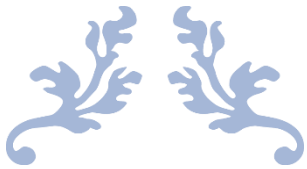

CENG 471- TERM PROJECT

STAGE 3

MAY 19, 2019

HASAN YENIADA
220201024

Table of Contents

1) INTRODUCTION

In that third stage of term project, it is asked for us to implement the following scenario:

The sender would like to send a document after he/she sign it. Then, the receiver should verify the sender's signature. However, it is also asked for us to implement solution to satisfy secrecy, identification, integrity and non-repudiation requirements. I will explain one by one how to implement the given scenario and satisfy the requirements in that report.

2) FIRST STEP – SHARED GLOBAL PUBLIC KEY VALUES GENERATION(P, Q, G)

In the first step, we should generate shared global public keys p, q and g. I have used DSAParams class from java.security library. Because if I generate them using the modules which I implemented for the first and second stage, the execution time of them become very high, so I generated them like that:

```
// I generated (p, q, g) using DSAParams class,
// because execution time become very high If I use my implemented modules.
public BigInteger generateQ() {
    this.globalPublicKeyQ = dsaParams.getQ();
    return this.globalPublicKeyQ;
}
public BigInteger generateP() {
    this.globalPublicKeyP = dsaParams.getP();
    return this.globalPublicKeyP;
}
public BigInteger generateG() {
    this.globalPublicKeyG = dsaParams.getG();
    return this.globalPublicKeyG;
}
```

```
The Shared Global Public Key Values Are Being Creating...

Generated q:
864205495604807476120572616017955259175325408501

Generated p:
178011905478542266528237562450159990145232156369120674273274450314442865788737020770612695

Generated g:
174068207532402095185811980123523436538604490794561350978495831040599953488455823147851597
Global Public Key Values (p, q, g) are created successfully...
```

3) SECOND STEP – PUBLIC/PRIVATE KEY GENERATION OF USERS

In the second step, the Sender Alice and the Receiver Bob generates their own random private keys X, and then compute their public keys Y, implementation of public and private key generation is:

```
public BigInteger generateRandomPrivateKeyX(BigInteger q) {

    this.privateKeyX = DSA.getRandomBigInteger(BigInteger.ONE, q);

    return this.privateKeyX;
}

public BigInteger computePublicKeyY(BigInteger g, BigInteger p) {

    BigInteger x = getPrivateKey();

    this.publicKeyY = fE.getModularExponentiation(g, x, p); //y = g**x(mod p)

    return this.publicKeyY;
}
```

Private Key = x is the random BigInteger such that: x < q

Public Key = y is computed as: y = g^^x (mod p)

One example execution:

```
Alice is Generating her Own Private/Public Keys
Public Key(y) of Alice:
15666028687140677317494158017927992569013376436018994636981589506254789233878046326971861
1021

Bob is Generating his Own Private/Public Keys
Public Key(y) of Bob:
88802322491954720554635487031027681619069323975155648584190258389477557434081566833817826
```

4) THIRD STEP – DSA DIGITAL SIGNATURE CREATION BY SENDER ALICE

After generating shared global public keys (p, q, g) and the privatee/public keys of the users, now we can create digital signature for the document, the implementation:

```
//_____________________________________________________________
//__________________3) DSA Signature Creation__________________

System.out.println("\nSender Alice is creating DSA Signature Pair (r, s)...");

List<BigInteger> signaturePair = ((Sender) senderAlice).createSignaturePair(document, p, q, g);

System.out.println("\nSignature Pair (r, s) has been created successfully...");
```

Alice creates signature pair like that:

```
public List<BigInteger> createSignaturePair(String message, BigInteger p,
                                             BigInteger q, BigInteger g) {

    System.out.println("\nStep 1) Generate random signature key k, k < q:");
    BigInteger k = generateK(q);
    System.out.println("\nGenerated k:");
    System.out.println(k);

    System.out.println("\nStep 2) Compute Signature Pair (r, s)...");
    BigInteger r = computeR(p, q, g, k);
    System.out.println("\nGenerated r:");
    System.out.println(r);

    BigInteger s = computeS(message, p, q, k, r, getPrivateKey());
    System.out.println("\nGenerated s:");
    System.out.println(s);

    signaturePair.add(r);
    signaturePair.add(s);

    return signaturePair; //Signature pair: (r, s)
}
```

One execution result:

```
Sender Alice is creating DSA Signature Pair (r, s)...

Step 1) Generate random signature key k, k < q:

Generated k:
480023603218038273038579901523312758985454145729

Step 2) Compute Signature Pair (r, s)...

Generated r:
99909001051679307991019885744152221026570207133

Generated s:
14010575612146975097373115022679723583238515743478337102547885169054942928217429172704183457180189144

Signature Pair (r, s) has been created successfully...
```

5) FOURTH STEP – DSA DIGITAL SIGNATURE VERIFICATION BY RECEIVER BOB

Up to now, the sender Alice created digital signature pair, and now the receiver Bob will use that signature pair (r, s), shared global public key values (p, q, g) and the public key of the Alice, then compute the v from them and compare that v with the r and returns signature is valid or not, here is the implementation:

```
//_____________________________________________________
//_________________3) DSA Signature Verification_________________

System.out.println("\nReceiver Bob is verifying the Signature...");

BigInteger r = signaturePair.get(0);
BigInteger s = signaturePair.get(1);

System.out.println("Bob is computing v...");

BigInteger v = ((Receiver) receiverBob).computeV(document, r, s, p, q, g, senderAlice.getPublicKey());
System.out.println("\nBom computed v as:");
System.out.println(v);

System.out.println("\nThe r was computed from Sender Alice as:");
System.out.println(r);

System.out.println("\nFinally Bob verifies the Signature...");
```

Bob takes all necessary datas as argument and computes v like that:

```
//___________Digital Signature Verification_______

public BigInteger computeV(String M, BigInteger r, BigInteger s, BigInteger p, BigInteger q, BigInteger g, BigInteger y) {

    BigInteger w = geteE().findMultiplicativeInverse(s, q);      //Gives inverse of s on mod q.
    System.out.println("\nGenerated w:");
    System.out.println(w);

    BigInteger u1 = (getSha().performSHA(M).multiply(w)).mod(q); //Gives [H(M) * w] mod q.
    System.out.println("\nGenerated u1:");
    System.out.println(u1);

    BigInteger u2 = (r.multiply(w)).mod(q);                      //Gives (r * w) mod q.
    System.out.println("\nGenerated u2:");
    System.out.println(u2);

    BigInteger num1 = getfE().getModularExponentiation(g, u1, p); //Gives g**u1 (mod p).
    BigInteger num2 = getfE().getModularExponentiation(y, u2, p); //Gives y**u2 (mod p).
    BigInteger num3 = (num1.multiply(num2)).mod(p);               //Gives [ (g**u1) * (y**u2) mod p ].

    v = num3.mod(q);  //Gives resulting v.

    return v;         //Finally I computed v, and completed to compute signature verification.
}
```

Validation result is computed as follows:

```
Receiver Bob is verifying the Signature...
Bob is computing v...

Generated w:
163653882033135510502957001950178797314812288 20

Generated u1:
734600787446869076413181144927756485315951766144

Generated u2:
615608658914033546224723347062939729375829708419

Bom computed v as:
508343342650831582981502744390576789834601560500

The r was computed from Sender Alice as:
508343342650831582981502744390576789834601560500

Finally Bob verifies the Signature...
(v = r) => Signature is Valid...
```

Note that the resulting v and r are identical so the created Signature is valid.

Now let's comment on the secrecy, identification, integrity and non-repudiation requirements.

6) COMMENTS ON THE REQUIREMENTS

Data integrity:

Data integrity is satisfied with SHA 256 Hash Function, if any change occur during transmission of the message to receiver, the Hash of the received message should give the identical value with the hassh of the sent message, because of the collision free property of the hash functions.

Identification:

Identification is satisfied with DSA, the sender signs the message with DSA and Bob verifies that Signature to identify whether received data is come from sender or not.

Secrecy:

DSA is just a digital signature scheme and cannot be used for encryption or key exchange, so to satisfy secrecy or confidentiality, we should add encryotion to that scenario.

Non-Repudiation:

Non-repudiation is satisfied with DSA, the valid signature can only be created by sender Alice, so she cannot be refuse her action later on, and non repudiation is satisfied.

7) REFERENCES

- http://www.java2s.com/Code/Java/Security/Digital-Signature-Algorithm-DSA.htm