MIECT: Security 2017-18

Practical Exercise: Asymmetric Key Cryptography (RSA)

September 20, 2017 Due date: no date

Changelog

- v1.0 - Initial Version.

Introduction

In order to elaborate this laboratory guide it is required to install the Java Development Environment (JDK). If you are using the Virtual Machine provided, this should be already present.

1 Key generation

Implement a small program to generate a pair of asymmetric keys to be used with the `RSA` algorithm. The program should accept the key size (1024, 2048, 3072, or 4096) as the first argument, and create the key with this size. After the keys are generated, it should write the public key to a file specified as the second argument, and the private key to a file specified as the third argument.

Therefore the program should be executed as:

```
java AsymKeys 1024 public.key private.key
```

The basic structure of such program will be:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance(algorithm);

kpg.initialize(keySize);
KeyPair keyPair = kpg.generateKeyPair();
```

Tip: use the classes java.security.**KeyPairGenerator**, java.security.**KeyPair** and the interfaces java.security.**PrivateKey** and java.security.**PublicKey**.

2 Ciphering with the `RSA` algorithm

Implement a second program to cipher the content of a file using the `RSA` algorithm. The program should accept three arguments: the first is the name of the file containing the (public) key to use; the second is the name of the file containing the clear text file to cipher; and the third is the name of the file to write the ciphertext.

The basic structure of such program will be:

```
Cipher cipher = Cipher.getInstance(algorithm);
cipher.init(Cipher.ENCRYPT_MODE, publicKey);
cipherText = cipher.doFinal(clearText.getBytes());
```

Tip: use the classes java.security.**KeyFactory**, javax.crypto.**Cipher** and java.security.spec.**X509EncodedKeySpec**.

Note: Take in consideration that a specific key size will impose a specific block size. Take a look at the `getBlockSize` method of the javax.crypto.**Cipher** class. Only use small files with the appropriate size. The methods to handle larger files will be addressed in subsection 3.1.

3 Deciphering with the `RSA` algorithm

Implement a second program to decipher the content of a file using the `RSA` algorithm. The program should accept three arguments: the first is the name of the file containing the (private) key to use; the second is the name of the file containing the ciphertext file to decipher; and the third is the name of the file to write the clear text.

Tip: use the classes java.security.**KeyFactory**, javax.crypto.**Cipher**, and java.security.spec.**PKCS8EncodedKeySpec**, as well as the interface java.security.**PrivateKey**.

3.1 How to cipher a big file?

As referred previously, when using `RSA` the maximum number of bytes that can be successfully ciphered is limited by the key size. For a key with 1024 bits, the block is of 117 bytes. This is a major limitation of `RSA` as the size of most files, emails, and webpages are well above this value.

A simple solution would be to break the input text in blocks, each with 117 bytes, and cipher the blocks independently. This is the approach followed when using the `ECB` approach in symmetric ciphers, with the important problems already discussed in the last class (patterns).

Moreover, `RSA` is very slow in comparison with the symmetric algorithms. As an example, in a commonly available laptop, it is possible to do 204 sign (cipher) operations per second with `RSA-1024`, while it is possible to do 150 million cipher operations over 128 bits when using `AES-128-CBC`[1].

Therefore the use of `RSA` for ciphering big texts (more than one block) is discouraged. Can you propose a solution for quickly securing a large file with `RSA`, so that only the owner of a given public key can decipher it? Maybe by combining different algorithms (AES and RSA).

Implement the respective hybrid cipher and decipher programs and evaluate your solution.

4 Implementing the `RSA` algorithm

While the implementation of cryptographic algorithms for use in final applications is discouraged, in relation to use well known, publicly available and properly reviewed libraries, implementing the `RSA` algorithm is a simple task with educational value.

The `RSA` key generation process can be quickly described according to the following steps. Take in consideration that in Java, manipulation of large integers can be done with the java.math.**BigInteger**.

- Generate two (large) random primes, P and Q, of approximately equal size. The java.math.**BigInteger** class provides a method (`probablePrime`) which generates probable primes with a given bit length. Take in consideration the P and Q are just part of the key. Therefore these prime numbers must be about half the size of the desired key;

[1] You can check your laptop running `openssl speed`

- Compute $N = P \times Q$. You can use the standard multiplication of the java.math.**BigInteger** class;
- Compute $Z = (P - 1) \times (Q - 1)$. You can use the standard multiplication of the java.math.**BigInteger** class;
- Choose an integer E, $1 < E < Z$, such that $GCD(E, Z) = 1$. GDC is the Greatest Common Divisor and is supported by the java.math.**BigInteger** class;
- Compute the secret exponent D, $1 < D < Z$, such that $E \times D \equiv 1 \bmod Z$. This can be computed using the `modInverse` method (e.g, `D = E.modInverse(Z)`)
- The public key is the tuple $< N, E >$ and the private key is the tuple $< N, D >$.

After the keys are generated, the algorithm supports two operations: cipher and decipher. Remember that the clear text is handled as a `BigInteger`, even if the clear text is some text file.

Ciphering a text requires computing $c = t^E \bmod N$

Deciphering a cipherText requires computing $t = c^D \bmod N$

Where c is the ciphertext, t is the clear text, D, E, N represent the key components.

Implement a program generating a key pair and then using the keys to cipher and decipher a clear text. Consider the the method `modPow` of the java.math.**BigInteger** class for ciphering and deciphering.

References

1. RSA Cryptography Specifications Version 2.0, `http://tools.ietf.org/html/rfc2437`
2. Basic RSA example, `http://www.java2s.com/Tutorial/Java/0490__Security/BasicRSAexample.htm`