Instituto Politécnico Nacional

Escuela Superior de Cómputo

Cryptography

AES

Grupo: 3CM15

Fecha de entrega: 25 de octubre de 2021

Alumnos:

Cazares Martínez Maximiliano

Cipriano Damián Sebastián

Murillo Granada Esteban Jair

Profesora:

Diaz Santiago Sandra.

Ejecución del programa en Python.

Existen dos archivos Python llamados _AES_.py y _AES_test.py, de los cuales, el primero contendrá todo el código responsable del cifrado y el descifrado de AES y el segundo archivo solo contiene código para ejecutar las funciones del primero. Para poder ejecutar _AES_test.py necesitamos que la biblioteca "pycryptodome" esté instalada.

Organización.

Las funciones "GenerationAESKey" y "GenerationAESiv" son las encargadas de generar llaves aleatorias y vectores de inicialización para AES.

Las llaves que podemos generar con dicha función pueden ser de tamaño 16, 24 y 32 bytes; para nuestro caso el tamaño será uno de los parámetros de la función generadora de llaves, además, ambas funciones antes mencionadas cuentan con un parámetro llamado "filename" el cual es el nombre del archivo en donde se guardaran dichos valores en base 64.

```python
def GenerationAESKey(key_size, filename):
    key = get_random_bytes(key_size)
    encode_key = b64.b64encode(key)
    StoreData(filename, encode_key)

def GenerationAESiv(filename):
    iv = get_random_bytes(AES.block_size)
    encode_iv = b64.b64encode(iv)
    StoreData(filename, encode_iv)
```

La función de cifrado tiene 4 parámetros esenciales, "plaintext_filename", "key_filename", "vector_filename" y "mode_operation", los cuales serán el nombre del archivo en plano a cifrar, el nombre del archivo donde esta almacenada la llave, el nombre donde se encuentra almacenada el vector de inicialización y el modo de operación que usará.

La función cifrará y almacenará el texto cifrado en base 64 en un archivo con el mismo nombre del archivo con el texto en plano pero con extensión ".aes".

```python
def AESDecipher(ciphertext_filename, key_filename, iv_filename, mode_operation):
    key = b64.b64decode(GetData(key_filename))
    iv = b64.b64decode(GetData(iv_filename))
    ciphertext = b64.b64decode(GetData(ciphertext_filename))

    decipher = AES.new(key, mode_operation, iv)
    decrypt_text = decipher.decrypt(ciphertext)
    plaintext = str(decrypt_text, 'utf-8')

    filename = f'{ciphertext_filename[:-4]}_.txt'
    StoreData(filename, plaintext)
```

La función de descifrado tiene los mismos parámetros que la fusión de cifrado solo que en lugar de tener el nombre del archivo del texto en plano tiene el nombre del archivo con el texto cifrado.

Dicha función descifra el texto cifrado y lo almacena en un archivo con el mismo nombre que el del texto cifrado, pero con una la terminación "_.txt" para ser diferenciado del texto en plano original.

Ambas funciones pueden usar los modos de operación OFB y CFB pero hay que tener cuidado con que el mismo modo de operación se use para cifrar y descifrar, pues en caso contrario no funcionara.

```python
def AESCipher(plaintext_filename, key_filename, vector_filename, mode_operation):
    key = b64.b64decode(GetData(key_filename))
    iv = b64.b64decode(GetData(vector_filename))
    plaintext = DataToBytes(plaintext_filename)

    cipher = AES.new(key, mode_operation, iv)
    ciphertext = cipher.encrypt(plaintext)

    encode_ciphertext = b64.b64encode(ciphertext)
    filename = f'{plaintext_filename[:-4]}.aes'
    StoreData(filename, encode_ciphertext)
```

Ejecución.

Generamos una llave aleatoria de 16, 24 o 32 bytes con las siguientes líneas de código. Dicha llave se guardará en un archivo con el nombre que se le pasa a la función como segundo parámetro.

```python
GenerationAESKey(16, '16Key.txt')
GenerationAESKey(24, '24Key.txt')
GenerationAESKey(32, '32Key.txt')
```

Para este ejemplo tenemos que comentar la línea 3 del cifrador y del descifrador para hacer uso de un vector de inicialización oficial para comprobar el correcto funcionamiento de AES.

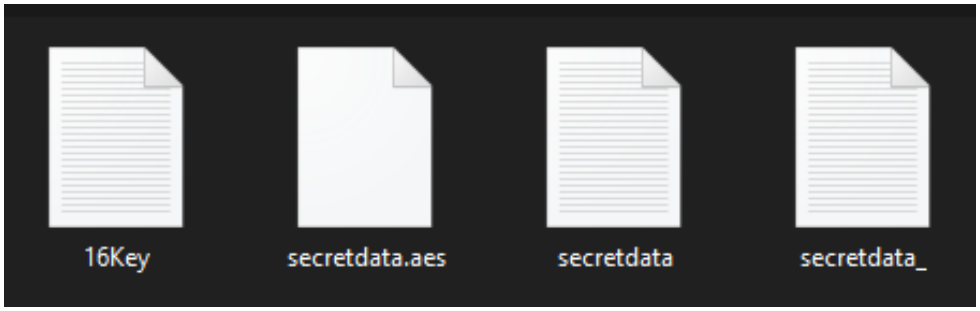
Así, para ejecutar el cifrador y el descifrador de AES tenemos que comentar o descomentar las variables respectivas de las llaves que generamos y el modo de operación que se desee usar.

```
key_filename = '16Key.txt'
key_filename = '24Key.txt'
key_filename = '32Key.txt'

iv = bytes.fromhex('80000000000000000000000000000000')

file_to_encrypt = 'secretdata.txt'
file_to_decrypt = 'secretdata.aes'

mode_operation = AES.MODE_OFB
mode_operation = AES.MODE_CFB
```

El cifrador y descifrador de AES se pueden usar simultáneamente o de forma separada, así que para ejecutarlos necesitamos comentar o descomentar cualquiera de las siguientes líneas.

```
AESCipher(file_to_encrypt, key_filename, iv, mode_operation)
AESDecipher(file_to_decrypt, key_filename, iv, mode_operation)
```

Para este ejemplo, generamos una llave de 16 bytes y hacemos uso del modo OFB para encriptar y desencriptar el archivo “secretdata.txt”. Al terminar la ejecución se generarán los archivos “secretdata.aes”, “secretdata_.txt” y “16Key.txt” en la carpeta donde se encuentre el archivo Python.



Ejecución del programa en C++.

Se hicieron dos programas para el cifrado de AES con diferentes modos de operación “CBC” y “CFB”, en estos vemos las diferentes formas de que se puede emplear el cifrado AES. Se debe de tener en cuenta que estos programas fueron realizados tomando como herramienta la librería “CryptoPP”, además se recomienda el uso de Linux para su rápida ejecución, ya que es más sencillo instalar ahí la librería que en Windows.

Organización.

En ambos programas se realizó la generación de “Key” e “IV” de manera aleatoria, para una rápida construcción, pero tomando en cuenta que, si se desea modificar más adelante para que la introducción sea manual con unas llaves de 16, 24 y 32 bytes, se pueda lograr.

AES_CBC:

```
/*
    byte key[AES::DEFAULT_KEYLENGTH] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef, 0x23,0x45,0x67,0x89,0xab,0xcd,0xef,0x01};
    byte iv[AES::BLOCKSIZE] = {0x12,0x34,0x56,0x78,0x90,0xab,0xcd,0xef, 0x34,0x56,0x78,0x90,0xab,0xcd,0xef,0x12};
*/
    byte key[ CryptoPP::AES::DEFAULT_KEYLENGTH ], iv[ CryptoPP::AES::BLOCKSIZE ];
    memset( key, 0x00, CryptoPP::AES::DEFAULT_KEYLENGTH );
    memset( iv, 0x00, CryptoPP::AES::BLOCKSIZE );
```

AES_CFB:

```
/*
    byte key[AES::DEFAULT_KEYLENGTH] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef, 0x23,0x45,0x67,0x89,0xab,0xcd,0xef,0x01};
    byte iv[AES::BLOCKSIZE] = {0x12,0x34,0x56,0x78,0x90,0xab,0xcd,0xef, 0x34,0x56,0x78,0x90,0xab,0xcd,0xef,0x12};
*/
    AutoSeededRandomPool prng;

    byte key[AES::DEFAULT_KEYLENGTH];
    byte iv[AES::BLOCKSIZE];

    prng.GenerateBlock(key, sizeof(key));
    prng.GenerateBlock(iv, sizeof(iv));
```

Los dos programas cuentan con 4 parámetros esenciales para su funcionamiento: “plaintext”, el cual es el texto que se cifrara, la “Key” y el “IV” los cuales son necesarios para el cifrado AES y por último el “decryptedtext” el cual conservará el texto descifrado por el programa.

El plaintext es leído y almacenado por la función “openField”

```
string openFile(string file){
    string ans = "";
    fstream myfile;
    myfile.open(file, ios::in);
    if(myfile.is_open()){
        string line;
        while(getline(myfile,line)){
            ans+=line;
        }
        myfile.close();
    }
    return ans;
}
```

Como se había mencionado con anterioridad el cifrado se realizó con diferentes modos, “CBC” y “CFB”. Recordemos que el modo CBC asegura que, si el bloque de texto sin formato se repite en el mensaje original, producirá un texto cifrado diferente para los bloques correspondientes. Tenga en cuenta que la clave que se utiliza en el modo CBC es la misma; solo el “IV” es diferente y el modo CFB significa Modo de retroalimentación de cifrado, hace las mismas funciones que el CBC pero cambiando de lugar los valores.

AES_CBC:

```
//Cipher
CryptoPP::AES::Encryption aesEncryption(key, CryptoPP::AES::DEFAULT_KEYLENGTH);
CryptoPP::CBC_Mode_ExternalCipher::Encryption cbcEncryption( aesEncryption, iv );

CryptoPP::StreamTransformationFilter stfEncryptor(cbcEncryption, new CryptoPP::StringSink( ciphertext ) );
stfEncryptor.Put( reinterpret_cast<const unsigned char*>( plaintext.c_str() ), plaintext.length() + 1 );
stfEncryptor.MessageEnd();
```

AES_CFB:

```
//Cipher
    CFB_Mode< AES >::Encryption e;
    e.SetKeyWithIV(key, sizeof(key), iv);
    StringSource(plain, true,
        new StreamTransformationFilter(e,new StringSink(cipher)));
```

Ambos fragmentos de códigos se encargan de realizar el cifrado del archivo “plainttext” con AES junto con sus respectivos módulos. Al finalizar se debe de realizar el cifrado a base64 y guardarlo en un archivo, para que este se pueda leer y descifrar después.

```
cout << "Encoded Base64: ";
cin >> file;
CryptoPP::StringSource scipher(ciphertext, true,
new CryptoPP::Base64Encoder(new CryptoPP::StringSink(encoded)));
saveFile64(encoded,file);
```

```
void saveFile64(string text, string file){
    fstream myfile;
    myfile.open(file, ios::out);
    if(myfile.is_open()){
            myfile << text;
        myfile.close();
    }
}
```

```
cout << "Decode Base64: ";
cin >> file;
decoded = openFile(file);
CryptoPP::StringSource sencoded(encoded, true,
new CryptoPP::Base64Decoder(new CryptoPP::StringSink(decoded)));
```

El descifrado se debe realizar con la lectura del archivo base64 y pasarlo a su forma “natural” para ser procesado por el descifrado de AES

AES_CBC:

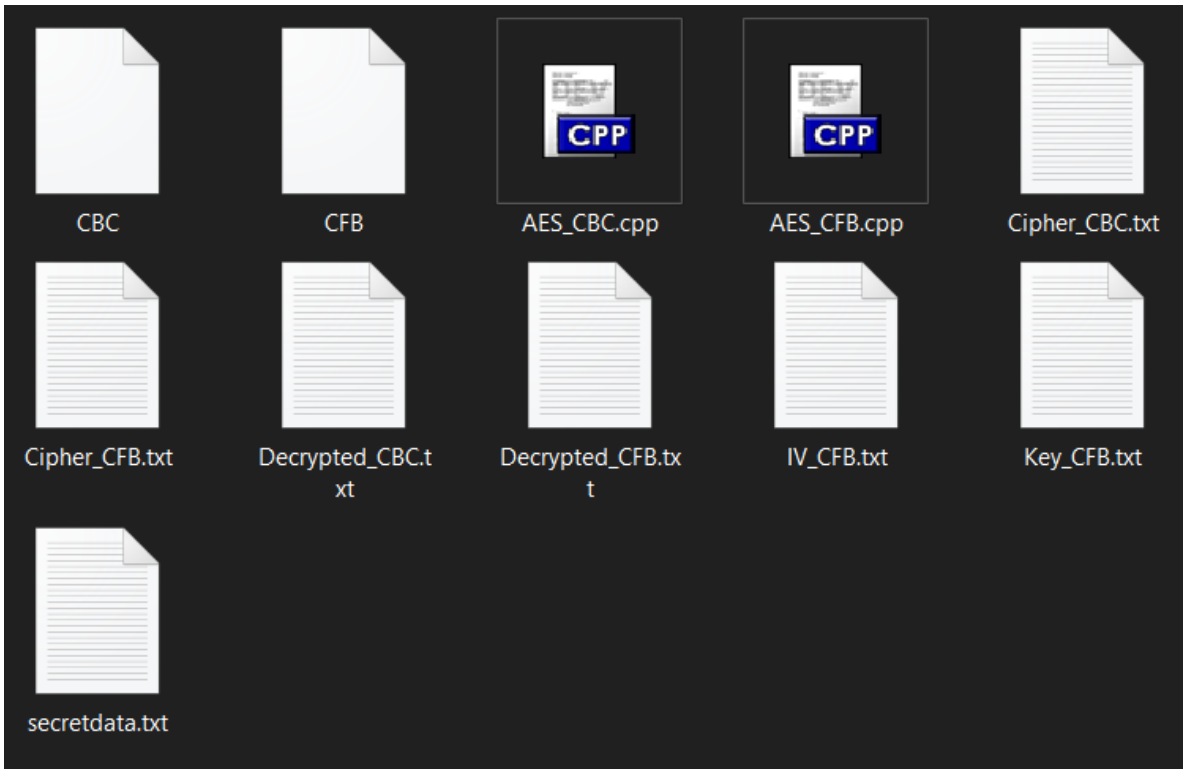
```
// Decrypted
CryptoPP::AES::Decryption aesDecryption(key, CryptoPP::AES::DEFAULT_KEYLENGTH);
CryptoPP::CBC_Mode_ExternalCipher::Decryption cbcDecryption( aesDecryption, iv );

CryptoPP::StreamTransformationFilter stfDecryptor(cbcDecryption, new CryptoPP::StringSink( decryptedtext ) );
stfDecryptor.Put( reinterpret_cast<const unsigned char*>( ciphertext.c_str() ), ciphertext.size() );
stfDecryptor.MessageEnd();
```

AES_CFB:

```
//Decrypted
    CFB_Mode< AES >::Decryption d;
    d.SetKeyWithIV(key, sizeof(key), iv);
    StringSource s(cipher, true,
        new StreamTransformationFilter(d,new StringSink(recovered)));
```

Al finalizar se crea un archivo para almacenar el texto descifrado.

Como ejemplo se corren ambos programas con “Key” e “IV” aleatorios y hacemos uso del archivo “secretdata.txt” para realizar las funciones de los programas. Al terminar se generarán los archivos correspondientes en la carpeta donde se encuentren los programas:



Observamos que las librerías tienen una diferencias significativa desde su instalación hasta su uso en práctica, esto se debe más que nada por el lenguaje en que se maneja, en Python para poder instalar una nueva librería únicamente es necesario el uso de una línea de codigo para que esta sea descargada e instalada: “Pip install pycryptodome”, mientras que en C++ la librería debe ser descargada desde un navegador, además de realizar unos pasos para que esta pueda estar lista para su uso, por lo que se requirió el cambio de sistemas operativo para la instalación y trabajo de esta, ya que de igual forma en Linux solo es necesario el uso de dos comandos para su uso: “sudo apt-get update” y “sudo apt-get install libcrypto++-dev libcrypto++-doc libcrypto++-utils”.

Como se mencionó el lenguaje hace una brecha aún más grande, ya que ambos son muy diferentes en términos de sintaxis, simplicidad, uso y enfoque general de la programación, en Python uno no tiene que estar peleando por el tipo de dato y lo que se puede hacer con él, pero en C++ las funciones son específicas para cada tipo de dato, por lo que tienes que encontrar la función indicada para realizar una tarea o tener que realizar diferentes pasos para que se cumplan dichas especificaciones. De esta manera se pueden mencionar una y otras más diferencias, pero al final las librerías son para su uso, por lo que, se tiene en claro y se recalca que las librerías ayudan en la simplificación de pasos para cifrar o descifrar.