MIECT: Security 2020-21

Practical Exercise: Asymmetric Key Cryptography (RSA)

October 6, 2020 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) or Python 3.

The examples provided will use both Java and Python. For Python you need to install the `cryptography` module.

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. Once having the keys, the program 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 in Java will be:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance( "RSA" );

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

Tip: use the classes

- java.security.**KeyPairGenerator**
- java.security.**KeyPair**

and the interfaces

- java.security.**PrivateKey**
- java.security.**PublicKey**

For Python consider the execution of the program with different parameters, namely a password for protecting the key file and a single file name with the PEM extension:

```
python3 asym_keys.py 1024 password key.PEM
```

The basic structure of such program in Python will be:

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend

# Set variables key_size and pwd out of the program arguments
...

# Use 65537 (2^16 + 1) as public exponent

priv_key = rsa.generate_private_key( 65537, key_size, default_backend() )

# Save the key pair to a PEM file protected by the password saved in variable pwd

pem_encoding = priv_key.private_bytes( serialization.Encoding.PEM,
                                       serialization.PrivateFormat.PKCS8,
                                       serialization.BestAvailableEncryption( bytes( pwd,
                                           "utf-8" ) ) )

# Save the contents of pem_encoding in a file
...
```

2 Ciphering with the RSA algorithm

Implement a second program to cipher the content of a file using the RSA algorithm and the PKCS #1 OAEP (Optimal Asymmetric Encryption Padding) padding. 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.

Tip: you can use less arguments and consider the use of STDIN and STDOUT in the absence of file specifications.

The basic structure of such program in Java will be:

```
Cipher cipher = Cipher.getInstance ( "RSA/ECB/OAEPWithSHA-1AndMGF1Padding" );
cipher.init( Cipher.ENCRYPT_MODE, publicKey );

// clearText must be a byte array

cryptogram = cipher.doFinal(cleartext);
```

Tip: use the classes

- java.security.**KeyFactory**,
- javax.crypto.**Cipher**
- java.security.spec.**X509EncodedKeySpec**.

Note: Take in consideration that a specific key size will impose a specific block size, which limits the amount of bytes that `cleartext` can have. 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 section 4.

Tip: for saving both private and public key contents in a single PEM file you can consider using the Bouncy Castle package[1].

The basic structure of such program in Python will be:

[1] `https://www.bouncycastle.org`

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend

# Load key pair to a PEM file protected by a password

with open( key_filename, "rb" ) as kf:
    priv_key = serialization.load_pem_private_key( kf.read(),
                                                   bytes( password, "utf-8" ),
                                                   default_backend() )

pub_key = priv_key.public_key()

# Calculate the maximum amount of data we can encrypt with OAEP + SHA256

maxLen = (pub_key.key_size // 8) - 2 * hashes.SHA256.digest_size - 2

# Read for plaintext no more than maxLen bytes from the input file

# Encrypt the plaintext using OAEP + MGF1(SHA256) + SHA256

ciphertext = pub_key.encrypt( plaintext,
                              padding.OAEP( padding.MGF1( hashes.SHA256() ),
                                            hashes.SHA256(), None ) )

# Write ciphertext in the ouput file
```

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**
- java.security.spec.**PKCS8EncodedKeySpec**

as well as the interface

- java.security.**PrivateKey**.

Note: for the Python program you may have to use an extra argument, a password, to get access to the private key stored inside the key file.

4 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 and OAEP with SHA-1, that maximum is 86 bytes[2]. For other paddings the maximum is different; for instance, for PKCS #1 v1.5 it is 117 bytes[3]. This is a major limitation of RSA as the size of most files, emails, and Web pages are well above this value.

A simple solution would be to break the input text in blocks, each with 86 bytes, and cipher the blocks independently. This is the approach followed when using the ECB approach in symmetric ciphers (but now without the pattern repetition problem exemplified in the last class – explain why!!).

[2] $86 = 128 - 2 \times 20 - 2$, where 20 is the SHA-1 output length.
[3] $117 = 128 - 11$

Moreover, `RSA` is very slow in comparison with the symmetric algorithms. As an example, in a commonly available laptop, it is possible to do 7254 sign (cipher) operations per second with `RSA-1024`, while it is possible to do 8 million cipher operations over 128 bits when using `AES-128-CBC`[4].

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.

5 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 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;
- Compute $N = P \times Q$. You can use the standard multiplication of the BigInteger class;
- Compute $Z = (P-1) \times (Q-1)$. You can use the standard multiplication of the 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 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 method `modPow` of the BigInteger class for ciphering and deciphering.

Note: In this example we are skipping the implementation of any padding before encryption, and unpadding (and validation) upon decryption, which usually takes place when using RSA.

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`

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