C18490924 Lab-4 Part C+D

Part C:

There are several cryptographic uses for a clock. Key management functions are often linked to deadlines. The current time can provide both a unique value and a complete ordering of events. Install either network time protocol (NTP) or precision time protocol (PTP) in your computer and check if it is working.

Text

Description automatically generated

List and give examples of attacks that can be mounted against a system with a clock. You

may wish to read the following book chapter and a paper. Write at most four paragraphs

and upload the report in Brightspace.

One example of an attack that can be mounted against a system with a clock is a real time attack using the real time clock chip. If someone could access the system, they could manipulate the clock in some way to damage the casing of the system or alter certain signals. Depending on the computer being attacked determines the complexity of the attack. Some systems require special or raised privileges to access the clock.

Another possible attack that could be mounted against a system with a clock is in the case an attacker sets the clock back. This will cause lots of problems as a result as the computer will believe its in a different time. Possible problems that could arise for this is payments being made at wrong times, system access being revoked or granted to certain people and files being corrupted.

Another way to mount an attack on a system with a clock is to stop the clock. This can cause a wide variety of problems especially in a business or company where many processes may be automated at set times. This could have terrible financial implications for a company. Setting the clock to the future is a similar attack which can have even worse consequences as your computer will think it is in the future and will not be able to securely preform any tasks to do with the time.

Attacks are frequently seen with NTP (Network Time Protocol) as it is often used over the internet, where its delay in between packets with network devices are a gateway for attacks such as DDOS and man-in-the-middle attacks.

Part D

You must write a program to implement RSA encryption and decryption. Small numbers can be used.

*/\*  
You must write a program to implement RSA encryption and decryption. Small numbers  
can be used.  
Author: Stephen Darcy  
Date: 16/12/2021  
\*/  
  
import* java.math.BigInteger;  
*import* java.util.Arrays;  
*import* java.util.Random;  
*import* java.util.Scanner;  
  
*public class* Main {  
  
 *// Public key: (n, e)  
 static* BigInteger publicKey;  
 *// Private key: d  
 static* BigInteger privateKey;  
 *// Modulus n  
 static* BigInteger n;  
 *// set length, so we can generate prime numbers and keys within this range  
 static int* length = 256;  
 *// random value we can use for generating p and q  
 static* Random random;  
 *// byte array to hold encrypted value  
 static byte*[] encrypted;  
  
 *public static void* main(String[] args) {  
 Scanner scanner = *new* Scanner(System.in);  
 System.out.println("\t\*\*\*\* RSA Encryption & Decryption \*\*\*\*\t");  
  
 *int* menuChoice = 3;  
 *// generating public and private keys* generateKeys();  
  
 *// displaying menu to user and looping until choice is 0  
 while* (menuChoice != 0) {  
 System.out.println("\tChoose (1) for Encryption\n\tChoose (2) for Decryption\n\tChoose (0) to exit");  
 System.out.println("\t\*\*\*\* To decrypt a string must first be encrypted \*\*\*\*\t");  
  
 *// getting users choice* menuChoice = scanner.nextInt();  
  
 *// if choice is 1 -> encrypt | else if choice is 2 -> decrypt  
 if* (menuChoice == 1) {  
 encryption();  
 } *else if* (menuChoice == 2) {  
 decryption(encrypted);  
 }  
 }  
  
 }  
  
 */\*\*  
 \* Method that takes the encrypted byte array, checks to make sure it has a value.  
 \* If it does it then uses modular exponentiation in the form of the .modPow method BigIntegers can utilise  
 \* This is preformed on the encrypted byte array (c), to the power of the private key (d) modulus n  
 \*  
 \* @param encrypted This is the ciphertext (c) stored in a byte[]  
 \*/  
 public static void* decryption(*byte*[] encrypted) {  
 *// check if null  
 if* (encrypted != *null*) {  
 *// displaying string before decryption* System.out.println("\tCiphertext to be decrypted is: " + Arrays.toString(encrypted));  
  
 *// preforming m^e mod n = c  
 byte*[] answer = (*new* BigInteger(encrypted).modPow(privateKey, n)).toByteArray();  
  
 *// displaying decrypted string* System.out.println("\tDecrypted string is: " + *new* String(answer) + "\n");  
 } *else* {  
 *// error message if no string set to be decrypted* System.out.println("\tError: Please encrypt a plaintext message first...\n");  
 }  
  
 }  
  
 */\*\*  
 \* Simple method that converts string to byte array  
 \*  
 \* @param ciphertext string to be converted  
 \* @return byte[] created from a string  
 \*/  
 private static byte*[] convertToBytes(String ciphertext) {  
 *return* ciphertext.getBytes();  
 }  
  
 */\*\*  
 \* Method that generates the public and private keys. To do this it also has to create two random prime numbers,  
 \* then calculate the modulus. These are then used to find phi which is used to get our keys  
 \*/  
 public static void* generateKeys() {  
 *// creating two random prime numbers, p and q* random = *new* Random();  
 BigInteger p = BigInteger.probablePrime(length, random); *//using .probablePrime as this uses Miller-Rabin algorithm* BigInteger q = BigInteger.probablePrime(length, random);  
  
 *//calculate modulus n* n = p.multiply(q);  
  
 *// calculate totient phi(n)* BigInteger t = calculateTotient(p, q);  
  
 *// calculate e for public key* publicKey = generateE(t, random);  
  
 *// calculate private key now we have public* privateKey = publicKey.modInverse(t);  
 }  
  
 */\*\*  
 \* Method that takes phi(t) and finds a number(e) co-prime to phi where 1<e<t and gcd(e,t) = 1  
 \*  
 \* @param t phi - used to calculated e  
 \* @param random random value used to calculate our prime numbers  
 \* @return e - a value co-prime to t and gcd(e,t) = 1  
 \*/  
 private static* BigInteger generateE(BigInteger t, Random random) {  
 *// setting e as a random prime in our range* BigInteger e = BigInteger.probablePrime(length / 2, random);  
  
 *// making sure GCD is = 1 and co-prime  
 while* (t.gcd(e).compareTo(BigInteger.valueOf(1)) > 0 && e.compareTo(t) < 0) {  
 e.add(BigInteger.valueOf(1));  
 }  
 *return* e;  
 }  
  
 */\*\*  
 \* Method used to calculate the totient - phi(n) or t = (p-1)\*(q-1)  
 \*  
 \* @param p prime number generated in generateKeys()  
 \* @param q prime number generated in generateKeys()  
 \* @return BigInteger containing the result of (p-1)\*(q-1)  
 \*/  
 public static* BigInteger calculateTotient(BigInteger p, BigInteger q) {  
 *return* (p.subtract(BigInteger.valueOf(1)).multiply(q.subtract(BigInteger.valueOf(1))));  
 }  
  
 */\*\*  
 \* Method used to read in a plain text string the user wishes to encrypt. This is then converted to a bytes array  
 \* using the convertToBytes() method. We then use BigIntegers .modPow function to use modular exponentiation to  
 \* calculate ciphertext (c) with the formula -> plaintext(m)^publicKey(e) modulus n. The result is then displayed  
 \* to the user.  
 \*/  
 public static void* encryption() {  
 *// scanner to read plaintext* Scanner scanner = *new* Scanner(System.in);  
 System.out.println("\tPlease enter plaintext to be encrypted: ");  
  
 *// getting plaintext* String plaintext = scanner.nextLine();  
  
 *// convert plaintext to bytes[]  
 byte*[] plaintextBytes = convertToBytes(plaintext);  
  
 *// preforming m^e mod n to get c  
 byte*[] answer = (*new* BigInteger(plaintextBytes).modPow(publicKey, n)).toByteArray();  
  
 System.out.println("\tEncrypted string is: " + Arrays.toString(answer) + "\n");  
  
 *// setting encrypted bytes[] to our answer for decryption* encrypted = answer;  
 }  
}