[CNT5410] CNS Assignment 2

Rahul Porwal

The goal of this assignment is to gain familiarity with the use of cryptographic libraries like openSSL. We had to implement encryption, decryption and transmission using C++. I have created 2 executables one for encryption (ufsend) and the other for decryption (ufrec).

1 Steps to Execute

- Local mode:
  - Consider example.txt as input for ufsend. Delete example.txt.ufsec, if present on the system.
  - ./ufsend example.txt -l . output will be example.txt.ufsec.
  - rm example.txt
  - ./ufrec example.txt.ufsec -l
  - Final output will be example.txt, which will be a decrypted file.
- Daemon mode:
  - Consider example.txt as the input file for ufsend.
  - ensure not to have example.txt on the system running ufrec and example.txt on the system running ufsend.
  - run server side first, that is, ./ufrec example.txt -d ip:port
  - this will be waiting for input on the specified port.
  - Now, run client side, that is, ./ufsend example.txt -d ip:port
  - Then encryption will occur at ufsend system and encrypted data will be sent to ufrec, where decryption takes place.

2 Encryption

- Initial Arguments: Here, I am passing the arguments input file to be encrypted, an option to choose between daemon mode (-d) and local mode(-l). If it is on daemon mode, then the ip and port number arguments are needed.
- Password generation: Here, I am taking a user defined input for password.
- Key generation (Task 1): Using PBKDF2 (Password Based Key Derivation Function) to generate key using the user defined password. The openssl function for PBKDF2 is PKC5_PBKDF2_HMAC(). I am generating a 32 byte key for this assignment.
- Random IV generation: I am using RAND_bytes() function from openssl to generate random iv. I am generating a 16 byte random IV.

```c
unsigned char* keygenerator(const char* pwd, const unsigned char* salt)  // implementing PBKDF2
{
    int keylen = 32;
    unsigned char *out;
    out = (unsigned char *)malloc(sizeof(char)*32); //out is the variable in which the result is saved
    unsigned int iterations = 4096;                 //iterations determines how slow or fast the key derivation runs
    PKCS5_PBKDF2_HMAC(pwd, strlen(pwd),               // salt are supposed to be randomly generated bytes, but for simplicity, we have used Sodium Chloride
                      salt, strlen(salt), iterations,
                      EVP_sha3_256(),
                      keylen, out);


    return out;

}
```

Figure 1: PBKDF2 code snippet

```
                                        const char*
rahul@rahul-virtual-machine:~/CNS Assignment2$ cp ufsend testfile.check
rahul@rahul-virtual-machine:~/CNS Assignment2$ ./ufsend testfile.check -l
Create a Password
Hello
Key: 18 f3 cf e2 23 2b 51 db 3a 08 fc 56 89 79 52 e6 22 82 76 f6 ea b5 7e 01 dd 83 2c db b6 51 01 59
```

Figure 2: PBKDF2 Result (Task 1)

```c
unsigned char* randomivgenerator()
{
    unsigned char *iv;
    iv = (unsigned char *)malloc(sizeof(char)*32);
    RAND_bytes(iv, 16); // generates 16 random bytes using a cryptographically secure pseudo random generator
    return iv;
}
```

Figure 3: Random IV generation code snippet

```
rahul@rahul-virtual-machine:~/CNS Assignment2/Porwal-assign2$ ./ufsend example.txt -l
Create a Password
HELLO
Key: 26 34 78 2b cf 2e 93 94 fe c9 59 13 f8 b1 9f a8 58 d9 14 d7 2f ce e7 b8 af 8b b2 ef 11 25 17 b8
IV: f7 45 ea 45 97 ea b4 0d 29 8a 65 a8 2d b1 c5 ca
```

Figure 4: Random IV generation result

- File Existence Check (Task 6): Check if the encrypted file of the input file is already present in the system.

```
if (fs::exists(encryptedfilename))     // checking if encrypted file already exists
{
    cout << "File Already Exists" << endl;
    return 33;
}
```

Figure 5: File Check code snippet

```
rahul@rahul-virtual-machine:~/CNS Assignment2$ ls
example.txt  example.txt.ufsec  iv.txt  Makefile  testfile.check  testfile.check.ufsec  ufrec  ufrec.cpp  ufsend  ufsend.cpp
rahul@rahul-virtual-machine:~/CNS Assignment2$ ./ufsend example.txt -l
File Already Exists
rahul@rahul-virtual-machine:~/CNS Assignment2$
```

Figure 6: code exits when file is present (Task 6)

- AES-256 Encryption (Task 2): Here, I am implementing AES-256 encryption in GCM mode. The function and API calls are provided by openSSL. The encrypt function returns the length of ciphertext generated. The algorithm to be used for encryption is specified by API calls coming from evp.h library. For AES256 in gcm mode, we use EVP_aes_256_gcm().

3 Transmission

- Local Transmission: Use argument (-l) for this. Here, ufsend takes input file encrypts the data and dumps it into another file with the same name but with added extension of ".ufsec". The IV is also dumped into a .txt file, which will the be read by ufrec in local mode. ufrec will read both .ufsec file for ciphertext, .txt for iv and decrypt the ciphertext and put it back into another file with same name and extension as that of input file for ufsend.
- Daemon mode(Task 3) : In daemon mode, ufsend will first encrypt input file, then send the ciphertext and the random generated iv over tcp to ufrec. ufrec will be running on daemon mode waiting for the data. Once it receives data from ufsend, it pushes it onto receive buffer and then stores it into a .ufsec extension file with the same name. Then cipher text is extracted from this .ufsec file and decrypted data is pushed into the input file without .ufsec extension.
- Note: ufsend is treated as client and ufrec is considered to be server.

4 Decryption

- Password request: Here, I am asking for password.
- Key generation: This is similar to what we do for encryption side.
- IV reception: In local mode, ufrec receives iv from iv.txt generated by ufsend.
- File Existence Check (Task 6): Check if the decrypted file of the input file is already present in the system.

```
int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key,
            unsigned char *iv, unsigned char *ciphertext)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        cout<<"error1";

    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, key, iv))  // Initializing encryption using aes 256 in gcm, 32 byte key, 16 byte iv

        cout<<"error2";

    /*
     * Provide the message to be encrypted, and obtain the encrypted output.
     * EVP_EncryptUpdate can be called multiple times if necessary
     */
    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
        cout<<"error3";
    ciphertext_len = len;

    /*
     * Finalise the encryption. Further ciphertext bytes may be written at
     * this stage.
     */
    if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
        cout<<"error4";
    ciphertext_len += len;


    EVP_CIPHER_CTX_free(ctx);              // freeing ctx

    return ciphertext_len;
}
```

Figure 7: AES256 encryption code snippet (Task 2)

```
rahul@rahul-virtual-machine:~/CNS Assignment2$ ./ufsend example.txt -l
Create a Password
HELLO
Key: 26 34 78 2b cf 2e 93 94 fe c9 59 13 f8 b1 9f a8 58 d9 14 d7 2f ce e7 b8 af 8b b2 ef 11 25 17 b8
cipher text:
0000 - ae d8 16 fe 0f 24 55 f0-5f d3 c4 48 3a 7a 3b a7   .....$U._..H:z;.
0010 - bd 4c f8 1e 79 aa d9 61-cc 14 c5 8f 7b 46 19 cc   .L..y..a....{F..
0020 - 9c 45 d5 77 e5 f2 a0 5b-b3 99 59 6b fa 9b e6 ce   .E.w...[..Yk....
0030 - 93 0b 93 eb 62 81 d4 59-18 4f 3a 08 f8 7c 44 b8   ....b..Y.O:..|D.
0040 - e3 76 c1 93 c0 ba dc 1c-0e b0 e1 25 f6 2b f7 8d   .v.........%.+..
0050 - 98 1b 49 da 32 7d 0e 69-97 42 4f 8a 3c 95 70 de   ..I.2}.i.BO.<.p.
0060 - e1 24 2c 84 1a 28 92 7c-40 9f ef d2 b9 3e 0d 6b   .$,..(.|@....>.k
0070 - 9a 96 fa 1a 72 80 0a 74-b5 b2 66 58 d7 bb 0d e0   ....r..t..fX....
0080 - bb d4 b2 8b 6c c0 09 20-fe 37 a6 b0 cd a3 c5 af   ....l.. .7......
0090 - ad cc 47 da ab 84 7e 89-56 29 48 c9 b9 dc d6 95   ..G...~.V)H.....
00a0 - 64 99 ac 27 9a 1d 72 a6-c8 2c 73 bc 03 f5 19 e4   d..'..r..,s.....
00b0 - b4 1b aa 55 05 2e 72 25-16 4c 4a 60 74 70 a1 a4   ...U..r%.LJ`tp..
00c0 - fd 71 0b ef e1 2d d9 8a-c2 3e b4 9f d2 65 5e 80   .q...-...>...e^.
00d0 - c2 24 d7 9b 91 d4 21 ac-62 7f fb 9c aa d9 44 a9   .$....!.b.....D.
00e0 - d0 1f 8b 94 cf c6 61 63-2c f8 20 af 8d 0f a5 d9   ......ac,. .....
00f0 - 61 e4 5f 02 c4 0f 10 51-4e 10 6e d5 ac c3 b5 2e   a._....QN.n.....
0100 - 5c d5 bd 38 76 4c b5 35-67 c8 a8 28 59 9e e1 5d   \..8vL.5g..(Y..]
0110 - fb 97 40 6b 1c 40 47 80-db 1d 13 80 ad 3a 6d cd   ..@k.@G......:m.
0120 - 45 94 4e f6 3b 25 7b a2-60 66 db e7 b9 93 51 7b   E.N.;%{.`f....Q{
0130 - 90 5d d9 b1 d0 95 4f 7d-70 bd 2f 57 1b f2 59 92   .]....O}p./W..Y.
0140 - 50 d5 a8 80 a8 c1 a2 6c-83 d0 a8 2b 6c 3c 0f 8d   P......l...+l<..
0150 - b5 66 66 f1 eb 09 26 0e-6f 44 c8 d5 10 6c b1 76   .ff...&.oD...l.v
```

Figure 8: AES256 GCM mode Encryption Result ciphertext

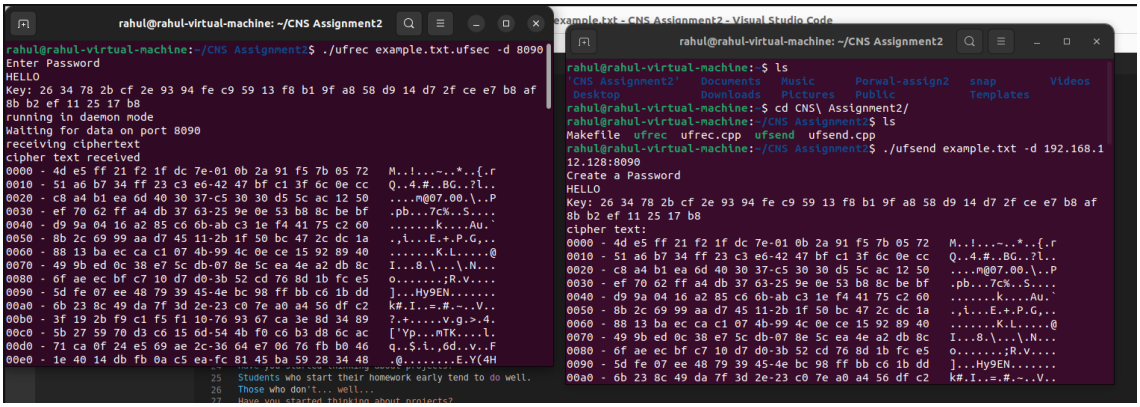

Figure 9: (Task 3)Transmission on daemon mode

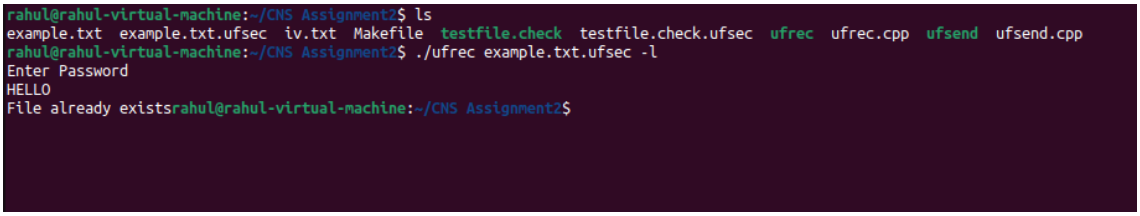

Figure 10: code exits when file is present (Task 6)

- AES-256 Decryption (Task 4 and Task 5): It uses the same key derived from PBKDF2 and IV received from ufsend either through iv.txt or over TCP to decrypt the ciphertext and push it to a file with the same name as that of the input for ufsend. If we modify the cipher text manually, then the deciphered text we get after decryption also gets corrupted.

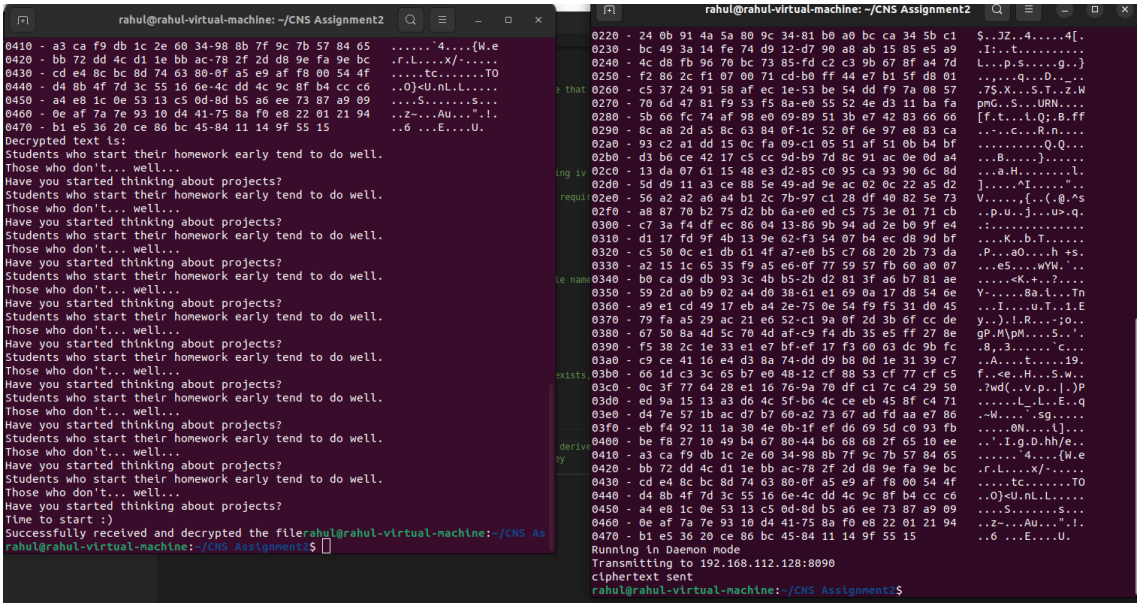

Figure 11: Decryption result (Task 4)

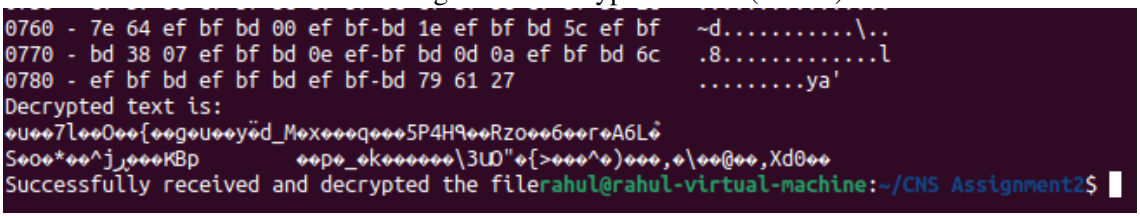

Figure 12: Task 5 - decryption after modifying ciphertext manually