in the user's profile. The user's profile, encrypted messages, and contacts are hashed using SHA-256 and stored in a hash file, which is used during login.

If the encryption step fails, the user will lose their data since future login attempts will fail due to errors decrypting the vault. In

that case, the user can copy their unencrypted data and reload it into the directory mid-session after deleting their user profile and creating a new one to ensure it is not lost.

5 USER INTERFACE

Closecircle features a Terminal User Interface (TUI). The application must be launched by the user in their preferred terminal emulator. This maximizes compatibility across platforms because a separate user interface implementation using native platform libraries is not needed. It also keeps the program lightweight.

6 IMPLEMENTATION

Go (Golang) and Python 3 were chosen as the programming languages for the two clients due to their simple syntax and ease of use. Python is familiar to the authors, and Go is known for being easy to learn for developers proficient in Python and C-like data types.

6.1 Python Client

The Python client follows an object-oriented programming (OOP) approach, organizing classes into four packages. Its source code is located in the "src" directory, with the `App` class in `src/App.py` as the entry point.

Table 1: Overview of Packages and Classes in Python Client

Package	Classes
`auth`	`Register`, `Login`
`chat`	`ZeroconfService`, `Connection`, `Handshake`, `ChatUI`
`user`	`User`, `ContactManager`, `MessageManager`
`util`	`FileEncryption`

6.1.1 Networking. The `ZeroconfService` class imports the `zeroconf` library [2] to initialize a `Zeroconf` object, managing server startup, user registration, service discovery, and shutdown. It uses `threading` to update the peer list, initializing a `ServiceBrowser` object outside the update thread to avoid thread limit issues. `Connection` class employs `socket` and `threading` for connection listening.

6.1.2 Messaging. The `Handshake` class contains nested `Message` and `Packet` classes. Each `Message` object is initialized with byte fields `data` and `iv`, with the `hash` generated if none is provided. A `Packet` represents a collection of `Message` objects, that can be converted into a JSON string using `json.dumps` or vice versa using `json.loads`. The `MessageManager` class loads and stores a user's messages.

6.1.3 Cryptography. The `Register` and `Login` classes import the `cryptography` library [4] for user authentication, initialization, or file checksum verification for the vault. The `ContactManager` class uses `cryptography` to stretch a password key with scrypt, while the `Handshake` class uses it to generate elliptic curve public keys. Both the `Handshake` and `FileEncryption` classes use `cryptography` for AES-256-CBC encryption with PKCS7 padding and SHA-256 hashing.

6.1.4 User Interface. The `App`, `Login` and `Register` classes use the `click` library [3] for user input and verification, while the `Connection`, `App` and `ChatUI` also use the `prompt-toolkit` library [1] for its shortcut dialog pop-ups, and to create a full window chat session.

6.2 Go Client

The Go client follows a modular approach, with the `peer` package implementing all of the main application code using the networking structures and functions defined in `networking`. `main` implements the user interface and acts as driver code but does not handle any core application logic.

Table 2: Overview of Packages and their functionalities in Go Client

Package	Functionality
`main`	`UI`, `Driver Code`
`peer`	`Peer Discovery`, `Crypto`, `Messaging`
`networking`	`MDNS`, `TCP`, `Connection Handling`

6.2.1 Networking. The `peer` package's `AppState` struct includes a `NetworkingState` struct from the `networking` package, used to manage the start and shutdown of network servers. Within the `networking` package, the `mdns_server.go` file imports the `grandcat/zeroconf`[13] package. This file defines the `MdnsServer` struct and provides functions to initialize, start, and stop the `zeroconf.Server` and uses Go's goroutines for asynchronous peer discovery. Additionally, the `networking/tcp_server.go` file defined the `TcpServer` struct manages connections using the `net` package, goroutines, and channels.

6.2.2 Messaging. In the `Message` struct, the `Data`, `IV`, and `Hash` fields are of type string, which will store base64-encoded data. The `NewMessage` function takes byte slices (`data`, `iv`, and `hash`) and creates a new `Message` instance with these fields base64 encoded.

The `encodePacket` function encodes a slice of `Message` structs into a JSON byte array using the `json.Marshal` function. The `decodePacket` function decodes a JSON byte array into a slice of `Message` structs using the `json.Unmarshal` function. It iterates through each message, decoding the base64-encoded fields (`Data`, `IV`, and `Hash`) back into byte slices and creating new `Message` instances with these decoded values using the `NewMessage` function, and returns them.

6.2.3 Cryptography. The `AesEncrypt` and `AesDecrypt` functions use `crypto/aes` and `crypto/cipher` to implement AES-256-CBC, using `crypto/rand` for IV and challenge generation, and `zenazn/pkcs7pad`[11] for the PKCS#7 padding scheme. These functions also provide integrity checking, using `crypto/sha256` for hashing; the encryption function returns the hash value alongside the ciphertext and IV, while the decryption function also requires a hash argument, returning an error in the case of a hash mismatch.

Elliptic Curve Diffie-Hellman is implemented using `crypto/ecdh` and the P256 curve. Finally, the

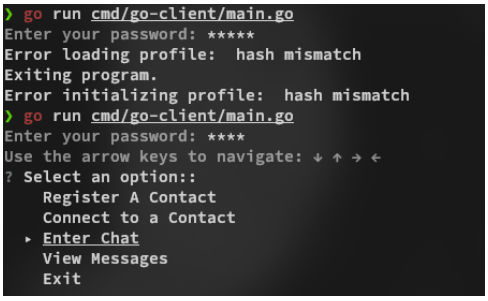

Figure 2: Demonstration of login and main menu interface in the Go client.

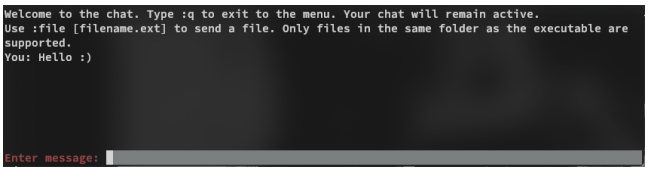

Figure 3: Demonstration of chat terminal user interface in the Go client.

`x/crypto/scrypt`[10] library provides the KDF used in DH-EKE handshake, while `x/crypto/pbkdf2`[9] is used to implement user profile security.

6.2.4 User Interface. The main menu will loop, prompting the user for a selection, until the user performs either a keyboard interrupt or selects the "Exit" option. To collect user input, the `manifoldco/promptui`[7] package is used to display both text input prompts and menu selection prompts. Password inputs are masked with asterisks. Selections can be made by using the arrow keys to navigate the options, and pressing enter to select. The chat interface covers the entire terminal window when open. This interface is implemented with `rivo/tview`[12] and `gdamore/tcell`[8] packages. The user can also input commands to send a file or quit the chat.

REFERENCES

[1] [n. d.]. Python Prompt Toolkit 3.0 - prompt_toolkit 3.0.43 documentation. https://python-prompt-toolkit.readthedocs.io/en/master/
[2] [n. d.]. Python-Zeroconf. https://github.com/python-zeroconf/python-zeroconf
[3] [n. d.]. Welcome to Click - Click Documentation (8.1.x). https://click.palletsprojects.com/en/8.1.x/
[4] [n. d.]. Welcome to pyca/cryptography - Cryptography 43.0.0.dev1 documentation. https://cryptography.io/en/latest/
[5] Furkan Alaca. 2024. Lecture 22 - CISC 468.
[6] Bitwarden. [n. d.]. Bitwarden Security Whitepaper. https://bitwarden.com/help/bitwarden-security-white-paper/
[7] James Bowes. 2021. promptui package - github.com/manifoldco/promptui. https://pkg.go.dev/github.com/manifoldco/promptui
[8] Garrett D'Amore. 2024. tcell package - github.com/gdamore/tcell/v2. https://pkg.go.dev/github.com/gdamore/tcell/v2
[9] Google. 2024. pbkdf2 package - golang.org/x/crypto/pbkdf2. https://pkg.go.dev/golang.org/x/crypto/pbkdf2
[10] Google. 2024. scrypt package - golang.org/x/crypto/scrypt. https://pkg.go.dev/golang.org/x/crypto/scrypt
[11] Carl Jackson. 2017. Pkcs7pad Package - Github.com/zenazn/pkcs7pad. https://pkg.go.dev/github.com/zenazn/pkcs7pad
[12] rivo. 2024. tview package - github.com/rivo/tview. https://pkg.go.dev/github.com/rivo/tview
[13] Stefan S. 2022. Zeroconf Package - Github.com/grandcat/zeroconf. https://pkg.go.dev/github.com/grandcat/zeroconf
[14] Benjamin Scott. 2020. Military-Grade Encryption Explained. https://nordpass.com/blog/military-grade-encryption-explained/
[15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. 2015. Sok: Secure messaging. *2015 IEEE Symposium on Security and Privacy* (May 2015). https://doi.org/10.1109/sp.2015.22