Ira A. Fulton Schools of Engineering
Arizona State University

CSE 539: Applied Cryptography

Diffie-Hellman and Encryption Project (50 points)

Purpose

In this project, you will be using existing libraries to perform key exchange and encryption using a 256-bit key. To generate the key, you will implement the Diffie-Hellman algorithm. You will then use a symmetric encryption scheme (e.g. AES) to encrypt and decrypt any given data. In doing so, you will also learn how to work with very large numbers that are too large to store in a standard data type.

Objectives

Students will be able to

- Implement Diffie-Hellman Key Exchange
- Employ existing algorithms to encrypt and decrypt data
    - https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- Apply concepts from number theory to solve real-world problems

Technology Requirements

It is recommended to implement programming assignments using .NET Core 6.0. **You are free to use any other modern general-purpose programming language of your choice**. However, it is your responsibility to investigate the availability of existing libraries to successfully complete the project if you decide to go with other languages.

Information on how to install the .NET Core SDK can be found here: Install .NET Core SDK.
You can create a new project using this command: `dotnet new console --output project_name`.
You can also run your project by going into your project directory and using: `dotnet run`.

Here are some helpful links for this particular project

- BigInteger Class
- BigInteger.Pow Method
- BigInteger.ModPow Method
- AES.Create Method

- AESManaged.CreateDecryptor Method
- AESManaged.CreateEncryptor Method

Directions

The first part of this assignment involves the creation of a 256-bit key for performing encryption. You will be implementing the Diffie-Hellman key exchange protocol. Typically, this protocol involves two parties concurrently sharing values and generating a key. In this assignment, you will be given all necessary values immediately and will not be required to send values over any channel. In this way, you can perform calculations as a single party.

Here is a brief reminder of how Diffie-Hellman works:

1. Alice and Bob agree on values for g and N.
2. Alice randomly picks x and Bob randomly picks y.
3. Alice computes g^x mod N and sends it to Bob (call this gx).
4. Bob computes g^y mod N and sends it to Alice (call this gy).
5. Alice computes (gy)^x mod N, and Bob computes (gx)^y mod N. These two values are equivalent and are the shared secret keys.

The encryption algorithm you will be using in this project is AES (i.e., Rijndael encryption). You will be using the 256-bit key mode. In order to perform the encryption, you will also need an initialization vector (IV). In this exercise, you will employ a 128-bit IV passed via command line in hex. Here is the list of values you will receive from the command line arguments, in order:

1. 128-bit IV in hex
2. g_e in base 10
3. g_c in base 10
4. N_e in base 10
5. N_c in base 10
6. x in base 10
7. g^y mod N in base 10
8. An encrypted message C in hex
9. A plaintext message P as a string

Hint: You may not need all of these values to perform the computations.

Note: Since g and N are very large numbers, to facilitate debugging, the values for g and N are given as a pair of values, the exponent (e) and the constant (c). You can calculate the value for either using the formula (2^e) - c. For testing, use the following example:
If g_e = 250, g_c = 207, N_e = 256, and N_c = 189,
g = 1809251394333065553493296640760748560207343510400633813116524750123642
650417
N = 1157920892373161954235709850086879078532699846656405640394575840079131
29639747

To perform these calculations without encountering an overflow, you cannot use int (32-bit), long (64-bit), or decimal (96-bit). Instead, you must make use of C#'s BigInteger struct (or equivalent), which supports any arbitrarily long integer.

After calculating the key, your program must perform a decryption of the given ciphertext bytes and an encryption of the given plaintext string. Your program should output these values as a comma separated pair (the decrypted text followed by the encrypted bytes). Here is a full example in .NET:

Command:

```
dotnet run "A2 2D 93 61 7F DC 0D 8E C6 3E A7 74 51 1B 24 B2" 251 465 255 1311 2101864342

8995936589171851885163650660432521853327227178155593274584417851704581358902

"F2 2C 95 FC 6B 98 BE 40 AE AD 9C 07 20 3B B3 9F F8 2F 6D 2D 69 D6 5D 40 0A 75 45 80 45

F2 DE C8 6E C0 FF 33 A4 97 8A AF 4A CD 6E 50 86 AA 3E DF"

AfYw7Z6RzU9ZaGUIoPhH3QpfA1AXWxnCGAXAwk3f6MoTx
```

Expected output:

```
uUNX8P03U3J91XsjCqOJ0LVqt4I4B2ZqEBfX1gCGBH4hH, 3D E9 B7 31 42 D7 54 D8 96 12 C9
97 01 12 78 F7 A2 4F 69 1A FF F4 42 99 13 A1 BD 73 52 E5 48 63 33 7A 39 BF C5 25 AD 53 26
53 0D E4 81 51 D1 3E
```

Note: We've highlighted to show the input and expected outputs:

Inputs: initialization vector, g_e, g_c, N_e, N_c, x, g^y mod N, encrypted message C, plaintext P

Outputs: decrypted text, encrypted cipher

While this is an example code for you to check your implementation and share a screenshot of the results, your code will be tested with other test cases as well. Make sure to support command line arguments as such for testing.

There is a possibility that you may get a different answer since some libraries/environments use the little endian byte order as default when converting your key to a byte array whereas other languages use the big endian byte ordering. The example provided to you in the document uses a little endian byte-array key. In your setup, if you have the provisions to convert the byte ordering, please do so. However, both answers will be considered correct and you may feel free to submit the encrypted/decrypted texts obtained using either endianness.

Code Structure

In summary, your task is to implement three essential functions: ***calculateSharedKey, encrypt***, and ***decrypt***. Below are the required function definitions in C#. While you have the flexibility to choose any programming language, it's essential that your code contains function definitions provided below.

```
public BigInteger calculateSharedKey(int g_e, int g_c, int N_e, int N_c, int x,
BigInteger gy_modN){
    // Implement the method
    // return key;
)

public byte[] encrypt(string plaintext, BigInteger[] key, byte[] IV){
    // Implement the method
    // return ciphertext;
}

public string decrypt(byte[] ciphertext, BigInteger[] key, byte[] IV){
    // Implement the method
    // return plaintext;
}
```

Submission Directions for Project Deliverables

- Your code
- Screenshots of running the example command provided above along with inputs and outputs.
- A README file for setup and steps necessary to run your program.
- Compress your project folder along with the screenshots into a .zip archive. and submit that file. Please name your submission Firstname_Lastname.zip