**Members:** Michael Crabb and Noah Erdachew

**Introduction**

The provided code is a complete, cross-platform secure chat application implemented in C++ that utilizes OpenSSL for cryptographic operations and supports both basic AES-CBC encryption and an advanced nested dual-AES cascade encryption mode. The application offers peer-to-peer encrypted communication using a shared passphrase or a Diffie-Hellman (DH) key exchange fallback mechanism. This report analyzes the architecture, major components, cryptographic design decisions, and the overall security posture of the application.

**1. Code Structure Overview**

The source code is organized into multiple functional sections:

* **Header Inclusion and Constants Definition:** Includes system and OpenSSL headers for networking and cryptography. Constants such as key length, IV length, and PBKDF2 iteration count are defined.
* **Utility Functions:** Contains helper functions for hexadecimal output and length-prefixed data transmission (send\_data, recv\_data).
* **Key Derivation:** Uses PBKDF2 with HMAC-SHA1 to derive a 128-bit AES key from a user-supplied passphrase.
* **AES-CBC Encryption/Decryption:** Handles symmetric encryption using the OpenSSL EVP interface.
* **Nested Dual-AES Cascade:** Implements two-layered AES-CBC encryption for enhanced security.
* **TCP Communication Setup:** Establishes client/server communication over sockets.
* **Diffie-Hellman Key Exchange:** Provides a fallback mechanism to derive a shared session key.
* **Main Chat Loop:** Manages I/O, encryption/decryption, key rotation, and secure message transmission.

**2. Networking Design**

The secure chat application is designed to support two communication modes: server and client. This is implemented using standard TCP sockets, providing a reliable, stream-oriented protocol for message exchange. TCP ensures ordered delivery, retransmission of lost packets, and flow control, making it well-suited for the needs of a chat application where message integrity and delivery order are critical.

The application uses conditional compilation to maintain platform compatibility between Windows and UNIX-like systems. On Windows, it includes winsock2.h and uses WSAStartup to initialize the networking subsystem. On UNIX-like systems, it uses the standard Berkeley sockets API and includes headers like sys/socket.h, arpa/inet.h, and unistd.h.

There are two key functions abstracting the communication roles:

* create\_server() is used when the application is run in server mode. It:
  + Creates a socket using socket(AF\_INET, SOCK\_STREAM, 0).
  + Binds the socket to an IP address and port using bind().
  + Listens for incoming connections using listen().
  + Accepts the first incoming connection using accept() and returns the client socket descriptor.
  + Logs status messages to the console to indicate server readiness and connection acceptance.
* create\_client() is used when the application runs in client mode. It:
  + Creates a socket in the same way.
  + Resolves the server IP and port.
  + Attempts to connect using connect().
  + If successful, it logs that the client has connected.

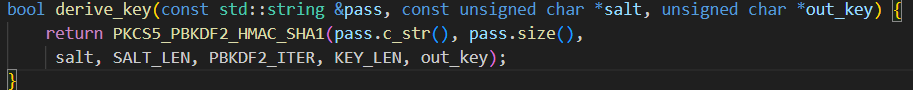
Both functions abstract the lower-level socket operations and return an active socket descriptor that is later used by the main chat loop for data transmission. These functions operate in blocking mode, which means function calls like accept(), recv(), and send() will block until the operation completes or fails. This approach simplifies the logic but can be enhanced in future revisions with asynchronous or non-blocking sockets for scalability.

By supporting both server and client roles, the application provides a flexible architecture for peer-to-peer communication. The dual-mode operation is determined at runtime by command-line arguments, offering a user-friendly mechanism to designate roles without code modification. This modularity is a hallmark of robust socket programming and makes the application extensible for future improvements such as multi-client support, TLS integration, or multiplexed sessions.

The inclusion of a cross-platform socket abstraction (CLOSESOCK) ensures proper resource cleanup regardless of the underlying operating system, avoiding leaks or crashes during teardown.

**3. Key Derivation: PBKDF2-HMAC-SHA1**

The function derive\_key() derives a 128-bit key using PBKDF2 with HMAC-SHA1. It takes a passphrase and salt, applies 100,000 iterations, and outputs a fixed-length key. This is essential for resisting brute-force attacks and ensuring that the same passphrase maps to different keys across sessions.



**4. AES-CBC Mode Implementation**

The application uses AES-128 in Cipher Block Chaining (CBC) mode as its primary symmetric encryption algorithm. CBC mode is one of the oldest and most widely adopted block cipher modes of operation and provides confidentiality by XORing each plaintext block with the previous ciphertext block before encryption. This mode requires a unique Initialization Vector (IV) for every encryption session to ensure semantic security, i.e., identical plaintexts encrypted separately will result in different ciphertexts.

The encryption is implemented using the OpenSSL EVP interface in the function aes\_cbc\_encrypt(). This function:

* Generates a random IV of 128 bits (16 bytes) using OpenSSL's RAND\_bytes() function.
* Initializes the cipher context with EVP\_EncryptInit\_ex() using AES-128-CBC.
* Encrypts the input plaintext using EVP\_EncryptUpdate() followed by EVP\_EncryptFinal\_ex().
* Ensures the output buffer ct is properly sized to accommodate padding (since AES is a block cipher that requires plaintext to be a multiple of the block size).

The decryption is handled by aes\_cbc\_decrypt(), which:

* Initializes a decryption context with the same AES-128-CBC cipher and uses the IV extracted from the transmitted ciphertext.
* Decrypts the ciphertext using EVP\_DecryptUpdate() and finalizes with EVP\_DecryptFinal\_ex(), which also validates and removes padding.

Both functions are robust in terms of memory management and use dynamically allocated buffers sized using EVP\_MAX\_BLOCK\_LENGTH to ensure the ciphertext or plaintext is fully accommodated. They also return the length of the encrypted or decrypted data, allowing safe truncation or appending during transmission or storage.

**5. Dual-AES Cascade Mode**

The cascade\_encrypt() and cascade\_decrypt() functions implement a two-stage nested encryption scheme using AES-128-CBC. This technique is often referred to as a cascade cipher, where data is encrypted multiple times with different keys and initialization vectors to enhance confidentiality and mitigate the impact of cryptographic weaknesses in any single layer.

The encryption procedure is structured as follows:

* The plaintext is first encrypted using the first key (key1) and a randomly generated initialization vector (IV1). This produces an intermediate ciphertext.
* The intermediate ciphertext is then encrypted again using a second key (key2) and a second random initialization vector (IV2).
* The final output includes both IVs concatenated (IV1 || IV2) followed by the double-encrypted ciphertext.

This layering introduces multiple independent cryptographic barriers that an attacker must overcome to successfully recover the original plaintext. Even if a weakness were discovered in one of the keys or a partial compromise occurred, the secondary encryption layer would act as a fail-safe, preserving the secrecy of the message.

The use of two different keys also means that brute-force or dictionary attacks must be performed separately and successfully on each layer. Moreover, because the cascade uses independent IVs for both stages, it mitigates risks related to IV reuse or IV prediction.

From an implementation perspective, cascade\_encrypt() carefully handles buffer sizing to accommodate padding and cipher block expansion. It dynamically allocates space for the two intermediate encryption steps using std::vector with a maximum size defined by EVP\_MAX\_BLOCK\_LENGTH. The final ciphertext is constructed by combining the two IVs and the output from the second encryption stage.

Decryption is handled by cascade\_decrypt() in the reverse order:

* First, the combined header (32 bytes) is parsed to extract IV1 and IV2.
* The ciphertext is decrypted using key2 and IV2, resulting in the intermediate ciphertext.
* The intermediate ciphertext is decrypted again using key1 and IV1 to obtain the original plaintext.

While dual-layer encryption can offer improved resistance against specific types of attacks (such as certain forms of cryptanalysis), it is not a silver bullet. It introduces additional computational overhead, and without message authentication (such as HMAC or an AEAD mode), it still remains vulnerable to tampering or chosen-ciphertext attacks.

In summary, the dual-AES cascade mode serves as an advanced encryption option within the secure chat application. It is well-suited for users who want extra protection at the cost of increased processing time and message size.

**6. Ciphertext/Plaintext Logging**

The application logs both ciphertext and decrypted plaintext:

A screen shot of a computer code

AI-generated content may be incorrect.This logging is valuable for debugging but must be disabled in production to prevent information leakage.

**7. Key Rotation**

To improve forward secrecy, the application rotates the session key after every 100 messages:

A computer code with text

AI-generated content may be incorrect.

This design prevents attackers from decrypting older messages if a key is compromised.

**8. Diffie–Hellman Fallback**

When no passphrase is provided, the application falls back to using a DH key exchange to generate a shared session key:

* Uses DH\_get\_2048\_256() to generate strong parameters.
* Exchanges public keys and derives a shared secret.
* Hashes the secret with SHA-256 and extracts a salt.

This ensures secure initialization without prior shared secrets.

**9. Main Chat Loop**

The main loop uses select() to wait for either user input or socket data. Upon sending, the input is encrypted and sent. Upon receiving, ciphertext is decrypted and printed:

* FD\_ISSET(0, &fds) handles standard input.
* FD\_ISSET(sock, &fds) handles incoming encrypted messages.

The application is designed to be interactive and responsive for real-time communication.

**10. Error Handling and Portability**

The code is carefully designed to be portable:

* Uses #ifdef \_WIN32 for Windows-specific socket handling.
* Gracefully handles socket closures, memory allocation failures, and OpenSSL errors.
* Uses atomic operations (std::atomic<int>) to track message count safely.

**11. Security Considerations**

* **Encryption Strength:** AES-128-CBC is widely trusted but lacks authentication (no MAC). This can be exploited in padding oracle attacks unless mitigated.
* **Nested Encryption:** Adds complexity but must be carefully implemented to avoid diminishing returns.
* **No Authentication Tag:** A major omission is the lack of message authentication codes (e.g., HMAC). A tampered message may still decrypt into garbage.
* **IV Handling:** IVs are randomly generated and correctly sent with each message.
* **Logging Risk:** Ciphertext and plaintext logging should be disabled outside of testing.

**12. Recommendations for Improvement**

* **Use AEAD modes like AES-GCM or ChaCha20-Poly1305** for built-in authentication.
* **Implement certificate-based mutual authentication** using TLS instead of custom schemes.
* **Enable optional logging via flags** rather than default printing.
* **Add message integrity checks** using HMAC or authenticated encryption.

**Conclusion**

This secure application demonstrates a strong understanding of cryptography, secure communication principles, and practical system programming. It integrates advanced encryption methods, flexible key derivation, and secure socket communication. While production deployment would require additional hardening, the modular design, clear abstractions, and attention to detail make it an excellent educational and experimental foundation for secure messaging systems.