Secure Chat Program Swap

Jawad Chowdhury

CSC 380

1 Overview

This group's implementation works like this:
Messages are encrypted using AES-CTR and tagged with an HMAC before being sent over the network. Each message includes a 16-byte header (nonce/IV), the encrypted message, and a 32-byte HMAC for authentication.
The program uses RSA key pairs stored in PEM files.
Their key assumption is that the client and server already have each other's RSA public keys stored in separate PEM files before starting the chat. The handshake protocol creates temporary X25519 keys for each session and signs them with the long-term RSA keys for authentication.
The message format sent over the network is: `[nonce][cipher length][ciphertext][mac]`
Looking at their code, they do use SHA-256 for key derivation and HMAC-SHA256 for message authentication.

2 Analysis

2.1 Protocol

Strengths:

- Uses X25519 for ephemeral key exchange, providing forward secrecy
- Signs ephemeral public keys with long-term RSA keys for authentication
- Verifies signatures before deriving shared secrets
- Uses separate keys for encryption and MAC

Areas of Concern:

- No replay protection - old messages can be retransmitted by attackers
- Buffer overflow vulnerabilities in handshake code

2.2 Vulnerabilities

2.2.1 Buffer Overflow Vulnerabilities

A bad third party could do a buffer overflow attack on `p_length`

Listing 1: Buffer Overflow in doHandShake

```
// Receive the stuff
int p_length, s_length;
recv(sockfd, &p_length, sizeof(int), 0);
unsigned char peer_buffer[512];
recv(sockfd, peer_buffer, p_length, 0);
recv(sockfd, &s_length, sizeof(int), 0);
unsigned char peer_signature[256];
recv(sockfd, peer_signature, s_length, 0);
```

2.2.2 Integer Overflow and Memory Issues

In `recvMsg()`, the cipher length is used for memory allocation without validation:

Listing 2: Unchecked Memory Allocation in Message Receiving

```
// Receive message components in order: [nonce] [cipher length] [
   ciphertext] [mac]
unsigned char nonce[16];
recv(sockfd, nonce, 16, MSG_WAITALL);

uint32_t cipher_length;
recv(sockfd, &cipher_length, sizeof(uint32_t), MSG_WAITALL);

unsigned char* ciphertext = malloc(cipher_length);
recv(sockfd, ciphertext, cipher_length, MSG_WAITALL);

unsigned char mac[32];
recv(sockfd, mac, 32, MSG_WAITALL);
```

2.2.3 Missing Replay Protection

An attacker can capture and replay old messages, which will be accepted as valid since they have correct MACs.
Suggested Fix: Add a counter to the plaintext before encryption

3 Overall Assessment

The implementation demonstrates good understanding of cryptographic principles and includes several strong design choices like ephemeral key exchange and signature verification. The protocol design is generally sound for protecting against network-level attacks.

However, with minor fixes to the code it could be made better suited for things like replay, and buffer overflow attacks.