# COMPUTER SYSTEM SECURITY

## Programming Assignment

## P2P Messaging System

Feyza Nur SAKA

1521221051

Computer Engineering

Ömer Korçak

## Platform and Language :

I made this project in java language on netbeans platform.

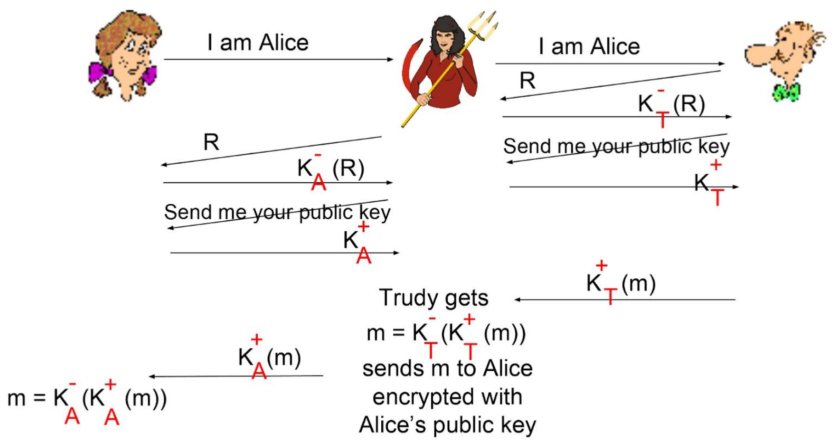
## The subject of the project :

In this project , I implemented a simple peer-to-peer (P2P) messaging system with several security features and there may be more than two users who get the contact information of other users from an application server.

## Features:

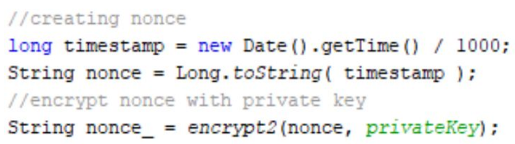
### 1. Handshaking :

After creating the peer to peer structure, I first added the handshaking feature.



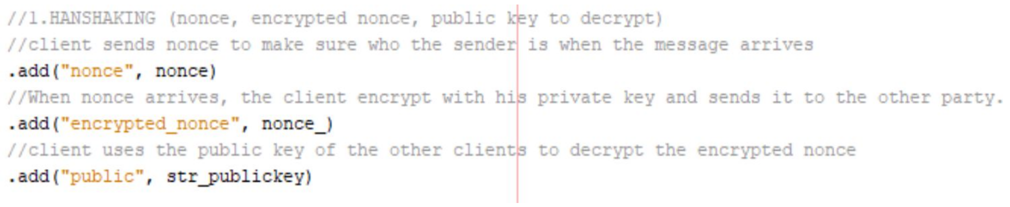
When the message came, Bob created a nonce and sent it to Alice.

I used the date as a basis so that the value produced is always different and that it cannot be repeated.



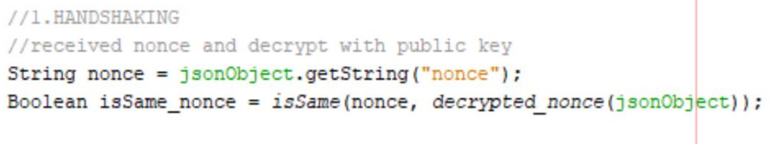
```
//creating nonce
long timestamp = new Date().getTime() / 1000;
String nonce = Long.toString( timestamp );
//encrypt nonce with private key
String nonce_ = encrypt2(nonce, privateKey);
```

Then Alice encrypted her private key with the nonce and sent it to Bob. I used RSA pair generator to generate public key and private key.



```
//1.HANSHAKING (nonce, encrypted nonce, public key to decrypt)
//client sends nonce to make sure who the sender is when the message arrives
.add("nonce", nonce)
//When nonce arrives, the client encrypt with his private key and sends it to the other party.
.add("encrypted_nonce", nonce_)
//client uses the public key of the other clients to decrypt the encrypted nonce
.add("public", str_publickey)
```

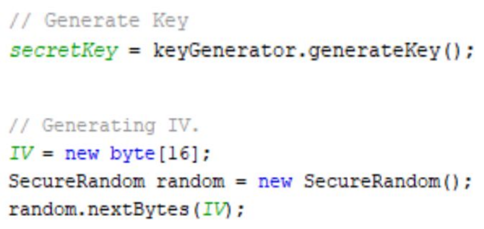
Finally, Bob decrypted encrypted nonce and compared with his own nonce. If they are the same, handshaking is done.



```
//1.HANDSHAKING
//received nonce and decrypt with public key
String nonce = jsonObject.getString("nonce");
Boolean isSame_nonce = isSame(nonce, decrypted_nonce(jsonObject));
```

2. **Key Generation :**

All users generates necessary keys for encryption and Message Authentication Code (MAC), as well as initialization vector(s) (IV).

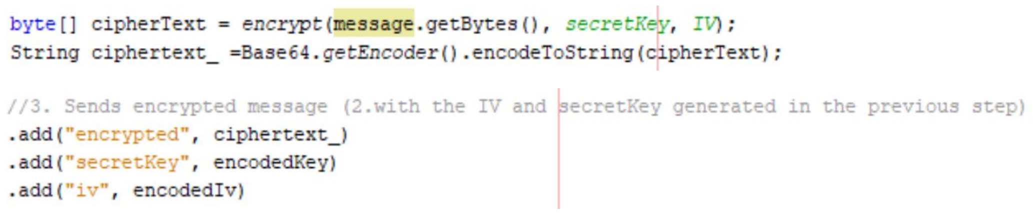


```
// Generate Key
secretKey = keyGenerator.generateKey();


// Generating IV.
IV = new byte[16];
SecureRandom random = new SecureRandom();
random.nextBytes(IV);
```

3. **Message Encryption :**

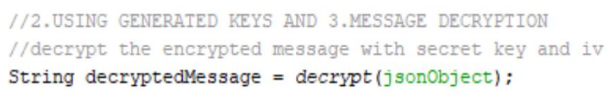
With the generated keys (secret key and IV), I encrypted the message and sent it to the  other party. (I used AES for encryption and decryption operations)



```
byte[] cipherText = encrypt(message.getBytes(), secretKey, IV);
String ciphertext_ =Base64.getEncoder().encodeToString(cipherText);

//3. Sends encrypted message (2.with the IV and secretKey generated in the previous step)
.add("encrypted", ciphertext_)
.add("secretKey", encodedKey)
.add("iv", encodedIv)
```

When the other party received this encrypted message, it decrypted it with the decrypt method.



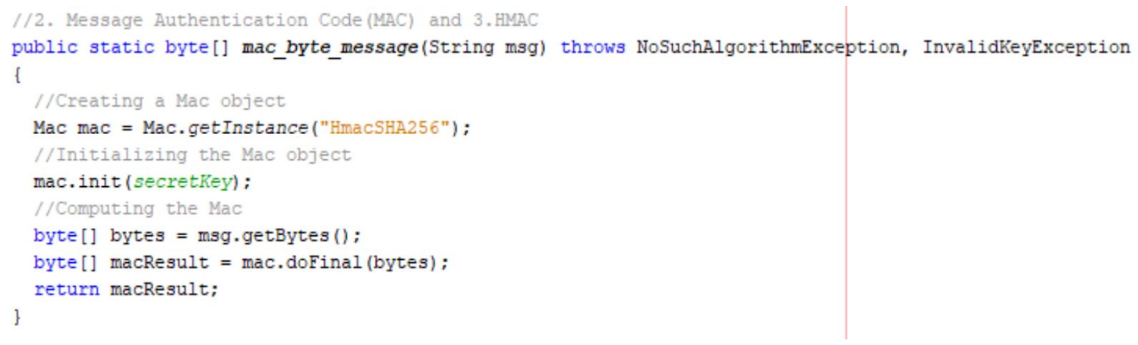
```
//2.USING GENERATED KEYS AND 3.MESSAGE DECRYPTION
//decrypt the encrypted message with secret key and iv
String decryptedMessage = decrypt(jsonObject);
```

4. **Integrity Check :**

Each message passing over the network must have a MAC to ensure that a malicious attacker who interferes with messages on the path is detected. I used the mac_byte_message function that I created with the hashing (HMAC).

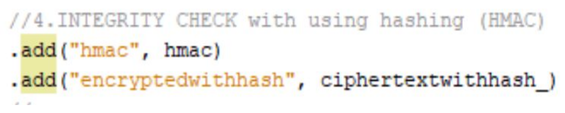
Like other types of MAC, HMAC can be used both to check data integrity and to confirm message content. Any cryptographic hash function can be used to calculate HMAC. For example, if the MD5 or SHA-1 summary function is used to calculate

HMAC, the corresponding MAC algorithm can also be named HMAC-MD5 or HMAC-SHA1 accordingly.



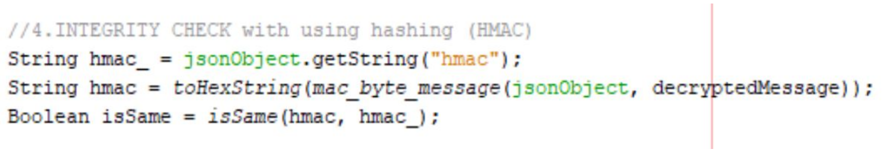
```
//2. Message Authentication Code(MAC) and 3.HMAC
public static byte[] mac_byte_message(String msg) throws NoSuchAlgorithmException, InvalidKeyException
{
  //Creating a Mac object
  Mac mac = Mac.getInstance("HmacSHA256");
  //Initializing the Mac object
  mac.init(secretKey);
  //Computing the Mac
  byte[] bytes = msg.getBytes();
  byte[] macResult = mac.doFinal(bytes);
  return macResult;
}
```

Then I encrypted the hashed message and sent the message to the other party with HMAC.



```
//4.INTEGRITY CHECK with using hashing (HMAC)
.add("hmac", hmac)
.add("encryptedwithhash", ciphertextwithhash_)
```

Finally, I decrypted it and got the hashed message. I would compare the hashed message from the other party to the message I decrypted. If it is the same, the message has been verified. In this way, I provided confidentiality, authentication and integrity.



```
//4.INTEGRITY CHECK with using hashing (HMAC)
String hmac_ = jsonObject.getString("hmac");
String hmac = toHexString(mac_byte_message(jsonObject, decryptedMessage));
Boolean isSame = isSame(hmac, hmac_);
```

5. **Resistance to Replay Attacks :**

Timestamps can be used in all messages to reduce the replay attack. In this way, hackers cannot access messages sent more than a certain time ago.

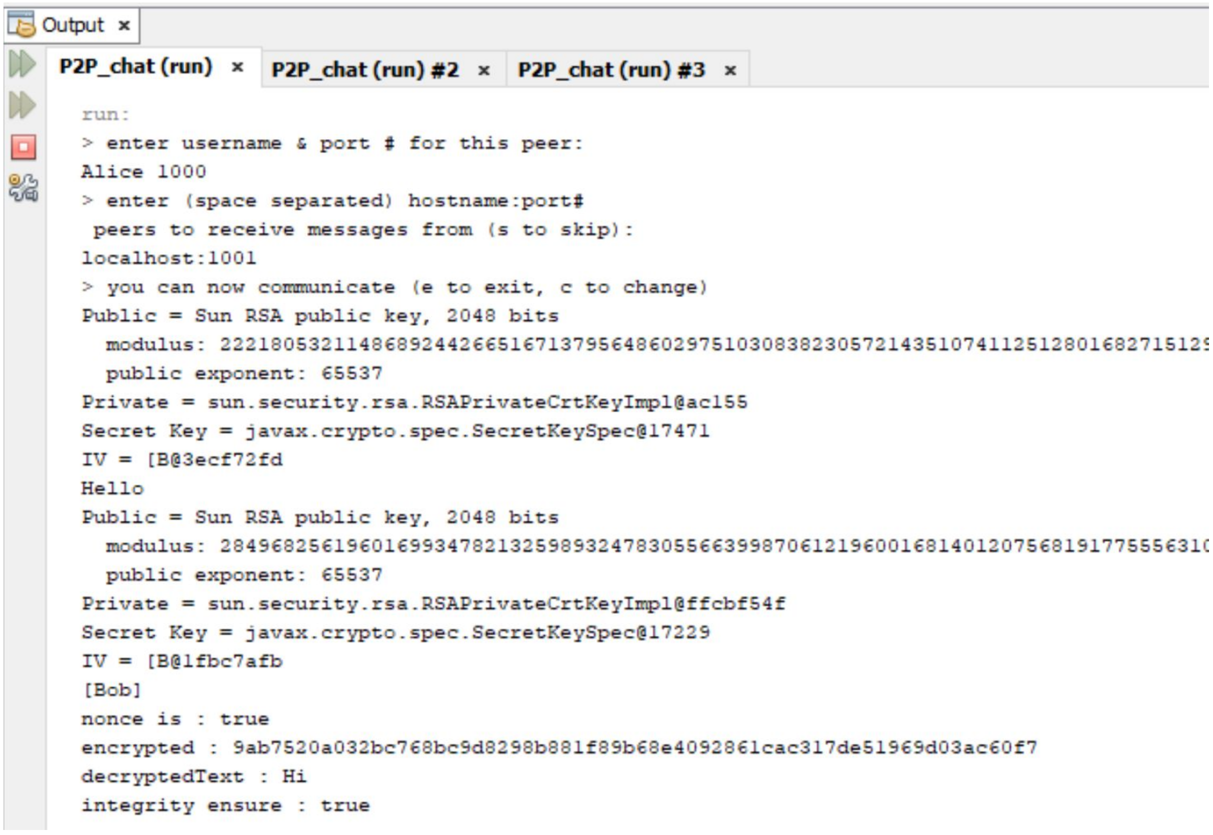
As a more effective method, one-time passwords can be used. Just like the Key update, the password can be updated. In this way, even if the attacker saves the message and throws it again, the password will not be valid and cannot reach what he wants.

6. **Key Update :**

I provided a key update mechanism to provide extra security. In order to be able to perform this process, I generated keys at every message sending stage. So that if any encryption key is compromised for any message, previous messages cannot be decrypted.

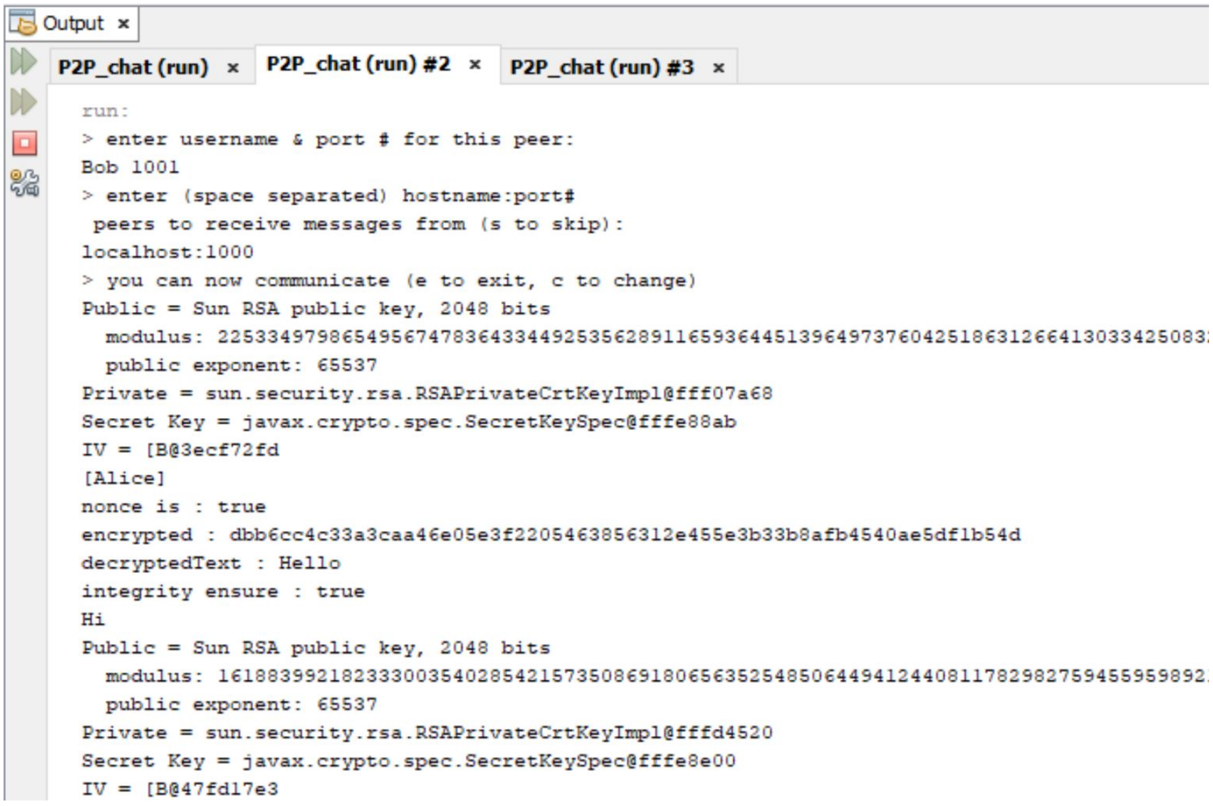
## Outputs :

Alice :



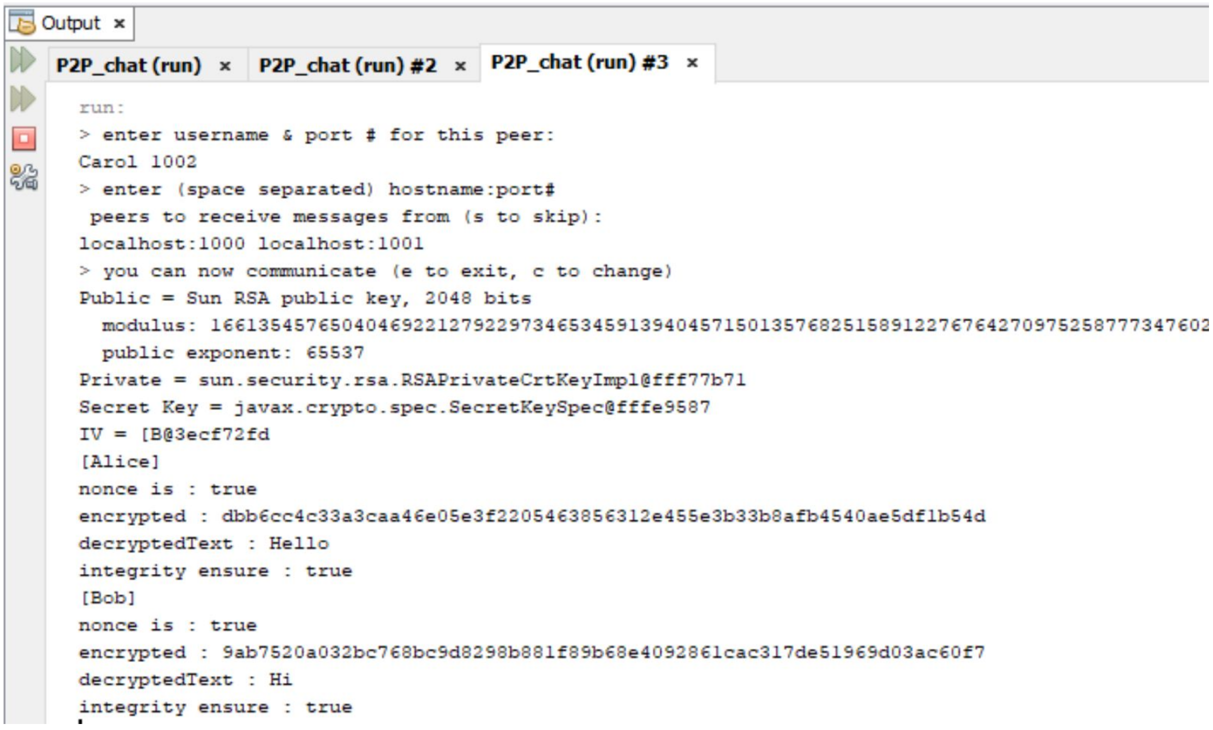
```
run:
> enter username & port # for this peer:
Alice 1000
> enter (space separated) hostname:port#
 peers to receive messages from (s to skip):
localhost:1001
> you can now communicate (e to exit, c to change)
Public = Sun RSA public key, 2048 bits
  modulus: 2221805321148689244266516713795648602975103083823057214351074112512801682715125
  public exponent: 65537
Private = sun.security.rsa.RSAPrivateCrtKeyImpl@ac155
Secret Key = javax.crypto.spec.SecretKeySpec@17471
IV = [B@3ecf72fd
Hello
Public = Sun RSA public key, 2048 bits
  modulus: 284968256196016993478213259893247830556639987061219600168140120756819177555631
  public exponent: 65537
Private = sun.security.rsa.RSAPrivateCrtKeyImpl@ffcbf54f
Secret Key = javax.crypto.spec.SecretKeySpec@17229
IV = [B@1fbc7afb
[Bob]
nonce is : true
encrypted : 9ab7520a032bc768bc9d8298b881f89b68e4092861cac317de51969d03ac60f7
decryptedText : Hi
integrity ensure : true
```

Bob :



```
run:
> enter username & port # for this peer:
Bob 1001
> enter (space separated) hostname:port#
 peers to receive messages from (s to skip):
localhost:1000
> you can now communicate (e to exit, c to change)
Public = Sun RSA public key, 2048 bits
  modulus: 225334979865495674783643344925356289116593644513964973760425186312664130334250833
  public exponent: 65537
Private = sun.security.rsa.RSAPrivateCrtKeyImpl@fff07a68
Secret Key = javax.crypto.spec.SecretKeySpec@fffe88ab
IV = [B@3ecf72fd
[Alice]
nonce is : true
encrypted : dbb6cc4c33a3caa46e05e3f2205463856312e455e3b33b8afb4540ae5df1b54d
decryptedText : Hello
integrity ensure : true
Hi
Public = Sun RSA public key, 2048 bits
  modulus: 161883992182333003540285421573508691806563525485064494124408117829827594559598922
  public exponent: 65537
Private = sun.security.rsa.RSAPrivateCrtKeyImpl@fffd4520
Secret Key = javax.crypto.spec.SecretKeySpec@fffe8e00
IV = [B@47fd17e3
```

Carol :



```
run:
> enter username & port # for this peer:
Carol 1002
> enter (space separated) hostname:port#
 peers to receive messages from (s to skip):
localhost:1000 localhost:1001
> you can now communicate (e to exit, c to change)
Public = Sun RSA public key, 2048 bits
  modulus: 166135457650404692212792297346534591394045715013576825158912276764270975258777347602
  public exponent: 65537
Private = sun.security.rsa.RSAPrivateCrtKeyImpl@fff77b71
Secret Key = javax.crypto.spec.SecretKeySpec@fffe9587
IV = [B@3ecf72fd
[Alice]
nonce is : true
encrypted : dbb6cc4c33a3caa46e05e3f2205463856312e455e3b33b8afb4540ae5df1b54d
decryptedText : Hello
integrity ensure : true
[Bob]
nonce is : true
encrypted : 9ab7520a032bc768bc9d8298b881f89b68e4092861cac317de51969d03ac60f7
decryptedText : Hi
integrity ensure : true
```

# KAYNAKÇA

https://www.kaspersky.com/resource-center/definitions/replay-attack
https://en.wikipedia.org/wiki/HMAC
https://docs.microsoft.com/tr-tr/dotnet/api/system.security.cryptography.hmacsha256?view=netcore-3.1
https://www.kaspersky.com/resource-center/definitions/replay-attack
http://www.sha1-online.com/sha256-java/
https://stackoverflow.com/questions/5531455/how-to-hash-some-string-with-sha256-in-java
https://www.novixys.com/blog/hmac-sha256-message-authentication-mac-java/