CSE4057 Introduction to Systems Security

Homework # 1

Ahmet Hakan Şimşek 150117060

Muhammed Bera Koç 150116062

Senanur Güvercinoğlu 150119740

1) Generation of public-private key pairs.

We initialize public and private key after that create a Public Key function and finally we make a function for Encryption and Decryption separately.

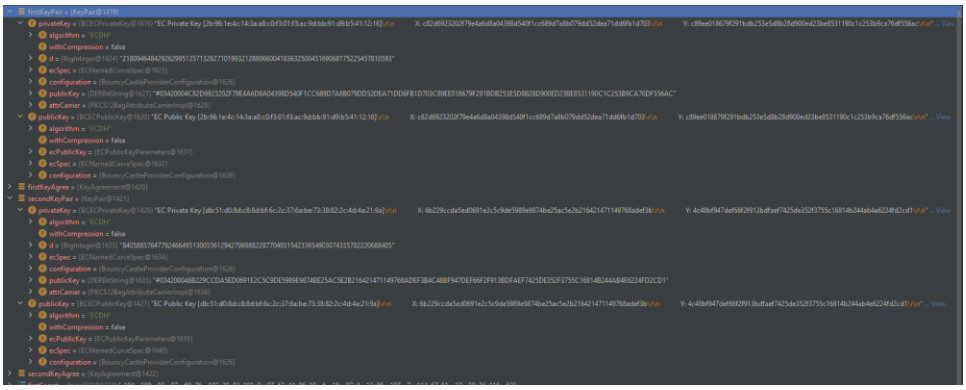

```
enter your text to encyrpt
take it easy!
Encrypted Text:zuo9CZj59g2zQtn8lDcZaCxhRF2ALzoiRlhNnQw1U90GPgYrTvs8ujB2mwJMIbfuvsRw07Qfm88HeqAnl3Lqh22Psw3DfEaSGXfmKkTdf5oIX1GRbk4VJ4sODzkHJhsPYYyBQaVxNZ5tK6eFutvEU3/d
FwDg5FaUnL8ZiYHO1Lrbw/gh5KcIdMzxQnoiSP5S7qc/oCrkf2GWizxrwt0ICLIxijyWkchNOV2Ojd6U7IemqkmOnlSPDnzLhivo4LBTriYPegQBsdFEG5mXFxBf+JEfjzBCPT7TiHpg5NCvmVQBLIjeIkust+JA0hmTFCw
4c1E2MK+iJSboVBb/1oy40Q==
press any key to decrypt text

Decrypted Message:take it easy!
```

1b-) Generate two Elliptic-Curve Diffie Helman public-private key pairs.

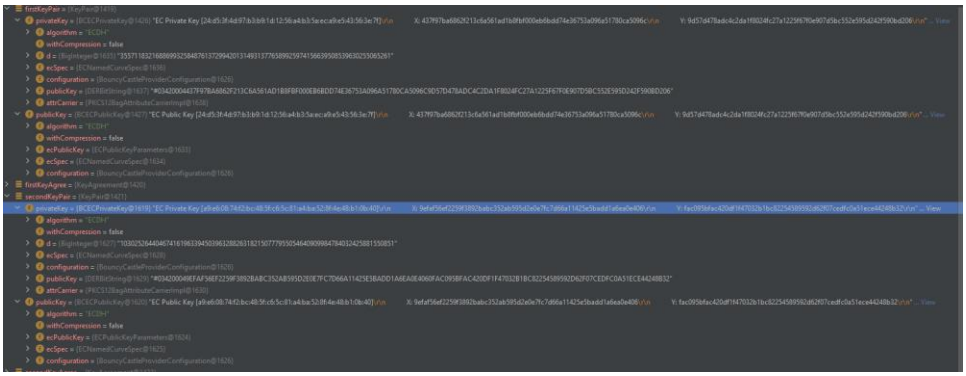
We feed parameter as ECDH, BC and used P-256 curve to generate keys.

```
static void GenerateECDHKeys() throws Exception {
    Provider BC = new org.bouncycastle.jce.provider.BouncyCastleProvider();
    ProviderList.newList(BC);
    ECNamedCurveParameterSpec eCurve = ECNamedCurveTable.getParameterSpec( s: "P-256");
    KeyPairGenerator keyPairGenerator = (KeyPairGenerator) KeyPairGenerator.getInstance( algorithm: "ECDH", BC);
    keyPairGenerator.initialize(eCurve, new SecureRandom());

    KeyPair firstKeyPair = keyPairGenerator.generateKeyPair();

    KeyAgreement firstKeyAgree = KeyAgreement.getInstance( algorithm: "ECDH", BC);

    firstKeyAgree.init(firstKeyPair.getPrivate());

    KeyPair secondKeyPair = keyPairGenerator.generateKeyPair();

    KeyAgreement secondKeyAgree = KeyAgreement.getInstance( algorithm: "ECDH", BC);

    secondKeyAgree.init(secondKeyPair.getPrivate());

    firstKeyAgree.doPhase(secondKeyPair.getPublic(), lastPhase: true);
    secondKeyAgree.doPhase(firstKeyPair.getPublic(), lastPhase: true);

    byte[] firstSecret = firstKeyAgree.generateSecret();
    byte[] secondSecret = secondKeyAgree.generateSecret();

    System.out.println(new String(firstSecret));
    System.out.println(new String(secondSecret));
}
```

First ECDH Public private key pairs



Second ECDH Public private key pairs



2a-) Generation of Symmetric keys

We encrypt and then decrypt 128 and 256 bits symmetric keys code block is below.

```
static void GenerationOfSymmetricKeys() throws Exception {
    SecretKey symmetricKey = giveAESKey();
    System.out.println("Symmetric key size : " + KEY_SIZE);
    System.out.println("Symmetric Key is :" + DatatypeConverter.printHexBinary(symmetricKey.getEncoded()));

    byte[] initializationVector = giveInitializationVector();

    String plainText = "CSE4057 Spring 2022 Information System Security";

    // Encrypting the message using the symmetric key
    byte[] cipher = AESEncryption(plainText,symmetricKey,initializationVector);

    System.out.println( "Encrypted Message : "+ DatatypeConverter.printHexBinary(cipher));

    // Decrypting the encrypted message
    String decrypted = AESDecryption( cipher,symmetricKey,initializationVector);

    System.out.println("Your original message is: "+ decrypted);
    System.out.print("");
}
```

128 bits key result

```
Symmetric key size : 128
Symmetric Key is :096FE0097522F73FD699324AEA0A5A6A
Encrypted Message : 3BD304A9C05F36B9609C185808E7CFF05FA8F5CCED3880F9AE1CCE94C3C1E7096C5446C30D1A389C7549A7564B2FB5C2
Your original message is: CSE4057 Spring 2022 Information System Security

Process finished with exit code 0
```

256 bits key result

```
Symmetric key size : 256
Symmetric Key is :C1B35559C876DB6A0856D213038C916CD808E8C9C0E3038AE1B2FEDA106E84B8
Encrypted Message : 357787494F60CF9F8B3B46AC8D2F944CF0A82CA4D0A050175DEA5C8D04F3112BA9169AA1778EEEBEE3FFEC8BEC54407D
Your original message is: CSE4057 Spring 2022 Information System Security

Process finished with exit code 0
```

2b) This part could not be done

3)Generation and Verification of Digital Signature

Firstly, we compute a keyed hash for a source file and create a target file with the keyed hash prepared to the contents of the source file.

```
namespace SignAndVerifyDigitalSignature
{
    class Program
    {
        static void Main(string[] args)
        {
            Protection protection = new Protection();

            string dataFile;
            string signedFile;


            try
            {
                if (args.Length < 2)
                {
                    dataFile = @"text.txt";
                    signedFile = "SignedFile.enc";

                    Write("Write some text to sign:");
                    string text = ReadLine();
                    if (!File.Exists(dataFile))
                    {
                        using(StreamWriter sw = File.CreateText(dataFile))
                        {
                            sw.WriteLine(text);
                        }

                    }
```

If no file names are specified, create them after that create file to write to create a random key using a random generator this would be the secret key shared by sender and receiver

```
byte[] secretKey = new byte[64];

using (RNGCryptoServiceProvider rng=new RNGCryptoServiceProvider())
{
    rng.GetBytes(secretKey);

    protection.SignFile(secretKey, dataFile, signedFile);

    protection.VerifyFile(secretKey, signedFile);

}
```

RNGCryptoServiceProvider is an implementation of a random number generator

The array is now filled with cyrptogrophically strong random bytes

Use the secret key to sign the message file after that we verify the signed file.

```
public void SignFile(byte[] key, string sourceFile, string destFile)
{
    using (HMACSHA256 hmac =new HMACSHA256(key))
    {
        using (FileStream inStream=new FileStream(sourceFile, FileMode.Open))
```

Computes a keyed hash for a source file and creates a target file with the keyed hash prepared to the contents of the source file

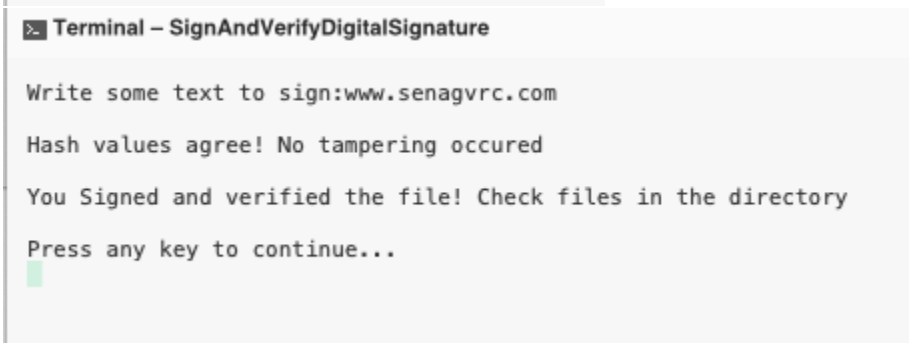
```
        using(FileStream outStream=new FileStream(destFile, FileMode.Create))
        {
            byte[] hashValue = hmac.ComputeHash(inStream);
            inStream.Position = 0;
            outStream.Write(hashValue, 0, hashValue.Length);
            int bytesRead;
            byte[] buffer = new byte[1024];
            do
            {
                bytesRead = inStream.Read(buffer, 0, 1024);
                outStream.Write(buffer, 0, bytesRead);

            } while (bytesRead > 0);
        }
    }
}
return;
```

Compute the hash of the input file then reset

```
public bool VerifyFile(byte[] key,string sourceFile)
{
    bool err = false;
    using(HMACSHA256 hmac=new HMACSHA256(key))
    {
        byte[] storedHash = new byte[hmac.HashSize / 8];

        using (FileStream inStream = new FileStream(sourceFile, FileMode.Open))

            inStream.Read(storedHash, 0, storedHash.Length);


            byte[] computedHash = hmac.ComputeHash(inStream);


            for(int i = 0; i < storedHash.Length; i++)
            {
                if(computedHash[i] != storedHash[i])
                {
                    err = true;

                }
            }
        }
```

Compares the key in the source file with a new key created for the data, if the keys are compare the data has not been tampered with then initialize the keyed hash object, create an array to hold the keyed hash value read from the file and FileStream for the source file. Read in the storedHash, compute the hash of the remaining contents of the file. The stream is properly positioned at the beginning of the content, immediately after the stored hash value. Compare the computed hash with the stored value.

```
            }
            if (err)
            {
                WriteLine(Environment.NewLine + "Hash values differ!Signed file has been tampered with");
                return false;
            }
            else
            {
                WriteLine(Environment.NewLine + "Hash values agree! No tampering occured");
                return true;

            }
        }

    }
}
```



4) AES Encryption

General steps of AES code is seen below

- Key generation
- İnitialization vector creation
- Encryption of cipher
- Writing to file
- Decryption of cipher
- Writing to file again

```java
//key generation part
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init( keysize: 128);
SecretKey symmetricKey_128 = keyGenerator.generateKey();


//initialization vector part
byte[] iv = new byte[16];
SecureRandom srandom = new SecureRandom();
srandom.nextBytes(iv);
IvParameterSpec ivSpec = new IvParameterSpec(iv);

// define encryption cipher part
Cipher encryptCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
encryptCipher.init(Cipher.ENCRYPT_MODE, symmetricKey_128,ivSpec);
byte[] cipherText = encryptCipher.doFinal(imageFile.getBytes());
String encrypted_string = Base64.getEncoder().encodeToString(cipherText);
System.out.println("Initialization Vector for CBC 128: " + Base64.getEncoder().encodeToString(iv));

// write image
FileOutputStream fosEncrypt = new FileOutputStream( name: "encrypted/1MB_image_128bitKeyCBC.jpg");
byte[] imageArray = Base64.getDecoder().decode(encrypted_string);
fosEncrypt.write(imageArray);
File fileEncrypt = new File( pathname: "encrypted/1MB_image_128bitKeyCBC.txt");
BufferedWriter bw = new BufferedWriter(new FileWriter(fileEncrypt));
bw.write(encrypted_string);

//define decryption cipher part
Cipher decryptCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
decryptCipher.init(Cipher.DECRYPT_MODE, symmetricKey_128,ivSpec);
byte[] plainText = decryptCipher.doFinal(Base64.getDecoder().decode(encrypted_string));
String decrypted_string = new String(plainText, charsetName: "utf-8");

//write image
FileOutputStream imageOutFile = new FileOutputStream( name: "decrypted/1MB_image_128bitKeyCBC.jpg");
byte[] decodedImageByteArray = Base64.getDecoder().decode(decrypted_string);
imageOutFile.write(decodedImageByteArray);
```

4a) 128 bits CBC mode

```
Initialization Vector for CBC 128: ldxgHP/50PxkfbVEUkNR9g==
Elapsed time : 426

Process finished with exit code 0
```

```
Initialization Vector for CBC 128: yg/lR8KqYMdYRr9ZudJ1Nw==
Elapsed time : 419

Process finished with exit code 0
```

As you see with different initialization vector we can get same encrypted and decrypted image

4b) 256 bits CBC mode

```
Initialization Vector for CBC 256: 86Kn/ZVnoXVcnmsvAYnwLQ==
Elapsed time : 368

Process finished with exit code 0
```

4c) 256 bits CTR mode

```
Elapsed time : 410

Process finished with exit code 0
```

Results of decryption part are same as seen below screenshot


1MB_image_128bitKeyCBC.jpg


1MB_image_256bitKeyCBC.jpg


1MB_image_256bitKeyCTR.jpg

As a comment, CTR mode seems (and expected) to be faster because parallel decryption and encryption can be done on the contrary of CBC mode and also can preprocess in advance

5)Message Authentication Codes

a) Generate a message authentication code (HMAC-SHA256) using any of the symmetric

keys.

```
enter string to hash:
take it easy
hashed string:fdf04e026531474b20d0989eb731f0fcc3fde3c2a63f193153ff9f53bf1f312b
```

Hash-based Message Authentication Code (MAC code, calculated using a cryptographic hash function). The resulting MAC code is a message hash mixed with the private key. Hashes have cryptographic properties: irreversible, collision resistant, etc. Hash_func can be any cryptographic hash function such as SHA256, SHA512, RIPEMD160, SHA3256, or BLAKE2.

```
using System;
using System.Security.Cryptography;
using System.Text;


namespace SHA256MessageAuthentication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("enter string to hash:");
            string input = Console.ReadLine();
            Console.WriteLine($"hashed string:{Hashing.ToSHA256(input)}");
            Console.ReadLine();
        }
    }
    public class Hashing
    {
        public static string ToSHA256(string s)
        {
            using var sha256 = SHA256.Create();
            byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(s));
            var sb = new StringBuilder();
            for (int i = 0; i < bytes.Length; i++)
            {
                sb.Append(bytes[i].ToString("x2"));
            }
            return sb.ToString();

        }
    }

}
```