Assignment 2
Message Authentication & Diffie-Hellman Key Exchange

L. Towell,
Student No: 201383538

December 10, 2019

Code repository: COMP522-Assignment 2

1 Comparison of methods for message authentication

1.1 Hash-functions

Hash functions are used in order to attempt to ensure to both the sender and the receiver that a message is authentic and has not been manipulated or interfered with in any way. Hash functions are typically used when a secret is not shared between the sender and the receiver.
The process of using a hash function is the following:
- An unencrypted message is sent from the sender to the receiver with a tag (hash) which has been generated by the sender.
- The receiver then received the message and the tag and in order to establish if the message has been tampered with they generate their own hash using the message from the sender.
- Once the receiver has generated their own hash they then compare the hash they generated to the hash they received as the tag. If the hash generated by the receiver doesn't match the tag hash then the message has been tampered with and the contents cant be trusted however if the hash and tag hash do match then the message hasnt been manipulated and can be trusted to be original.

Hash functions are typically referred to as one way hashes. This is because it is easy to generate a hash using a message however it is very difficult to generate the message from the hash. The tag hash that is generated using a hash function algorithm e.g. the SHA1 typically produces a fixed length hash which is not dependent on the size of the message being sent.

Appendix A is an example of the SHA1 hash function which takes a string input and digests the input using the SHA1 algorithm to produce a byte array hash.
Input = "This is a string"
Output in the form of a hash using the SHA1 algorithm = f72017485fbf6423499baf9b240daa14f5f095a1

The advantage of hash functions are that they are very simple to compute and are an effective way of telling if a message is genuine or if it has been manipulated.

The disadvantage of Hash functions are that since the message is unencrypted, the content of the message is not secure so anyone can read the original message and as long as they dont change the hash then the hash will still match the original message. The hash is also easy to replace so the message could be read and manipulated by an illicit party who could then generate a new hash tag and send the message on to the original recipient with a new content and a new hash which will match when they attempt to compare generated hashes. Hash functions also do not satisfy the weak and strong collision policys so it is possible you could generate a duplicate hash. A further issue with the SHA1 algorithm has been identified that it has a mathematical weakness which makes the outputs of the algorithm vulnerable to decryption.

1.2 RSA + SHA1 method

The methodology discussed in this section is the combination of the hashing mechanism SHA1 with the RSA algorithm which combines the hashing algorithm discussed previously in *Section 1.a.* and demonstrated in *Appendix A* together with the use of public and private keys in order to create a more secure message sharing methodology.

The combination of RSA and SHA1 hashing allows a sender (*Appendix B*) to generate a hash using a hash algorithm e.g. SHA1. Then this hash tag is encrypted using a private key which has been generated as part of the key pair to produce a ciphertext.

The ciphertext, plain text message and corresponding public key is sent to the verifier.

The verifier then uses the same hashing algorithm and the plain text message to generate their own hash. They also decrypt the ciphertext using the public key provided in the message. Once the verifier has their own hash and has the decrypted hash they can compare the values to see if they match. As in Hash functions if the hash tags do not match then the message is untrustworthy and needs to be discarded.

An example of how a tampered with message can be detected can be created using *Appendix B* and *Appendix C*. In the experiment below I have generated a message containing the string "This is a sample message" which has then been encrypted and sent to the receiver class. However on the way to the varifier the message has been intercepted and the plain text has been changed to "This is not a sample message". As you can see from the screen shot below because the message has been changed the hash no longer matches and the message cannot be trusted.

```
This is a sample message
sender digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
cipher: 747e277a0eb719f7e08181dff5ddeae40bbd4844fd34f2bbba0766f1aa8e2d282756a57fdf83d784ee965c57840a59dc8822b56c760d10a0f24507e421494366
Message sent by sender: This is a sample message
message received by verifier: This is a sample message
digest values match
verifier hash: 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
sender hash  : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
```

Figure 1: Output of non intercepted message

```
sender digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
cipher: 7126cf6f4058c503da3874f836cf559413c704fc62be0897e2ffeeaef3a050b9f1d9262c92b0aae93352a64221065ee77f94cb76e10c75879c52ba4e6365b1b8
Message sent by sender: This is a sample message
message received by verifier: This is not a sample message
hashed values dont match this message is not trustworthy.
verifier hash: 5f35515a716b5fa74b9cde257badc4b67c21fcce
sender hash  : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
```

Figure 2: Output of intercepted message detection

1.3 DSA method

The DSA methodology is Similar to the RSA + Hashing methodology however the DSA method does not encrypt the message hash like in the RSA methodology. The DSA method available in JPA takes the original message from the sender, creates a hash and signs the message with a private key in order to create a signature that is unique to the message being sent. The message is then sent to the user which contains the original message, the signature hash that was generated by the sender and the public key of the sender.

When the message is received by the verifier they are able to take the original message and generate a message hash. The received then takes their message hash and the senders public key and verifies that the public key being supplied is a match to the private key that was used to sign the content of the message. If the verify function of the DSA algorithm comes back as "True" then the message that has been received was the message that was sent and is genuine. If either the signature or the message has changed then the verify function will return false and the message can not be trusted as with all other message authentication methods already discussed.

The figures below show the results of experiments that have been ran on the DSA algorithm using the sender and verifier programs (*Appendix D & Appendix E*).

figure 3. Shows that when the message is not intercepted and the original message and the signed hash are original then the DSA algorithm is able to run as expected and verify that the message has not been manipulated, meaning the message contents can be trusted.

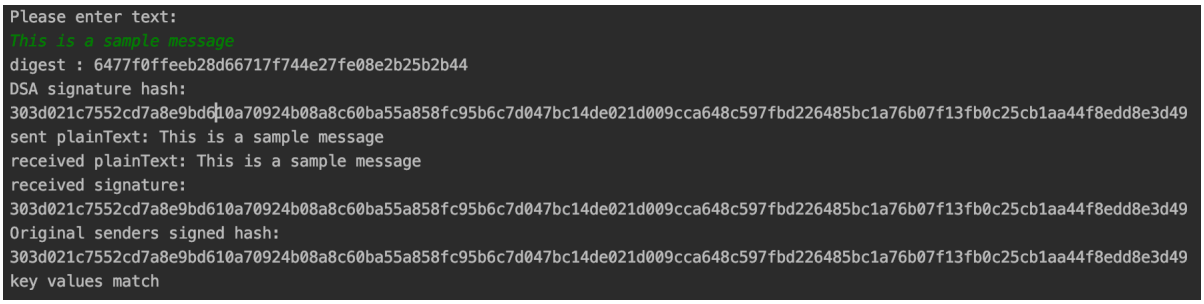

Figure 3: Output of non intercepted message

figure 4. Shows that when the message has been manipulated then the hash that is produced is different and the comparison of keys at the verify function is incompatible therefor the message cannot be trusted.

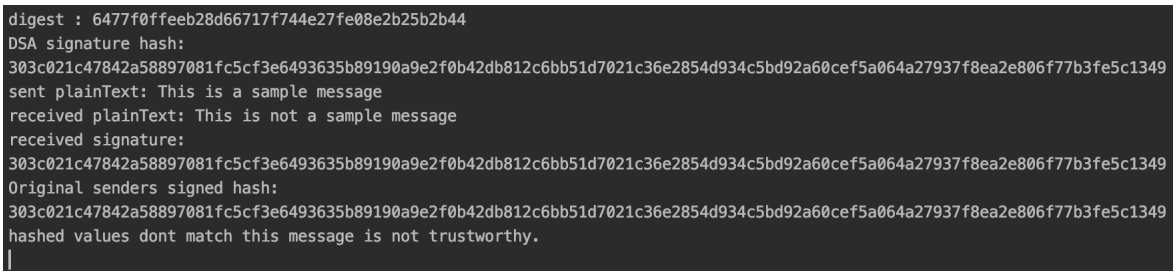

Figure 4: Output of intercepted message detection

figure 5. Shows that even if the original message is the same but the signed hash has been manipulated then the DSA algorithm throws and exception because of the incompatible signature and some form of manipulation has taken place therefor the message cannot be trusted.

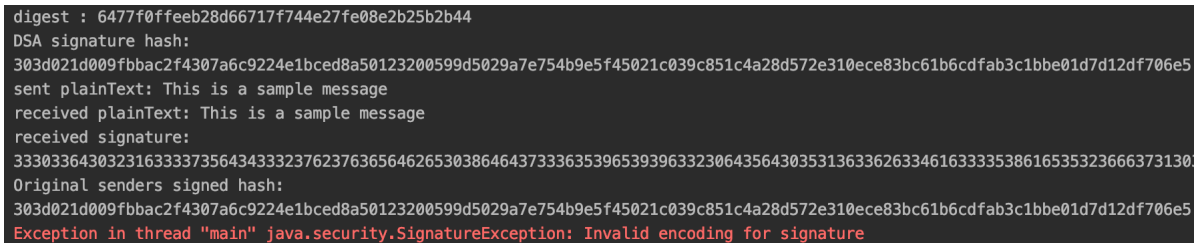

Figure 5: Output of intercepted message with bad signature detection

The advantages of the DSA algorithm are that all keys are generated locally at each sender therefor the keys can easily be changed if a key is believed to have been compromised which enhances the security of communications. Another advantage of the DSA algorithm is that it is relatively easy to implement especially if using the JPA toolset.

A disadvantage of the algorithm is that it can be computationally expensive especially if generating new keys for every message sent.

1.4 HMAC-SHA256 method

MAC(Message Authentication Code) is another method of method authentication, the HMAC-SHA256 method is very similar to the Hash functions that are discussed above in *Section 1.A*. The purpose of a MAC address like all method authentication methods is to ensure that when messages are passed between 2 parties they are passed without interference of manipulation. If a message has been interfered with or manipulated then the receiver should be able to identify this.

- Mac functions assume that a shared key has already been generated and shared between sender and receiver.

- The sender then takes their plainText message and generates a MAC address using the shared private key. This MAC address is then used as a tag like in the hashing functions already discussed. The sender then sends the MAC address, the original message to the receiver.

- The receiver is then able to take the message that has been received, the private key which was already shared and generate a mac address themselves.

- The receiver then compares the received mac addres and the one they generated to see if the message has been interfered with. If the mac address has been manipulated then the macs wont match and the message is untrustworthy. Likewise if the message has been changed then the hash generated by the receiver wont match the hash from the sender and the message is untrustworthy. As has previously been discussed in the previous sections the only way for the messages to be considered trustworthy is if the mac addresses generated by both sender and receiver match.

An advantage of MAC is that like hash functions the MAC address could be considered as a one way hash function because MAC functions are not reversible. However a disadvantage is that the MAC algorithm requires that the sender and receiver share a secret key. If the secret key becomes compromised then they will both need to share a new secret together which can introduce an overhead.

2 Diffie-Hellman with 4 parties

2.1 Design for the Diffie-Hellman Protocol

Following the principles of the Diffie Hellman Protocol means that in order to read the messages that are being transmitted every party needs to know the public key of all of the other members in the network in order to generate a shared secret key.

The process is largely the same when more parties are added however because more parties have been added it means that more keys need to be generated and shared. Each party starts the process by using the shared prime number (q) and the shared primitive root (α of q).

Each key then generates their public keys using an random integer that they have generated.

$Y_A = \alpha^{X_A}$ mod q
$Y_B = \alpha^{X_B}$ mod q
$Y_C = \alpha^{X_C}$ mod q
$Y_D = \alpha^{X_D}$ mod q

The above keys are then shared among all members of the network and then then the following calculations are performed in order to calculate the shared secret Keys

A calculates K = $(Y_{BCD})^{X_A}$ mod q
B calculates K = $(Y_{ACD})^{X_B}$ mod q
C calculates K = $(Y_{ABD})^{X_C}$ mod q
D calculates K = $(Y_{ABC})^{X_D}$ mod q

Figure 6. is an example of the stages that each member of the network goes through in order to finally reach a shared secret key.

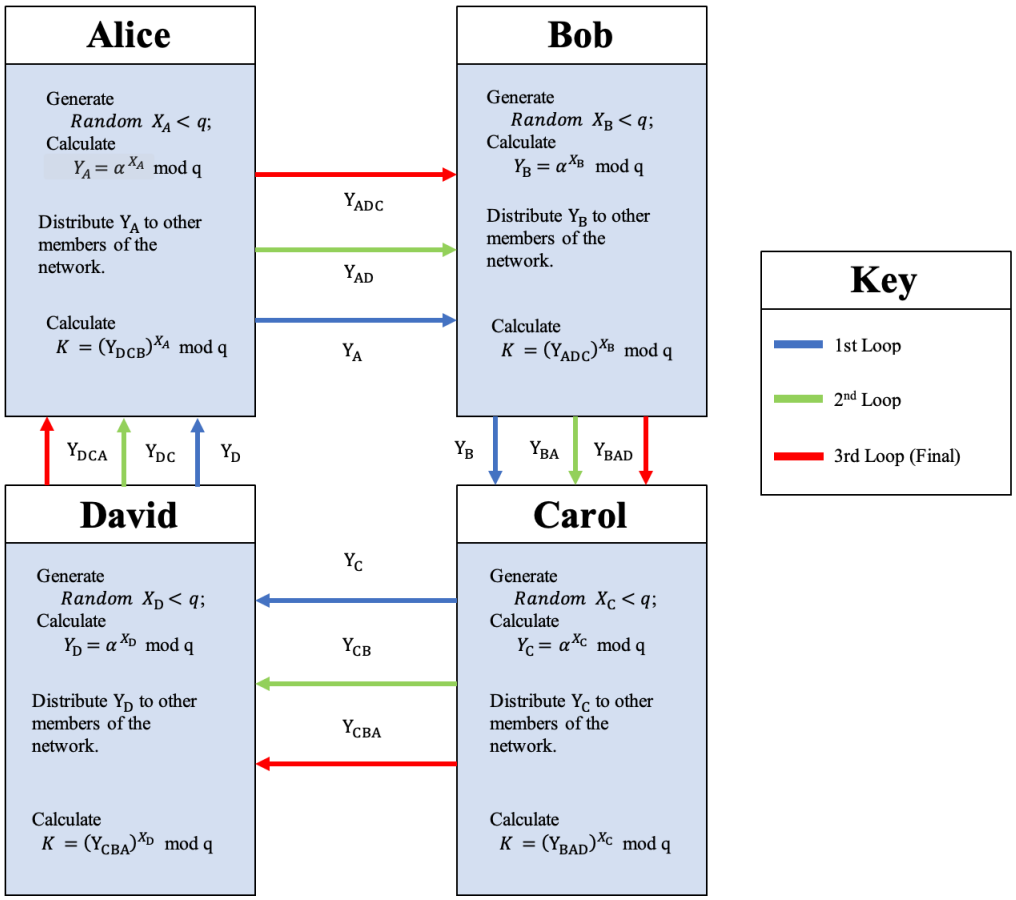

Figure 6: Diagram of how keys are passed between members of network.

2.2 Implementation of the Protocol with 4 Parties

As discussed above in section 2.1 and can be seen from the code in *Appendix F.* I have had to exchange all of the keys amongst each of the members in the network. I have done this through multiple iterations (phases) of the network in order for everyone within the network to have all of the keys needed to successfully generate a shared secret key for decoding messages that are sent. *figure 6.* shows the iteration cycle of how keys are exchanged over multiple iterations.

Table 1. below details how the keys are gathered within the program. In iteration 0 the only keys that each members have are their own however as we iterate around the network the keys are gathered until every member has every other members public keys. Once every member has all of the other members public keys they are able to calculate the shared secret for the decoding of all messages. This is displayed in the code output via printing the secrets generated by each individual member and then comparing the keys. If the keys do not match then the program with throw an error however if all keys match the output should be the same as *Appendix E.*

	Public Keys held by network members			
N	Alice	Bob	Carol	David
0	A	B	C	D
1	AD	BA	CB	DC
2	ADC	BAD	CBA	DCB
3	ADCB	BADC	CBAD	DCBA

Appendices

A SHA1 Message Digestor

Note: This small digestor is used within the example for RSA Encryption with SHA-1 Hashing (*Appendix B & Appendix C*).

```
import java.security.MessageDigest;

/**
 *
 */
public class MessageDigestor {
    public static void main(String[] args) throws Exception {
    }

    public static byte[] messageDigest(String input) throws Exception {

        java.security.MessageDigest hash = java.security.MessageDigest.getInstance("SHA1");

        hash.update(Utils.toByteArray(input));

        return hash.digest();
    }
}
```

B RSA Encryption + SHA1 Hash Sender Program

Note: This program is the code for the sender section of the RSA + SHA1 method. This method calls its counterpart verifier program *Appendix C* passing in the message sent to the verifier.

```
import javax.crypto.Cipher;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.util.Scanner;

public class Sender {
    public static void main(String[] args) throws Exception {

        // declare requirements
        Scanner input = new Scanner(System.in);
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        byte[] hashedInput = "".getBytes();
        SecureRandom random = new SecureRandom();
        KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

        // take the input text
        System.out.println("Please enter text:");
        String inputPlainText = input.nextLine();

        // retrieve the hash of the input text or if that fails then exit the program.
        try {
            hashedInput = MessageDigestor.messageDigest(inputPlainText);
        } catch (Exception e) {
            System.out.println("error: " + e);
            System.exit(1);
        }
        System.out.println("sender digest : " + Utils.toHex(hashedInput));

        // generate the keys
        generator.initialize(512, random);

        // keys
        KeyPair pair = generator.generateKeyPair();
        Key pubKey = pair.getPublic();
        Key privKey = pair.getPrivate();

        // encrypt the hashed input and print out
        cipher.init(Cipher.ENCRYPT_MODE, privKey);
        byte[] cipherText = cipher.doFinal(hashedInput);
        System.out.println("cipher: " + Utils.toHex(cipherText));
        System.out.println("Message sent by sender: " + inputPlainText);


        // generate the message
        Message senderMessage = new Message(inputPlainText, cipherText, pubKey);

        // send the message to the verifier
        Verifier.verifyMessage(senderMessage);

    }
}
```

C RSA Encryption + SHA1 Hash Verifier Program

```
import javax.crypto.Cipher;

public class Verifier {
    public static void main(String[] args) {

    }
    public static void verifyMessage (Message senderMessage) throws Exception{
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        byte[] verifierHashedInput = "".getBytes();

        //Decrypt the senders cipherText to get the hashedMessage using the publicKey
        cipher.init(Cipher.DECRYPT_MODE, senderMessage.getPubKey());
        byte[] plainText = cipher.doFinal(senderMessage.getHashedMessage());
        // Sender digest
        String senderDigest = Utils.toHex(plainText);

        //String interceptedMessage = "This is not a sample message";

        System.out.println("message received by verifier: " + senderMessage.getOriginalMessage());

        //calculates own hashed message
        try{
            verifierHashedInput =
                MessageDigestor.messageDigest(senderMessage.getOriginalMessage());
        }catch (Exception e ){
            System.out.println("error: "+ e);
            System.exit(1);
        }

        String verifierDigest = Utils.toHex(verifierHashedInput);

        //compares if the hashed value generated by the verifier using the original message
            matches the sender hash which was decrypted using the public key.

        if( verifierDigest.equals(senderDigest)){
            System.out.println("digest values match");
            System.out.println("verifier hash: " + verifierDigest);
            System.out.println("sender hash : " + senderDigest);
        } else {
            System.out.println("hashed values dont match this message is not trustworthy.");
            System.out.println("verifier hash: " + verifierDigest);
            System.out.println("sender hash : " + senderDigest) ;
        }

    }
}
```

D DSA Sender Program

```
import javax.crypto.Cipher;
import java.security.*;
import java.security.spec.DSAParameterSpec;
import java.util.Scanner;

public class Sender {
    public static void main(String[] args) throws Exception {

        //declare requirements
        Scanner input = new Scanner(System.in);
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        byte[] hashedInput = "".getBytes();
        SecureRandom random = new SecureRandom();

        //Initiate KeyPairGenerator and Set signature type.
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
        Signature dsa = Signature.getInstance("SHA256withDSA");

        // take the input text from user.
        System.out.println("Please enter text:");
        String inputPlainText = input.nextLine();


        // retrieve the hash of the input text or if that fails then exit the program.
        try{
            hashedInput = MessageDigestor.messageDigest(inputPlainText);
        }catch (Exception e ){
            System.out.println("error: "+ e);
            System.exit(1);
        }
        System.out.println("digest : " + Utils.toHex(hashedInput));

        //generate the keys

        //keys
        keyGen.initialize(2048, random);
        KeyPair pair = keyGen.generateKeyPair();
        PublicKey pubKey = pair.getPublic();
        PrivateKey privKey = pair.getPrivate();

        //Sign the hash using the private key
        dsa.initSign(privKey);
        dsa.update(hashedInput);
        byte[] signedHash = dsa.sign();

        System.out.println("DSA signature hash: \n" + Utils.toHex(signedHash));

        //generate the message
        Message senderMessage = new Message(inputPlainText, signedHash, pubKey);

        //send the message to the verifier
        Verifier.verifyMessage(senderMessage);

    }
}
```

E DSA Verifier Program

```
import javax.crypto.Cipher;
import java.security.Signature;

public class Verifier {
    public static void main(String[] args) {

    }
    public static void verifyMessage (Message senderMessage) throws Exception{
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        Signature dsa = Signature.getInstance("SHA256withDSA");

        //Initiate verifier hashed input.
        byte[] verifierHashedInput = "".getBytes();

        //If the message has been intercepted and changed then again the signatures will not
            match when they are generated.
        String interceptedMessage = "This is not a sample message";

        //assign received plain text to verifier variable
        String verifierPlainText = senderMessage.getOriginalMessage();

        System.out.println("sent plainText: " + senderMessage.getOriginalMessage());
        System.out.println("received plainText: " + verifierPlainText);
        System.out.println("received signature: \n" + Utils.toHex(senderMessage.getSignedHash()));
        System.out.println("Original senders signed hash: \n" +
            Utils.toHex(senderMessage.getSignedHash()));

        //calculates own hashed message using the input that you have provided in the message
            being received
        try{
            verifierHashedInput = MessageDigestor.messageDigest(verifierPlainText);
        }catch (Exception e ){
            System.out.println("error: "+ e);
            System.exit(1);
        }

        //start verify password using the public key from the message
        dsa.initVerify(senderMessage.getPubKey());
        //run through the DSA using the hashed value that verifier has generated
        dsa.update(verifierHashedInput);

        //then verify the updated dsa against the key that was passed with the sender.
        //If the values verify then they will return true otherwise they will return false.
        boolean verifies = dsa.verify(senderMessage.getSignedHash());

        //verifier using the original message matches the sender hash which was decrypted using
            the public key compares the signatures to see if they match
        if(verifies){
            System.out.println("key values match");
        } else {
            System.out.println("hashed values dont match this message is not trustworthy.");
        }



    }
}
```

F Diffie Hellman 4 Party Key Exchange

```
/*
A Lisitsa, 2019, The code below was taken without any changes from
https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#DH3Ex
*/
/*
 * Copyright (c) 1997, 2017, Oracle and/or its affiliates. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *   - Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *
 *   - Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 *   - Neither the name of Oracle nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
/*
 * This program executes the Diffie-Hellman key agreement protocol between
 * 3 parties: Alice, Bob, and Carol using a shared 2048-bit DH parameter.
 */
public class DHKeyAgreement4 {
    private DHKeyAgreement4() {}
    public static void main(String argv[]) throws Exception {
        // Alice creates her own DH key pair with 2048-bit key size
        System.out.println("ALICE: Generate DH keypair ...");
        KeyPairGenerator aliceKpairGen = KeyPairGenerator.getInstance("DH");
        aliceKpairGen.initialize(2048);
        KeyPair aliceKpair = aliceKpairGen.generateKeyPair();

        //Do you want to see the output from the shared key generation?
        Boolean showKeys = false;

        // This DH parameters can also be constructed by creating a
        // DHParameterSpec object using agreed-upon values
        DHParameterSpec dhParamShared = ((DHPublicKey)aliceKpair.getPublic()).getParams();
        // Bob creates his own DH key pair using the same params
        System.out.println("BOB: Generate DH keypair ...");
```

```
KeyPairGenerator bobKpairGen = KeyPairGenerator.getInstance("DH");
bobKpairGen.initialize(dhParamShared);
KeyPair bobKpair = bobKpairGen.generateKeyPair();
// Carol creates her own DH key pair using the same params
System.out.println("CAROL: Generate DH keypair ...");
KeyPairGenerator carolKpairGen = KeyPairGenerator.getInstance("DH");
carolKpairGen.initialize(dhParamShared);
KeyPair carolKpair = carolKpairGen.generateKeyPair();
// David creates his own DH key pair using the same params
System.out.println("CAROL: Generate DH keypair ...");
KeyPairGenerator davidKpairGen = KeyPairGenerator.getInstance("DH");
davidKpairGen.initialize(dhParamShared);
KeyPair davidKpair = davidKpairGen.generateKeyPair();

// Alice initialize
System.out.println("ALICE: Initialize ...");
KeyAgreement aliceKeyAgree = KeyAgreement.getInstance("DH");
aliceKeyAgree.init(aliceKpair.getPrivate());
// Bob initialize
System.out.println("BOB: Initialize ...");
KeyAgreement bobKeyAgree = KeyAgreement.getInstance("DH");
bobKeyAgree.init(bobKpair.getPrivate());
// Carol initialize
System.out.println("CAROL: Initialize ...");
KeyAgreement carolKeyAgree = KeyAgreement.getInstance("DH");
carolKeyAgree.init(carolKpair.getPrivate());
//David initialize
System.out.println("DAVID: Initialize ...");
KeyAgreement davidKeyAgree = KeyAgreement.getInstance("DH");
davidKeyAgree.init(davidKpair.getPrivate());

System.out.println("Iteration 0: All parties only have their own public keys");

//gets previous persons Key
System.out.println("Iteration 1: All parties pass public key forward one.");
// Alice uses Davids's public key
Key ad = aliceKeyAgree.doPhase(davidKpair.getPublic(), false);
// Bob uses Alice's public key
Key ba = bobKeyAgree.doPhase(aliceKpair.getPublic(), false);
// Carol uses Bob's public key
Key cb = carolKeyAgree.doPhase(bobKpair.getPublic(), false);
// David uses Carol's public key
Key dc = davidKeyAgree.doPhase(carolKpair.getPublic(), false);


//gets previous previous persons key
System.out.println("Iteration 2: All parties pass public key forward two.");
// Alice uses David's result from above
Key adc = aliceKeyAgree.doPhase(dc, false);
// Bob uses Alice's result from above
Key bad = bobKeyAgree.doPhase(ad, false);
// Carol uses Bob's result from above
Key cba = carolKeyAgree.doPhase(ba, false);
//David uses Carols result from above
Key dcb = davidKeyAgree.doPhase(cb, false);


//gets previous previous previous persons Key
System.out.println("Iteration 3: All parties pass public key forward Three.");
// Alice uses David's result from above
aliceKeyAgree.doPhase(dcb, true);
// Bob uses Alice's result from above
bobKeyAgree.doPhase(adc, true);
```

```
        // Carol uses Bob's result from above
        carolKeyAgree.doPhase(bad, true);
        //David uses Carol's result from above
        davidKeyAgree.doPhase(cba, true);

        System.out.println("All parties have all public keys for everyone in the network");


        // Alice, Bob and Carol compute their secrets
        System.out.println("Calculating shared secret key.");
        byte[] aliceSharedSecret = aliceKeyAgree.generateSecret();
        byte[] bobSharedSecret = bobKeyAgree.generateSecret();
        byte[] carolSharedSecret = carolKeyAgree.generateSecret();
        byte[] davidSharedSecret = davidKeyAgree.generateSecret();

        if (showKeys) {
            System.out.println("Alice secret: " + toHexString(aliceSharedSecret));
            System.out.println("Bob secret: " + toHexString(bobSharedSecret));
            System.out.println("Carol secret: " + toHexString(carolSharedSecret));
            System.out.println("David secret: " + toHexString(davidSharedSecret));
        }

        // Compare Alice and Bobx
        if (!java.util.Arrays.equals(aliceSharedSecret, bobSharedSecret))
            throw new Exception("Alice and Bob differ");
        System.out.println("Alice and Bob are the same");
        // Compare Bob and Carol
        if (!java.util.Arrays.equals(bobSharedSecret, carolSharedSecret))
            throw new Exception("Bob and Carol differ");
        System.out.println("Bob and Carol are the same");
        if (!java.util.Arrays.equals(carolSharedSecret, davidSharedSecret))
            throw new Exception("Carol and David differ");
        System.out.println("Carol and David are the same");
    }
    /*
     * Converts a byte to hex digit and writes to the supplied buffer
     */
    private static void byte2hex(byte b, StringBuffer buf) {
        char[] hexChars = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
                '9', 'A', 'B', 'C', 'D', 'E', 'F' };
        int high = ((b & 0xf0) >> 4);
        int low = (b & 0x0f);
        buf.append(hexChars[high]);
        buf.append(hexChars[low]);
    }
    /*
     * Converts a byte array to hex string
     */
    private static String toHexString(byte[] block) {
        StringBuffer buf = new StringBuffer();
        int len = block.length;
        for (int i = 0; i < len; i++) {
            byte2hex(block[i], buf);
            if (i < len-1) {
                buf.append(":");
            }
        }
        return buf.toString();
    }
}
```

G Diffie Hellman 4 Party Output

Note: If you would like to print the keys out in order to visually compare them yourself you can change this with the boolean flag in the code base. Change the variable *showKeys* to false and then compile and run the program again.

These Instructions assume you are compiling from the directory that the code file is in.
Compile:javac ./DHKeyAgreement4.java
Run:java DHKeyAgreement4

Output:

```
ALICE: Generate DH keypair ...
BOB: Generate DH keypair ...
CAROL: Generate DH keypair ...
CAROL: Generate DH keypair ...
ALICE: Initialize ...
BOB: Initialize ...
CAROL: Initialize ...
DAVID: Initialize ...
Iteration 0: All parties only have their own public keys
Iteration 1: All parties pass public key forward one.
Iteration 2: All parties pass public key forward two.
Iteration 3: All parties pass public key forward Three.
All parties have all public keys for everyone in the network
Calculating shared secret key.
Alice and Bob are the same
Bob and Carol are the same
Carol and David are the same
```