TOPIC - Implementing Security algorithms for an OS

PRADITA G-23011102062

S.VARSHA-2301102082

SANJANAA SHREE C H-23011102088

AIM: Implementing a Secure File Encryption and Decryption System in an OS Environment.

EXISTING SOLUTION(LITERATURE SURVEY):

Many existing file encryption solutions use symmetric key encryption methods such as AES-128, AES-256, and DES. However, some implementations lack:

- **Proper IV (Initialization Vector) Management**
  - Some solutions use a fixed or predictable IV, making encryption vulnerable to attacks.
  - IV reuse leads to patterns in ciphertext, reducing security.
- **Efficient Handling of Large Files**
  - Many encryption tools process files in memory entirely, leading to performance issues and crashes in low-memory environments.
  - Lack of chunk-based encryption causes inefficiencies in large file processing.
- **Secure Key Storage Mechanisms**
  - Some implementations store encryption keys in plain text or within the code, making them prone to theft.
  - Lack of proper key management strategies like secure vaults or hardware security modules (HSMs).
- **Padding Mechanisms to Prevent Data Corruption**
  - Incorrect padding handling can cause decryption failures or expose patterns in encrypted data.
  - Some implementations use weak padding techniques like zero-padding, which may compromise security.

Some open-source encryption tools like VeraCrypt and OpenSSL-based implementations provide robust encryption but may require complex configurations for file-based encryption. Additionally, these tools may not address key management, usability, or integrity verification, leading to security risks in real-world applications.

NOVEL APPROACH:

1. **Random IV Generation for Security:**
   - The function generateIV() creates a random Initialization Vector (IV) using RAND_bytes().
   - This ensures that each encryption operation produces a unique ciphertext, even if the same plaintext and key are used.
   - IV is stored at the beginning of the encrypted file, allowing it to be retrieved during decryption.
   - This method enhances security by preventing attackers from detecting patterns across multiple encrypted files.
2. **File-Based AES-256 Encryption Using EVP API**
   - Your implementation encrypts entire files rather than just text or memory buffers.
   - Uses AES-256-CBC, which is a widely accepted secure encryption mode.
   - OpenSSL's EVP API is used for encryption and decryption, ensuring a modular and flexible approach rather than low-level cryptographic operations.
   - This allows easy modifications (e.g., switching to AES-GCM for authenticated encryption).
3. **Writing and Reading IV to/from File**
   - Instead of handling the IV separately, your approach appends the IV to the encrypted file itself.
   - On decryption, the IV is read from the file header before decrypting the actual content.
   - This makes the implementation self-contained, meaning an encrypted file always carries the required IV for decryption, making it portable.
4. **Chunk-Based File Processing for Efficiency**
   - Instead of loading the entire file into memory, your code reads and encrypts/decrypts data in 1024-byte chunks.
   - This prevents memory overflow and excessive resource consumption, making the program suitable for large files.
   - By processing data incrementally, it is efficient and scalable for real-world applications.
5. **Automatic Padding Handling**

- The EVP_EncryptFinal_ex() and EVP_DecryptFinal_ex() functions handle PKCS#7 padding, ensuring that:
- The last block of data is properly padded if it's smaller than the AES block size.
- Padding is automatically removed during decryption.
- This avoids data corruption and misalignment issues, making the implementation more error-proof.

6. **Error Handling and Validation Mechanisms**
   - The program contains basic error-handling mechanisms such as:
   - Checking file opening errors (perror("File opening failed")).
   - Handling encryption/decryption failures (EVP_EncryptFinal_ex() and EVP_DecryptFinal_ex()).
   - Detecting IV reading failures (fread() validation).
   - These checks help prevent incorrect output or silent failures, ensuring robustness.
7. **Simple Command-Line Interface for User Interaction**
   - The main() function allows users to:
   - Choose encryption or decryption interactively.
   - Provide custom file names for input and output.
   - This makes the program user-friendly while maintaining secure encryption.

IMPLEMENTATION:

- **Encryption Process:**
  1. **Generating the IV:**
     - A random 16-byte IV is generated using OpenSSL's RAND_bytes() function.
     - This IV is essential for ensuring ciphertext uniqueness.
  2. **Writing the IV to the Encrypted File:**
     - Before encrypting any data, the IV is written as the first 16 bytes of the output file.
     - This allows the decryption process to retrieve the correct IV.
  3. **Initializing AES-256-CBC Encryption:**
     - An encryption context is created using OpenSSL's EVP_CIPHER_CTX_new().
     - The AES-256-CBC cipher is initialized with:
     - The 256-bit key.
     - The generated IV.
  4. **Encrypting File Data in Blocks:**
     - The plaintext file is read in fixed-size chunks (e.g., 1024 bytes at a time).

- Each chunk is encrypted using AES-256 and written to the output file.

5. **Padding the Final Block:**
    - If the last block is smaller than 16 bytes, padding is added using PKCS#7.
    - Padding ensures that the final block matches AES's 16-byte requirement.
6. **Writing the Encrypted Data to the File:**
    - The encrypted blocks are continuously written to the output file.
7. **Finalizing the Encryption:**
    - The encryption process is finalized using EVP_EncryptFinal_ex(), ensuring that all remaining bytes (including padding) are properly encrypted.
8. **Closing the File Streams:**
    - The input and output files are closed to prevent memory leaks and data corruption.
    - The encryption context is freed.

- **Decryption process:**

1. **Reading the IV:**
    - The first 16 bytes of the encrypted file are read to extract the IV.
    - This IV is used to initialize the decryption process.
2. **Initializing AES-256-CBC Decryption:**
    - A decryption context is created using OpenSSL.
    - AES-256-CBC is initialized using:
    - The same secret key used for encryption.
    - The retrieved IV from the encrypted file.
3. **Decrypting File Data in Blocks:**
    - The encrypted file is read in fixed-size chunks.
    - Each chunk is decrypted using AES-256 and stored in a buffer.
4. **Handling Padding:**

- Once all data is decrypted, the PKCS#7 padding is removed to restore the original plaintext.

5. **Writing Decrypted Data to the File:**
    - The decrypted plaintext is written to an output file.
6. **Finalizing the Decryption:**
    - The decryption process is finalized using EVP_DecryptFinal_ex().
    - If decryption is successful, the plaintext is fully recovered.
7. **Closing the File Streams:**
    - The input and output files are closed, and the decryption context is freed.

- **User interaction:**

- The user is prompted to choose encryption ('e') or decryption ('d').
- The user provides:
    - For encryption: The name of the plaintext file and the name of the encrypted output file.
    - For decryption: The name of the encrypted file and the name of the decrypted output file.
    - The system processes the file according to the chosen operation.
    - The output file is generated, either containing the encrypted or decrypted data.

➢ **Security measures implemented:**

1. Secure Key Management:

- The 256-bit AES key is not hardcoded in a real-world implementation.
- It should be stored securely using:
- Environment variables.
- Secure key vaults.
- User-generated passwords with key derivation (PBKDF2, Argon2, or bcrypt).

2. IV Randomization:

- A new IV is generated for every encryption operation.
- The IV is stored in the encrypted file to prevent reuse attacks.

3. Efficient File Handling:

- The system reads and writes files in chunks to ensure memory efficiency, allowing it to handle large files.

4. Integrity Verification

- The current implementation does not include integrity verification.
- A cryptographic hash function (HMAC) can be added to verify if the file has been tampered with.

<u>TOOLS USED:</u>

Tools Used in the Secure File Encryption and Decryption System (C with OpenSSL):

1. OpenSSL Library:

- Purpose: Provides cryptographic functions to perform AES encryption and decryption.
- Usage in the Project:
    1. EVP_aes_256_cbc(): Uses AES-256-CBC encryption mode.
    2. EVP_EncryptInit_ex() / EVP_DecryptInit_ex(): Initializes encryption and decryption processes.
    3. EVP_EncryptUpdate() / EVP_DecryptUpdate(): Encrypts/decrypts data in chunks.
    4. EVP_EncryptFinal_ex() / EVP_DecryptFinal_ex(): Completes encryption/decryption and handles padding.
    5. RAND_bytes(): Generates a secure random Initialization Vector (IV).

2. GCC (GNU Compiler Collection):

- Purpose: Compiles the C++ program with OpenSSL dependencies.

3. Standard C++ Libraries:

- Purpose: Handles file operations, input/output, and memory management.
- Usage in the Project:
    - stdio.h: File handling (fopen(), fread(), fwrite(), fclose()).
    - stdlib.h: Memory management and error handling.string.h: String operations (used for key initialization).

4. Linux Terminal / Command Prompt:

- Purpose: Executes the compiled binary and interacts with the user.

5. File System (Text & Binary Files):

- Purpose: Handles plaintext input files and stores encrypted/decrypted data.
- Usage in the Project:
    - Input File (Plaintext): Read as a binary file for encryption.
    - Encrypted File: Stores ciphertext (AES encrypted binary data).
    - Decrypted File: Stores the original plaintext after decryption.

TOOL/LIBRARY	PURPOSE
OpenSSL	Perform AES encryption and decryption
GCC (GNU Compiler Collection)	Compiling the C program with OpenSSL dependencies.
Standard C Libraries	Managing file handling, input/output and memory
Linux Terminal/Command Prompt	Executes and interacts with the encryption system
File System (Binary & Text Files)	Stores encrypted and decrypted data securely.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <openssl/evp.h>

#include <openssl/rand.h>

#define AES_KEY_SIZE 32  // 32 bytes = 256-bit AES key

#define AES_BLOCK_SIZE 16

// Function to generate a random IV

void generateIV(unsigned char *iv) {

    if (!RAND_bytes(iv, AES_BLOCK_SIZE)) {

        fprintf(stderr, "Error generating IV\n");

        exit(EXIT_FAILURE);

    }

}

// Encrypt function

int encryptFile(const char *inputFile, const char *outputFile, const unsigned char *key) {

    unsigned char iv[AES_BLOCK_SIZE];

    generateIV(iv); // Generate a random IV

    FILE *in = fopen(inputFile, "rb");

    FILE *out = fopen(outputFile, "wb");

    if (!in || !out) {

        perror("File opening failed");

        return 0;

    }
```

```
    // Write IV at the beginning of the file
    fwrite(iv, 1, AES_BLOCK_SIZE, out);
    printf("Encryption IV: ");
    for (int i = 0; i < AES_BLOCK_SIZE; i++) printf("%02x ", iv[i]);
    printf("\n");
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
    unsigned char buffer[1024], cipherBuffer[1024 + AES_BLOCK_SIZE];
    int bytesRead, cipherLen;
    while ((bytesRead = fread(buffer, 1, sizeof(buffer), in)) > 0) {
        EVP_EncryptUpdate(ctx, cipherBuffer, &cipherLen, buffer, bytesRead);
        fwrite(cipherBuffer, 1, cipherLen, out);
    }
    EVP_EncryptFinal_ex(ctx, cipherBuffer, &cipherLen);
    fwrite(cipherBuffer, 1, cipherLen, out);
    EVP_CIPHER_CTX_free(ctx);
    fclose(in);
    fclose(out);
    printf("Encryption completed!\n");
    return 1;
}

// Decrypt function
int decryptFile(const char *inputFile, const char *outputFile, const unsigned char *key) {
    FILE *in = fopen(inputFile, "rb");
```

```
FILE *out = fopen(outputFile, "wb");
if (!in || !out) {
    perror("File opening failed");
    return 0;
}

unsigned char iv[AES_BLOCK_SIZE];
if (fread(iv, 1, AES_BLOCK_SIZE, in) != AES_BLOCK_SIZE) {
    printf("Error reading IV!\n");
    fclose(in);
    fclose(out);
    return 0;
}
printf("IV Read Successfully\n");
EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
unsigned char buffer[1024 + AES_BLOCK_SIZE], plainBuffer[1024];
int bytesRead, plainLen;
while ((bytesRead = fread(buffer, 1, sizeof(buffer), in)) > 0) {
    printf("Read %d bytes from encrypted file\n", bytesRead);
    EVP_DecryptUpdate(ctx, plainBuffer, &plainLen, buffer, bytesRead);
    printf("Decrypted %d bytes\n", plainLen);
    fwrite(plainBuffer, 1, plainLen, out);
}
if (EVP_DecryptFinal_ex(ctx, plainBuffer, &plainLen) == 0) {
```

```c
        printf("Decryption Finalization Failed! Possible incorrect key or corrupted data.\n");
    } else {
        printf("Final Decrypted %d bytes\n", plainLen);
        fwrite(plainBuffer, 1, plainLen, out);
    }
    EVP_CIPHER_CTX_free(ctx);
    fclose(in);
    fclose(out);
    printf("Decryption Completed\n");
    return 1;
}
int main() {
    const unsigned char key[AES_KEY_SIZE] = {
        't', 'h', 'i', 's', '_', 'i', 's', '_', 'a', '_', '3', '2', '_', 'b', 'y', 't',
        'e', '_', 'k', 'e', 'y', '_', 'f', 'o', 'r', '_', 'A', 'E', 'S', '!', '\0', '\0'
    };

    char inputFile[256], outputFile[256], choice;
    printf("Enter 'e' for encryption or 'd' for decryption: ");
    scanf(" %c", &choice);
    if (choice == 'e') {
        printf("Enter input file name: ");
        scanf("%s", inputFile);
        printf("Enter output encrypted file name: ");
        scanf("%s", outputFile);
```

```
        if (encryptFile(inputFile, outputFile, key))
            printf("File encrypted successfully!\n");
        else
            printf("Encryption failed!\n");
    } else if (choice == 'd') {
        printf("Enter encrypted file name: ");
        scanf("%s", inputFile);
        printf("Enter output decrypted file name: ");
        scanf("%s", outputFile);
        if (decryptFile(inputFile, outputFile, key))
            printf("File decrypted successfully!\n");
        else
            printf("Decryption failed!\n");
    } else {
        printf("Invalid choice!\n");
    }
    return 0;
}
```

OUTPUT:

- **Initial text file:**

```
os.txt
1  Encryption has been successfully implemented clear
2
```

- **Implementation of encryption and decryption output:**

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ echo "Encryption has been implemented successfully clear" >os.txt
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ gcc osproject.c -o osproject -lssl -lcrypto
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ ./osproject
Enter 'e' for encryption or 'd' for decryption: e
Enter input file name: os.txt
Enter output encrypted file name: encrypt.bin
Encryption IV: ed b8 4b 86 e0 1e 05 9c 1c 41 6e 7c b7 47 c3 73
Encryption completed!
File encrypted successfully!
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ cat encrypt.bin
�K���An|�G�s�1�Dd�`*��$q��6����)�JP�o©+e�
                        ��w����3\B�� ��!��qF�F��pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ gcc osproject.c -o osproject -lssl -lcrypto
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ ./osproject
Enter 'e' for encryption or 'd' for decryption: d
Enter encrypted file name: encrypt.bin
Enter output decrypted file name: decrypt.txt
IV Read Successfully
Read 64 bytes from encrypted file
Decrypted 48 bytes
Final Decrypted 3 bytes
Decryption Completed
File decrypted successfully!
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$ cat decrypt.txt
Encryption has been implemented successfully clear
pradita@DESKTOP-NGNFG3M:/mnt/d/osproject$
```

- **Decrypted text file output:**

```
decrypt.txt
1  Encryption has been successfully implemented clear
2
```

CONCLUSION:

This project successfully implements a secure AES-256 file encryption and decryption system using OpenSSL. It ensures:

- Secure key and IV management.
- Efficient file processing for large files.
- Robust error handling mechanisms.
- Easy command-line interaction.

Future enhancements could include adding integrity checks, using AES-GCM for authenticated encryption, and improving key management practices.