

ENCRYPTO-DECRYPT

BY DIFFE-HELLMAN METHOD

SURUCHI PATIL

JSSATE-B

ABHA ANAND

RVCE

HARSHITHA .M

BIT

NAYANASHRI.M

BMSCE

SARANYA .T

CMR UNIVERSITY

ACKNOWLEDGEMENT

The knowledge and satisfaction that accompany the successful completion of any task would be incomplete without mention of people who made it possible, whose guidance and encouragement crowned my effort with success. We would like to thank all and acknowledge the help we have received to carry out this Internship Project.

We would like to convey our thanks to **Mr.Supreeth**, Tequed Labs for being kind enough to provide the necessary support to carry out the internship project.

We would be most humbled to mention the enthusiastic influence provided by the **Mr..Dinesh Parangathan**, Hackup Technology , on the project for their ideas, time to time suggestions for being a constant guide and co-operation showed during the venture and making this project a great success.

We would also take this opportunity to thank my friends and family for their constant support and help. We are very much pleased to express our sincere gratitude to the friendly co-operation showed by all the staff members of Tequed Labs.

INDEX

1. Problem Statement and Description.....	4
2. Proposed Solution.....	5
3. Visualization of Diffie-Hellman.....	6
4. Literature survey.....	7
5. Requirement specification.....	7
6. Methodology.....	8
7. Testing.....	10
8. Future Enhancement.....	14
9. References.....	15
10. Appendix.....	16

PROBLEM STATEMENT

1. Traditionally, symmetric encryption suffered one enormous shortcoming – it was necessary for either the sender or the recipient to create a key and then send it to the other party. While the key was in transit, it could be stolen or copied by a third party who would then be able to decrypt any cipher texts encrypted with that key.
2. Another problem is that a large number of key pairs are needed between communicating parties. This quickly becomes difficult to manage the more there are. This can be calculated as $n(n-1)/2$ where n is the number of communicating parties.

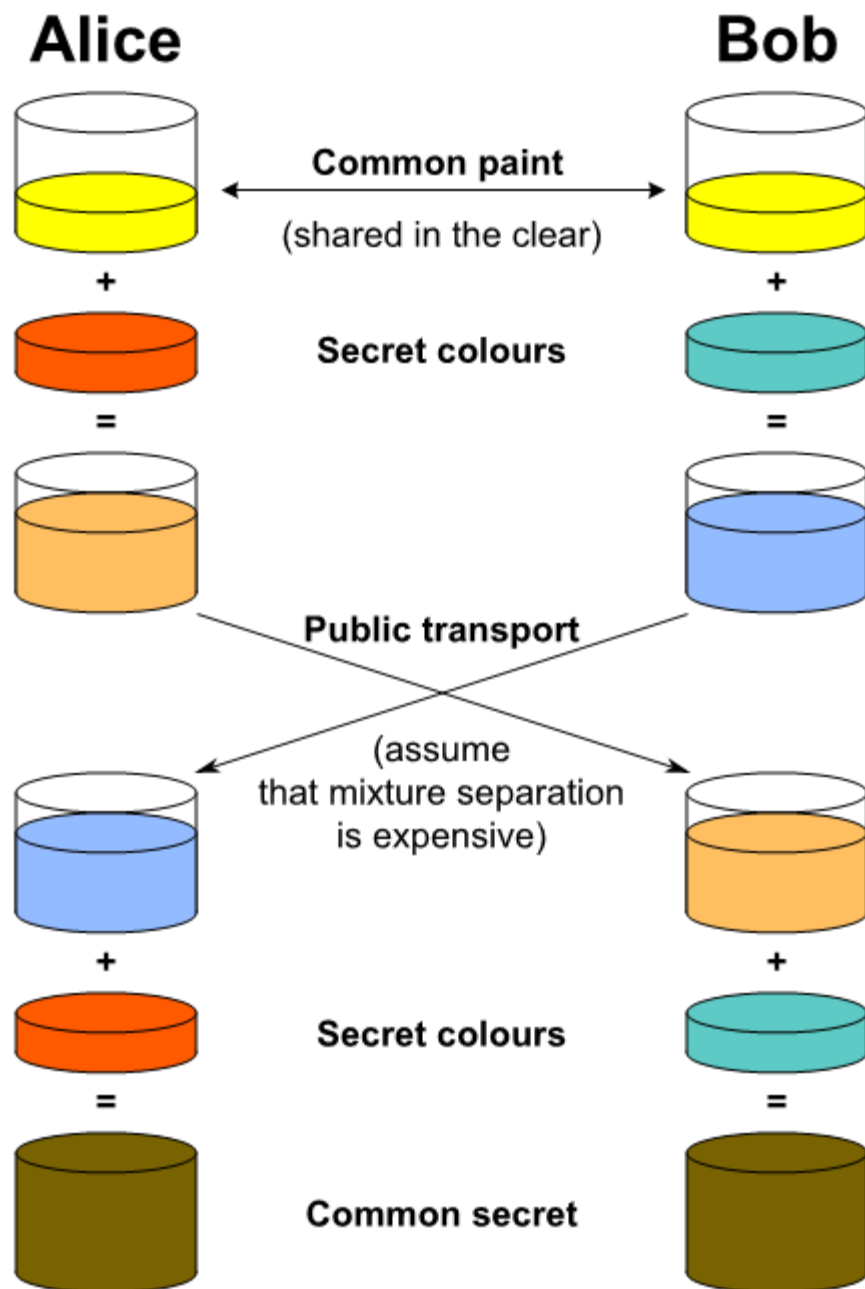
DESCRIPTION

We create a cryptographic class which has two methods - encrypt and decrypt. These two methods will allow us to exchange public key with the other party and decrypt the secret messages with private key. The secret message will be encrypted using standard AES encryption.

PROPOSED SOLUTION

- Step 01 - Create a Class Library
- Step 02 - Add fields
- Step 03 - Add a constructor
- Step 04 - Expose Public Key and IV
- Step 05 - Create an Encrypt method
- Step 06 - Create a Decrypt method
- Step 07 - Dispose unmanaged resources
- Step 08 - Create a test class

VISUALIZATION OF DIFFIE-HELLMAN



Litrature survey

Eun-Jun Yoon et al proposed an efficient DiffieHellman-MAC key exchange scheme providing same securities as proposed by Jeong et al. who proposed a strong Diffie-Hellman- DSA key exchange scheme providing security against session state reveal attacks as well as forward secrecy and key independence. The proposed scheme is based on the keyed MAC hash function to provide efficiency. Then, they proposed a strong DiffieHellman-DSA key exchange scheme providing security against session state reveal attacks as well as forward secrecy and key independence. Emmanuel Bresson et al. has investigated the Group Diffie-Hellman protocols for authenticated key exchange(AKE) are designed to provide a pool of players with ashared secret key which may later be used, for example, toachieve multicast message integrity. Over the years, severalschemes have been offered.

REQUIREMENT SPECIFICATION

- **Software specifications:**

Visual studio

.NET code

METHODOLOGY

Step 01 - Create a Class Library

Open Visual Studio and go to "*File > New > Project*" and select "*Class Library*". Give your project a name (e.g. *SecureKeyExchange*) and click "OK". After your project is created, rename the "Class1.cs" file to "DiffieHellman.cs".

Step 02 - Add fields

We need to add three fields; one that contains a reference to the **Aes**-class, the second field to store a reference to the **ECDiffieHellmanCng**-class and the last fields to store our public key. The **Aes**-reference will be used to encrypt/decrypt the messages. The **ECDiffieHellmanCng**-reference will be used to create a derived key between the two parties.

Step 03 - Add a constructor

Our constructor should now initialize these fields:

Once the **ECDiffieHellmanCng** instance has been initialized, we can set our **publicKey** field to the **PublicKey** of the **ECDiffieHellmanCng** instance. We are going to send this public key along with the secret message to the other party.

Step 04 - Expose Public Key and IV

Let's expose both our public key and IV through properties. Add the following properties respectively:

These properties will be sent to the other party to decrypt the secret message using their own private key.

Step 05 - Create an Encrypt method

We are going to create a method that takes the public key of the other party as well as the secret message to encrypt.

We will use the other party's public key to generate a *derived key* (see "Common secret" in the paint analogy above) which will be used to encrypt the message. Add the **Encrypt** function.

Now our message is encrypted and we can send it to the other party. But first need to add a function to decrypt this secret message.

Step 06 - Create a Decrypt method

Our **Decrypt** function will take in 3 parameters:

The public key and IV of the other party as well as the secret message. Let's add the function:

We can now decrypt the secret message.

Step 08 - Create a test class

Right click on the solution and select "*Add > New Project > Unit Test Project*" and give your project a name (e.g. "*SecureKeyExchange.Tests*"). Rename your "*UnitTest1.cs*" to "*DiffieHellmanTests.cs*" and add a reference to the "*SecureKeyExchange*" project. To do this, right-click on the "*References*" node under the test project and select "*Add Reference > Projects > Select the project > OK*".

We can now add a breakpoint and debug our test (press Ctrl+R, Ctrl+A) to see the results:

TESTING

Diffie-Hellman key exchange is a specific method of securely exchanging cryptographic keys over a public channel. This algorithm allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This secret key can then be used to encrypt subsequent communications.

Cryptographic explanation from wikipedia to generate common key between two users. The simplest and the original implementation of the protocol use the multiplicative group of integers modulo p , where p is prime, and g is a primitive root modulo p . Here is an example of the protocol, with non-secret values in blue, and secret values in red.

Alice and Bob agree to use a modulus $p = 23$ and base $g = 5$ (which is a primitive root modulo 23).

Alice chooses a secret integer $a = 6$, then sends Bob $A = g^a \bmod p = 5^6 \bmod 23 = 8$

Bob chooses a secret integer $b = 15$, then sends Alice $B = g^b \bmod p = 5^{15} \bmod 23 = 19$

Alice computes $s = B^a \bmod p$

$s = 19^6 \bmod 23 = 2$ (after computing alice got 2 as secret key)

Bob computes $s = A^b \bmod p$

$s = 8^{15} \bmod 23 = 2$ (Bob also got same key)

Alice and Bob now share a secret (the number 2).

Both Alice and Bob have arrived at the same value

Diffie-Hellman Key Exchange

Bob and Alice know and have the following :
 $p = 23$ (a prime number) $g = 11$ (a generator)

Alice

Alice chooses a secret random number $a = 6$

Alice computes : $A = g^a \bmod p$
 $A = 11^6 \bmod 23 = 9$

Alice receives $B = 5$ from Bob

Secret Key = $K = B^a \bmod p$
 $K = 5^6 \bmod 23 = 8$

Bob

Bob chooses a secret random number $b = 5$

Bob computes : $B = g^b \bmod p$
 $B = 11^5 \bmod 23 = 5$

Bob receives $A = 9$ from Alice

Secret Key = $K = A^b \bmod p$
 $K = 9^5 \bmod 23 = 8$

The common secret key is : 8

N.B. We could also have written : $K = g^{ab} \bmod p$

[Home](#)
[View Pseudo Code](#)
[Run Diffie-Hellman](#)
[Contact Us](#)

```

//read prime no
String prime=request.getParameter("t1").trim();
//alice secret value
String alice=request.getParameter("t2").trim();
//bob secret value
String bob=request.getParameter("t3").trim();
//generating primitive root for given prime no
String primitive_root = PrimitiveRoot.getG(Integer.parseInt(prime));
    
```

```

//bob secret value
String bob=request.getParameter("t3").trim();
//generating primitive root for given prime no
String primitive_root = PrimitiveRoot.getG(Integer.parseInt(prime));
//convert all primitive root in array
String values[] = primitive_root.split(",");
//prime no
BigInteger prime_no = new BigInteger(prime);
//choosing first value from primitive root array as gbase
BigInteger gbase = new BigInteger(values[0]);
//Alice, compute A = gbase.pow(alice_pri_key).mod(p) and send to bob
BigInteger A = gbase.pow(Integer.parseInt(alice)).mod(prime_no);
//Bob, compute B = gbase.pow(bob_pri_key).mod(p) and send to alice
BigInteger B = gbase.pow(Integer.parseInt(bob)).mod(prime_no);
//now alice compute B.pow(alice_pri_key).mod(p) to generate common secret key
BigInteger alice_compute = B.pow(Integer.parseInt(alice)).mod(prime_no);
//now bob compute A.pow(bob_pri_key).mod(p) to generate common secret key
BigInteger bob_compute = A.pow(Integer.parseInt(bob)).mod(prime_no);
//result printing
StringBuilder sb = new StringBuilder();
sb.append("Prime No : "+prime_no);

```

```

//prime no
BigInteger prime_no = new BigInteger(prime);
//choosing first value from primitive root array as gbase
BigInteger gbase = new BigInteger(values[0]);
//Alice, compute A = gbase.pow(alice_pri_key).mod(p) and send to bob
BigInteger A = gbase.pow(Integer.parseInt(alice)).mod(prime_no);
//Bob, compute B = gbase.pow(bob_pri_key).mod(p) and send to alice
BigInteger B = gbase.pow(Integer.parseInt(bob)).mod(prime_no);
//now alice compute B.pow(alice_pri_key).mod(p) to generate common secret key
BigInteger alice_compute = B.pow(Integer.parseInt(alice)).mod(prime_no);
//now bob compute A.pow(bob_pri_key).mod(p) to generate common secret key
BigInteger bob_compute = A.pow(Integer.parseInt(bob)).mod(prime_no);
//result printing
StringBuilder sb = new StringBuilder();
sb.append("Prime No : "+prime_no);
sb.append("Primitive Root Modulo Of "+prime_no+" : "+primitive_root);
sb.append("Chosen Primitive Root (g) : "+gbase);
sb.append("Alice Send Bob : "+A);
sb.append("Bob Send Alice : "+B);
sb.append("Alice Compute Secret Key : "+alice_compute);
sb.append("Bob Compute Secret Key : "+bob_compute);

```

Diffie-Hellman Key Exchange



Alice

Bob and Alice know and have the following :
 $p = 23$ (a prime number) $g = 11$ (a generator)

Alice chooses a secret random number $a = 6$

Alice computes : $A = g^a \text{ mod } p$
 $A = 11^6 \text{ mod } 23 = 9$

Alice receives $B = 5$ from Bob

Secret Key = $K = B^a \text{ mod } p$

$K = 5^6 \text{ mod } 23 = 8$



Bob

Bob chooses a secret random number $b = 5$

Bob computes : $B = g^b \text{ mod } p$
 $B = 11^5 \text{ mod } 23 = 5$

Bob receives $A = 9$ from Alice

Secret Key = $K = A^b \text{ mod } p$

$K = 9^5 \text{ mod } 23 = 8$

The common secret key is : 8

N.B. We could also have written : $K = g^{ab} \text{ mod } p$

[Home](#)
[View Pseudo Code](#)
[Run Diffie-Hellman](#)
[Contact Us](#)

Secret key Computation Screen

Enter Prime No

Alice Secret No

Bob Secret No

Diffie-Hellman Key Exchange



Alice

Bob and Alice know and have the following :
 $p = 23$ (a prime number) $g = 11$ (a generator)


Alice chooses a secret random number $a = 6$

Alice computes : $A = g^a \text{ mod } p$
 $A = 11^6 \text{ mod } 23 = 9$

Alice receives $B = 5$ from Bob

Secret Key = $K = B^a \text{ mod } p$

$K = 5^6 \text{ mod } 23 = 8$



Bob

Bob chooses a secret random number $b = 5$

Bob computes : $B = g^b \text{ mod } p$
 $B = 11^5 \text{ mod } 23 = 5$

Bob receives $A = 9$ from Alice

Secret Key = $K = A^b \text{ mod } p$

$K = 9^5 \text{ mod } 23 = 8$

The common secret key is : 8

N.B. We could also have written : $K = g^{ab} \text{ mod } p$

Prime No : 23

Primitive Root Modulo Of 23 : 5,7,10,11,14,15,17,19,20,21

Chosen Primitive Root (g) : 5

Alice Send Bob : 8

Bob Send Alice : 16

Alice Compute Secret Key : 4

Bob Compute Secret Key : 4

FUTURE ENHANCEMENT

In spite of the fact that Diffie-Hellman is a public key algorithm, specialists say it doesn't scale well for the future. As of right now it is expressed that Diffie-Hellman keys shorter than 900 bits are not sufficiently secure. To make Diffie-Hellman keys, which now can go to 1,024 bits, secure for the following 10 to 20 years, associations would need to grow to key lengths of no less than 2,048 bits, as per Stephen Kent, chief researcher at BBN Technologies. In the long run, key sizes would need to grow to 4,096 bits. Researchers from the NIST's security technology group expect, that it is exceptionally conceivable, that Diffie-Hellman will be broken inside of 10 years or somewhere in the vicinity. The cryptographic security standards utilized as a part of public-key infrastructures, RSA and Diffie-Hellman, were presented in the 1970s. And although they haven't been broken, their time could be running out. That is one reason the National Security Agency needs to move to elliptic-curve cryptography (ECC) for cyber security. ECC, a complex mathematical algorithm used to secure information in transit, may replace Diffie-Hellman in light of the fact that it can give much more prominent security at a smaller key size. ECC takes less computational time and can be utilized to secure data on smaller machines, including mobile phones, smart cards and wireless devices.

REFERENCES

- *Cryptography, Network Security, and Cyber Laws* by Bernard L. Menezes And Ravinder Kumar.
- Keith Palmgren, *CISSP* (2006, August). *Diffie-Hellman Key Exchange - A Non-Mathematician's Explanation*
http://academic.regis.edu/cias/ia/palmgren_-_diffie-hellman_key_exchange.pdf
- Hellman M. (2002, May). *An Overview of Public Key Cryptology*.
<http://www-ee.stanford.edu/~hellman/publications/31.pdf>
- Eun-Jun Yoon and Kee-Young Yoo, “An Efficient Diffie-HellmanMAC Key Exchange Scheme”, 2009 Fourth International Conference on innovative Computing, Information and Control.

APPENDIX

```
using System;

using System.IO;

using System.Security.Cryptography;

using System.Text;


namespace SecureKeyExchange
{
    public class DiffieHellman : IDisposable
    {
        #region Private Fields

        private Aes aes = null;

        private ECDiffieHellmanCng diffieHellman = null;


        private readonly byte[] publicKey;

        #endregion


        #region Constructor

        public DiffieHellman()
        {
            this.aes = new AesCryptoServiceProvider();
```



```

        this.diffieHellman = new ECDiffieHellmanCng
        {
            KeyDerivationFunction =
ECDiffieHellmanKeyDerivationFunction.Hash,
            HashAlgorithm = CngAlgorithm.Sha256
        };

        // This is the public key we will send to the other party
        this.publicKey = this.diffieHellman.PublicKey.ToByteArray();
    }
#endregion

#region Public Properties
public byte[] PublicKey
{
    get
    {
        return this.publicKey;
    }
}

public byte[] IV
{

```

```

        get
        {
            return this.aes.IV;
        }
    }
#endregion

#region Public Methods
public byte[] Encrypt(byte[] publicKey, string secretMessage)
{
    byte[] encryptedMessage;

    var key = CngKey.Import(publicKey, CngKeyBlobFormat.EccPublicBlob);
    var derivedKey = this.diffieHellman.DeriveKeyMaterial(key); // "Common
secret"

    this.aes.Key = derivedKey;

    using (var cipherText = new MemoryStream())
    {
        using (var encryptor = this.aes.CreateEncryptor())
        {
            using (var cryptoStream = new CryptoStream(cipherText, encryptor,
CryptoStreamMode.Write))

```

```

        {
            byte[] ciphertextMessage =
Encoding.UTF8.GetBytes(secretMessage);

            cryptoStream.Write(ciphertextMessage, 0,
ciphertextMessage.Length);
        }
    }

```

```

        encryptedMessage = cipherText.ToArray();
    }

```

```

    return encryptedMessage;
}

```

```

public string Decrypt(byte[] publicKey, byte[] encryptedMessage, byte[] iv)
{
    string decryptedMessage;

    var key = CngKey.Import(publicKey, CngKeyBlobFormat.EccPublicBlob);
    var derivedKey = this.diffieHellman.DeriveKeyMaterial(key);

    this.aes.Key = derivedKey;
    this.aes.IV = iv;
}

```

```

        using (var plainText = new MemoryStream())
        {
            using (var decryptor = this.aes.CreateDecryptor())
            {
                using (var cryptoStream = new CryptoStream(plainText, decryptor,
CryptoStreamMode.Write))
                {
                    cryptoStream.Write(encryptedMessage, 0,
encryptedMessage.Length);
                }
            }

            decryptedMessage = Encoding.UTF8.GetString(plainText.ToArray());
        }

        return decryptedMessage;
    }

#endregion

#region IDisposable Members

    public void Dispose()
    {
        Dispose(true);
    }

```

```

        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (this.aes != null)
                this.aes.Dispose();

            if (this.diffieHellman != null)
                this.diffieHellman.Dispose();
        }
    }
    #endregion
}

```

TEST CODES:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```
namespace SecureKeyExchange.Tests
```

```
{
    [TestClass]
```

```

public class DiffieHellmanTests
{
    [TestMethod]
    public void Encrypt_Decrypt()
    {
        string text = "Hello World!";

        using (var bob = new DiffieHellman())
        {
            using (var alice = new DiffieHellman())
            {
                // Bob uses Alice's public key to encrypt his message.
                byte[] secretMessage = bob.Encrypt(alice.PublicKey, text);

                // Alice uses Bob's public key and IV to decrypt the secret
                message.
                string decryptedMessage = alice.Decrypt(bob.PublicKey,
                secretMessage, bob.IV);
            }
        }
    }
}

```