Análisis en Profundidad del Código C++: Cifrado y Descifrado de Archivos con AES-256 y TOTP

Garnachos

August 15, 2024

Abstract

Este artículo analiza la implementación de un sistema de cifrado y descifrado de archivos en C++, el cual combina el cifrado simétrico AES-256 en modo CTR con un mecanismo dinámico de generación de claves basado en TOTP (Time-based One-Time Password). Además, se emplea HMAC con SHA-256 para la verificación de la integridad de los datos. Este enfoque criptográfico tiene como objetivo garantizar la seguridad y protección de archivos en sistemas de información.

1 Introducción

En el mundo moderno, la seguridad de la información es un aspecto crucial. Este trabajo presenta un análisis detallado de un sistema de cifrado y descifrado de archivos, el cual emplea múltiples técnicas criptográficas para proporcionar un alto nivel de seguridad. Se exploran las características de AES-256 en modo CTR, el uso de claves dinámicas mediante TOTP, y la implementación de HMAC con SHA-256 para garantizar la integridad de los datos cifrados.

2 Análisis del Código en C++

2.1 Definiciones Globales

El código comienza definiendo varias constantes que son fundamentales para el proceso criptográfico:

- `KEY_SIZE`: El tamaño de la clave AES en bits (256 bits).
- `IV_SIZE`: El tamaño del vector de inicialización (IV) en bytes (16 bytes o 128 bits).
- `CHUNK_SIZE`: El tamaño de los bloques de datos para procesar durante el cifrado/descifrado (16 KB).
- `HMAC_SIZE`: La longitud de salida del HMAC generado con SHA-256 (32 bytes).
- `SECRET_KEY`: La clave secreta utilizada para generar las claves AES dinámicas mediante TOTP.
- `INTERVAL_SECONDS`: Intervalo de tiempo en segundos para la rotación de claves TOTP.

2.2 Funciones Clave

2.2.1 Generación de la clave AES

La función `generateAESKey()` genera una clave AES dinámica utilizando el algoritmo TOTP. Esta clave cambia cada `INTERVAL_SECONDS`, agregando una capa de seguridad adicional frente

a ataques de repetición. La clave se obtiene aplicando HMAC-SHA256 al `timestamp` actual dividido por el intervalo de tiempo.

2.2.2 Procesamiento de Bloques

La función `processChunk()` realiza el cifrado o descifrado de bloques de datos utilizando el contexto de cifrado previamente inicializado. Este proceso se basa en la función `EVP_CipherUpdate()` de la biblioteca OpenSSL, que permite manejar datos cifrados de manera segura.

2.2.3 Generación y Verificación de HMAC

La integridad de los datos cifrados se garantiza mediante la función `generateHMAC()`, que utiliza la clave secreta para calcular un código de autenticación (HMAC) sobre los datos. Durante el proceso de descifrado, se verifica que el HMAC coincida, asegurando que los datos no han sido modificados.

2.2.4 Orquestación del Cifrado/Descifrado

La función `processFile()` gestiona el flujo completo de cifrado o descifrado de un archivo. Este proceso incluye la generación del IV, la lectura y escritura de los archivos en bloques, y el cálculo del HMAC en el caso de cifrado.

3 Seguridad y Criptografía

3.1 AES-256 en Modo CTR

El algoritmo AES es ampliamente utilizado por su seguridad. Al usar una clave de 256 bits en modo CTR, se obtiene un cifrado de flujo que permite un procesamiento eficiente y paralelo de los datos. Este modo de operación también evita problemas asociados con los modos de cifrado por bloques, como el relleno.

3.2 TOTP como Generador de Claves Dinámicas

El uso de TOTP para generar claves dinámicas es una estrategia innovadora en este código. Esta técnica hace que la clave cambie constantemente, lo cual dificulta los ataques de fuerza bruta o repetición, ya que el atacante tendría que comprometer la clave dentro de un intervalo de tiempo limitado.

3.3 HMAC y Verificación de Integridad

HMAC-SHA256 garantiza que los datos cifrados no han sido alterados durante el transporte o almacenamiento. El código genera y verifica un HMAC sobre los datos cifrados, proporcionando una capa adicional de seguridad que protege contra ataques de modificación de datos.

4 Conclusión

El sistema implementado en este código C++ ofrece una solución robusta para el cifrado y descifrado de archivos, combinando AES-256 en modo CTR, TOTP y HMAC-SHA256. Estas técnicas criptográficas trabajan juntas para garantizar tanto la confidencialidad como la integridad de los datos, proporcionando una capa de seguridad adecuada para diversas aplicaciones en sistemas de información.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <ctime>
#include <stdexcept>

const int KEY_SIZE = 32;  // 256 bits
const int IV_SIZE = 16;   // 128 bits
const int CHUNK_SIZE = 16 * 1024;  // 16 KB chunks
const int HMAC_SIZE = 32; // SHA-256 HMAC output size

const unsigned char SECRET_KEY[] = "mysecretkey"; // Change this to a secure key
const int INTERVAL_SECONDS = 30;

// Function to generate a dynamic AES key using TOTP-like algorithm
std::vector<unsigned char> generateAESKey() {
    unsigned long long timestamp = std::time(nullptr) / INTERVAL_SECONDS;
    unsigned char* hmac = HMAC(EVP_sha256(), SECRET_KEY, sizeof(SECRET_KEY) - 1, reinterpret_cast<unsigned char*>(&timestamp), sizeof(timestamp), nullptr, nullptr);

    std::vector<unsigned char> aesKey(hmac, hmac + KEY_SIZE);
    return aesKey;
}

std::vector<unsigned char> processChunk(EVP_CIPHER_CTX* ctx, const std::vector<unsigned char>& chunk, bool& isLastChunk) {
    std::vector<unsigned char> out(chunk.size() + EVP_MAX_BLOCK_LENGTH);
    int outlen = 0;

    if (EVP_CipherUpdate(ctx, out.data(), &outlen, chunk.data(), chunk.size()) != 1) {
        throw std::runtime_error("Error in CipherUpdate");
    }

    out.resize(outlen);
    return out;
}

// Function to generate HMAC for a given data buffer
std::vector<unsigned char> generateHMAC(const std::vector<unsigned char>& data, const std::vector<unsigned char>& key) {
    unsigned char* hmac = HMAC(EVP_sha256(), key.data(), key.size(), data.data(), data.size(), nullptr, nullptr);
    return std::vector<unsigned char>(hmac, hmac + HMAC_SIZE);
}

void processFile(const std::string& inputFile, const std::string& outputFile, bool encrypt) {
    std::ifstream inFile(inputFile, std::ios::binary);
    if (!inFile) {
        throw std::runtime_error("Cannot open input file");
    }

    std::ofstream outFile(outputFile, std::ios::binary);
    if (!outFile) {
        throw std::runtime_error("Cannot open output file");
    }
```

```
55
56      unsigned char iv[IV_SIZE];
57      std::vector<unsigned char> aesKey = generateAESKey();
58
59      if (encrypt) {
60          if (RAND_bytes(iv, IV_SIZE) != 1) {
61              throw std::runtime_error("Error generating random IV");
62          }
63          outFile.write(reinterpret_cast<const char*>(iv), IV_SIZE);
64      } else {
65          inFile.read(reinterpret_cast<char*>(iv), IV_SIZE);
66          if (!inFile) {
67              throw std::runtime_error("Error reading IV from input file");
68          }
69      }
70
71      EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
72      if (!ctx) {
73          throw std::runtime_error("Error creating cipher context");
74      }
75
76      if (EVP_CipherInit_ex(ctx, EVP_aes_256_ctr(), nullptr, aesKey.data(), iv,
            encrypt ? 1 : 0) != 1) {
77          EVP_CIPHER_CTX_free(ctx);
78          throw std::runtime_error("Error initializing cipher");
79      }
80
81      std::vector<unsigned char> chunk(CHUNK_SIZE);
82      std::vector<unsigned char> encryptedData;
83
84      while (inFile) {
85          inFile.read(reinterpret_cast<char*>(chunk.data()), CHUNK_SIZE);
86          std::streamsize bytesRead = inFile.gcount();
87
88          if (bytesRead > 0) {
89              chunk.resize(bytesRead);
90              bool isLastChunk = inFile.eof();
91              std::vector<unsigned char> processedChunk = processChunk(ctx, chunk
                    , isLastChunk);
92              encryptedData.insert(encryptedData.end(), processedChunk.begin(),
                    processedChunk.end());
93          }
94      }
95
96      // If encrypting, append HMAC to the encrypted data
97      if (encrypt) {
98          std::vector<unsigned char> hmac = generateHMAC(encryptedData, aesKey);
99          encryptedData.insert(encryptedData.end(), hmac.begin(), hmac.end());
100     }
101
102     // Write the final data to the output file
103     outFile.write(reinterpret_cast<const char*>(encryptedData.data()),
            encryptedData.size());
104
105     EVP_CIPHER_CTX_free(ctx);
106     inFile.close();
107     outFile.close();
108 }
109
110 void encryptFile(const std::string& inputFile, const std::string& outputFile) {
111     processFile(inputFile, outputFile, true);
112 }
113
```

```cpp
void decryptFile(const std::string& inputFile, const std::string& outputFile) {
    try {
        processFile(inputFile, outputFile, false);
    } catch (const std::exception& e) {
        std::cerr << "Decryption failed with current timestamp key: " << e.what() << std::endl;
        std::cerr << "Trying with previous interval key..." << std::endl;

        // Try decryption with the key from the previous interval
        unsigned long long timestamp = std::time(nullptr) / INTERVAL_SECONDS - 1;
        unsigned char* hmac = HMAC(EVP_sha256(), SECRET_KEY, sizeof(SECRET_KEY) - 1, reinterpret_cast<unsigned char*>(&timestamp), sizeof(timestamp), nullptr, nullptr);

        std::vector<unsigned char> previousAESKey(hmac, hmac + KEY_SIZE);

        std::ifstream inFile(inputFile, std::ios::binary);
        if (!inFile) {
            throw std::runtime_error("Cannot open input file");
        }

        std::ofstream outFile(outputFile, std::ios::binary);
        if (!outFile) {
            throw std::runtime_error("Cannot open output file");
        }

        unsigned char iv[IV_SIZE];
        inFile.read(reinterpret_cast<char*>(iv), IV_SIZE);
        if (!inFile) {
            throw std::runtime_error("Error reading IV from input file");
        }

        EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
        if (!ctx) {
            throw std::runtime_error("Error creating cipher context");
        }

        if (EVP_CipherInit_ex(ctx, EVP_aes_256_ctr(), nullptr, previousAESKey.data(), iv, 0) != 1) {
            EVP_CIPHER_CTX_free(ctx);
            throw std::runtime_error("Error initializing cipher");
        }

        std::vector<unsigned char> chunk(CHUNK_SIZE);
        std::vector<unsigned char> decryptedData;

        while (inFile) {
            inFile.read(reinterpret_cast<char*>(chunk.data()), CHUNK_SIZE);
            std::streamsize bytesRead = inFile.gcount();

            if (bytesRead > 0) {
                chunk.resize(bytesRead);
                bool isLastChunk = inFile.eof();
                std::vector<unsigned char> processedChunk = processChunk(ctx, chunk, isLastChunk);
                decryptedData.insert(decryptedData.end(), processedChunk.begin(), processedChunk.end());
            }
        }

        EVP_CIPHER_CTX_free(ctx);
        inFile.close();
```

```
170
171         // Verify and remove HMAC from the decrypted data
172         if (decryptedData.size() >= HMAC_SIZE) {
173             std::vector<unsigned char> dataWithoutHMAC(decryptedData.begin(),
                    decryptedData.end() - HMAC_SIZE);
174             std::vector<unsigned char> hmac(decryptedData.end() - HMAC_SIZE,
                    decryptedData.end());
175             std::vector<unsigned char> expectedHMAC = generateHMAC(
                    dataWithoutHMAC, previousAESKey);
176
177             if (hmac == expectedHMAC) {
178                 outFile.write(reinterpret_cast<const char*>(dataWithoutHMAC.
                        data()), dataWithoutHMAC.size());
179                 std::cout << "File decrypted successfully with previous
                        interval key." << std::endl;
180             } else {
181                 throw std::runtime_error("HMAC verification failed with
                        previous interval key.");
182             }
183         } else {
184             throw std::runtime_error("File too small to contain valid HMAC.");
185         }
186
187         outFile.close();
188     }
189 }
190
191 int main(int argc, char* argv[]) {
192     if (argc != 4) {
193         std::cerr << "Usage: " << argv[0] << " <encrypt|decrypt> <input_file> <
                output_file>" << std::endl;
194         return 1;
195     }
196
197     std::string mode = argv[1];
198     std::string inputFile = argv[2];
199     std::string outputFile = argv[3];
200
201     try {
202         if (mode == "encrypt") {
203             encryptFile(inputFile, outputFile);
204             std::cout << "File encrypted successfully." << std::endl;
205         } else if (mode == "decrypt") {
206             decryptFile(inputFile, outputFile);
207             std::cout << "File decrypted successfully." << std::endl;
208         } else {
209             std::cerr << "Invalid mode: " << mode << std::endl;
210             return 1;
211         }
212     } catch (const std::exception& e) {
213         std::cerr << "Error: " << e.what() << std::endl;
214         return 1;
215     }
216
217     return 0;
218 }
```