**Project2Task5Client**

import java.math.BigInteger;  
import java.net.\*;  
import java.io.\*;  
import java.security.NoSuchAlgorithmException;  
import java.util.Random;  
import java.util.Scanner;  
import java.security.MessageDigest;  
// Name: Leo Lin  
// Andrew ID: hungfanl  
public class SigningClientTCP {  
 static Socket *clientSocket* = null;  
 static int *serverPort*;  
 static BufferedReader *in*;  
 static PrintWriter *out*;  
 static BigInteger *n*, *e*, *d*, *id*;  
 static String *public\_key*, *private\_key*;  
 static final int *NUMBER\_OF\_ID\_DIGITS* = 20; // This is the  
 public static void main(String args[]) {  
 Scanner readInput = new Scanner(System.*in*);  
 // arguments supply hostname  
 System.*out*.println("The client is running.");  
 *generateKey*();  
 *generateID*();  
 // get the port number from the client  
 System.*out*.println("Please enter server port:");  
 *serverPort* = readInput.nextInt();  
 try {  
 *clientSocket* = new Socket("localhost", *serverPort*);  
 String message;  
 String encryptedMessage;  
 System.*out*.println();  
 do{  
 // Initiate the getOption function  
 message = *getMessage*();  
 // If user doesn't insert 4, then pass function will pass the string to the server  
 if(!message.equals("4")){  
 encryptedMessage = *sign*(message);  
 String pack = message + " " + encryptedMessage;  
 int result = *pass*(pack);  
 System.*out*.println("The result is " + result +".");  
 System.*out*.println();  
 }  
 }while(!message.equals("4"));  
 System.*out*.println("Client side quitting. The remote variable server is still running.");  
 } catch (IOException e) {  
 System.*out*.println("IO Exception:" + e.getMessage());  
 } finally {  
 try {  
 if (*clientSocket* != null) {  
 *clientSocket*.close();  
 }  
 } catch (IOException e) {  
 // ignore exception on close  
 }  
 }  
 }  
 // This function returns a string concatenating type, id, number  
 // And returns 4 if the user insert 4.  
 public static String getMessage(){  
 Scanner readInput = new Scanner(System.*in*);  
 String[] options = {"Add a value to your sum.","Subtract a value from your sum.","Get your sum.","Exit client."};  
 for(int i = 0; i < options.length; i++){  
 System.*out*.println((i+1) + ". " + options[i]);  
 }  
 int operator = Integer.*parseInt*(readInput.nextLine());  
 String number = "0";  
 // return string based on user's insertion.  
 switch(operator) {  
 // if the user insert 4, then return 4 to exit the client program  
 case 4:  
 return String.*valueOf*(operator);  
 case 1:  
 System.*out*.println("Enter value to add:");  
 number = readInput.nextLine();  
 break;  
 case 2:  
 System.*out*.println("Enter value to subtract:");  
 number = readInput.nextLine();  
 break;  
 default:  
 }  
 return *id* + "," + *e* + "," + *n* + "," + String.*valueOf*(operator) + "," + number;  
 }  
 //This function takes the concatenated string from the client and pass it to the server  
 public static int pass(String s){  
 String data = null;  
 try {  
 String signed\_s = *sign*(s);  
 *in* = new BufferedReader(new InputStreamReader(*clientSocket*.getInputStream()));  
 *out* = new PrintWriter(new BufferedWriter(new OutputStreamWriter(*clientSocket*.getOutputStream())));  
 *out*.println(s);  
 *out*.flush();  
 data = *in*.readLine(); // read a line of data from the stream  
 System.*out*.println("Received: " + data);  
 } catch (IOException e) {  
 System.*out*.println("IO Exception:" + e.getMessage());  
 }  
 return Integer.*parseInt*(data);  
 }  
 // Code Exploit from RSAExample,  
 // it will generate the public key and private key everytime the client starts the program  
 public static void generateKey(){  
 System.*out*.println("Generating keys");  
 Random rnd = new Random();  
 // Step 1: Generate two large random primes.  
 // We use 400 bits here, but best practice for security is 2048 bits.  
 // Change 400 to 2048, recompile, and run the program again, and you will  
 // notice it takes much longer to do the math with that many bits.  
 BigInteger p = new BigInteger(400, 100, rnd);  
 BigInteger q = new BigInteger(400, 100, rnd);  
  
 // Step 2: Compute n by the equation n = p \* q.  
 *n* = p.multiply(q);  
  
 // Step 3: Compute phi(n) = (p-1) \* (q-1)  
 BigInteger phi = (p.subtract(BigInteger.*ONE*)).multiply(q.subtract(BigInteger.*ONE*));  
  
 // Step 4: Select a small odd integer e that is relatively prime to phi(n).  
 // By convention the prime 65537 is used as the public exponent.  
 *e* = new BigInteger("65537");  
  
 // Step 5: Compute d as the multiplicative inverse of e modulo phi(n).  
 *d* = *e*.modInverse(phi);  
 *public\_key* = String.*valueOf*(*e*) + String.*valueOf*(*n*);  
 *private\_key* = String.*valueOf*(*d*) + String.*valueOf*(*n*);  
 System.*out*.println("your public key = (" + *public\_key* + ")"); // Step 6: (e,n) is the RSA public key  
 System.*out*.println("your private key = (" + *private\_key* + ")"); // Step 7: (d,n) is the RSA private key  
 }  
 // This method generate a unique id with the last 20 byte of the public key  
 public static void generateID(){  
 try {  
 MessageDigest md = MessageDigest.*getInstance*("SHA-256");  
 md.update(*public\_key*.getBytes());  
 byte[] hash\_value = md.digest();  
 byte[] id\_byte = new byte[*NUMBER\_OF\_ID\_DIGITS*];  
 int len\_of\_hash\_value = hash\_value.length;  
 // copy the last 20 bytes to id\_byte  
 for(int i = 0; i < *NUMBER\_OF\_ID\_DIGITS*; i++){  
 id\_byte[*NUMBER\_OF\_ID\_DIGITS*-i-1] = hash\_value[len\_of\_hash\_value - i - 1];  
 }  
 *id* = new BigInteger(id\_byte);  
 System.*out*.println("Your id is: " + *id*);  
 }  
 catch(NoSuchAlgorithmException e) {  
 System.*out*.println("No Hash available" + e);  
 }  
 }  
 // compute the signature (the hash\_value of the whole message)  
 static public String sign(String message) {  
 // compute the digest with SHA-256  
 BigInteger c = null;  
 try{  
 byte[] bytesOfMessage = message.getBytes("UTF-8");  
 MessageDigest md = MessageDigest.*getInstance*("SHA-256");  
 byte[] bigDigest = md.digest(bytesOfMessage);  
 // we add a 0 byte as the most significant byte to keep  
 // the value to be signed non-negative.  
 // Copy every byte of bigDigest  
 byte[] messageDigest = new byte[bigDigest.length + 1];  
 messageDigest[0] = 0; // most significant set to 0  
 for(int i = 0; i < bigDigest.length; i++){  
 messageDigest[i+1] = bigDigest[i];  
 }  
 // From the digest, create a BigInteger  
 BigInteger m = new BigInteger(messageDigest);  
 // encrypt the digest with the private key  
 c = m.modPow(*d*, *n*);  
 // return this as a big integer string  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 return c.toString();  
 }  
}