MIECT: Security 2015-16

Practical Exercise: Symmetric Key Cryptography

September 7, 2015 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). This is provided by `openjdk-6-jdk` or `openjdk-7-jdk`. If you use the virtual machine provided this should be already installed.
Because you will need to visualize files in its binary form, the *ghex* application may also be useful. It is available for installation in the *Ubuntu* repositories using `apt-get install ghex`.

1 Symmetric Key Cryptography

- `Cipher` An instance of the `Cipher` allows to cipher and decipher. Some important methods are: `getInstance`, `init` e `doFinal`.
- `KeyGenerator` An instance of the class `KeyGenerator` is a generator of symmetric keys. Some important methods are: `getInstance` and `generateKey`.
- `SecretKey` An instance of a class implementing the interface `SecretKey` is a symmetric key.

1.1 Creation of a symmetric key

Develop a simple program to generate a symmetric key for the `DES` algorithm, and save it to a file. The file name should be provided as the first argument of the program.

The program should generate keys which are of 56 bits, as this is required by the `DES` algorithm.

Tip: Consider the class `SecretKeySpec` for future use.

The following code, exemplifies how to convert a `byte` array named `keyData` into a `DES SecretKey`:

```
String algo = "DES";
byte[] keyData = "password".getBytes();
SecretKeySpec sks = new SecretKeySpec(keyData, algo);
SecretKeyFactory kf = SecretKeyFactory.getInstance(algo);
SecretKey secretKey = kf.generateSecret(sks);
```

1.2 Ciphering a file using the `DES` algorithm

Modify the program so that you can cipher the content of the file using the `DES` cipher with the generated key. The file should accept to additional parameters, the input file, and the output file.

Ciphers can be made through the `Cipher` class, as depicted in the following example:

```
File fin = new File("input.txt");
byte[] cipherText;
FileInputStream fisin = new FileInputStream(fin);

Cipher c = Cipher.getInstance("DES/ECB/NoPadding");
c.init(Cipher.ENCRYPT_MODE, secretKey);

long bytesRead = 0;
long fileSize = fin.length();
int blockSize = c.getBlockSize();

while (bytesRead < fileSize){
    byte[] dataBlock = new byte[blockSize];
    bytesRead += fisin.read(dataBlock);
    cipherText = c.update(dataBlock);
    //Do something with cipherText
}

cipherText = c.doFinal();
```

Tip: this will use the `ECB` ciphering mode with `NoPadding`

You may find that you are constrained in the files that you are able to cipher. How can these issues be solved?

1.3 Deciphering a file using `DES`

Develop a program similar to the previous but able to decipher a file. Three arguments should be sent to the program: the file to decipher, the output file and the file containing the key to use.

1.4 Ciphering and deciphering a file using `AES`

Modify the previous programs in order to accept an additional parameter stating the algorithm to use. Consider that two options can be provided: `AES` and `DES`.

Take in consideration that `AES` requires keys with at least 128 bits.

2 Cipher modes

2.1 Initialization Vector

Some cipher modes requiring feedback information (`CBC`, `OFB`, and `CFB`), must use an Initialization Vector (IV). The `Cipher` class automatically creates a random IV, but it also allows the developer to provide a specific IV.

Modify the previous program so that a file name containing the IV, and the cipher mode is also provided by argument. The content of this file should be used to initialize the `Cipher` object when deciphering text, or to save a random IV generated when ciphering. The algorithm and cipher can be specified as *Algorithm/Mode*.

Take in consideration that only `CBC`, `OFB`, and `CFB` cipher modes require the use of an IV.

Tip: consider using the `IvParameterSpec` class.

2.2 Propagation of Patterns

In this exercise the objective is to analyze the impact of using `ECB` and `CBC` from the perspective of the propagation of patterns between the text and the ciphertext.
The approach followed will be to use a *BMP* file, cipher it, and visualize the resulting ciphertext. The *BMP* format is very simple and as long as the

first 54 bytes are intact (the header), the remaining content will be shown as an image.

With the program you developed, and using the `ECB` mode, with any algorithm, cipher the file `security.bmp` to another file, named `security-ecb.bmp`. Then restore the header of the ciphered file with the original one. This will allow for any graphics application to interpret the file as a *BMP* file.

The following command can be used:

```
dd if=security.bmp of=security-ecb.bmp bs=1 count=54 conv=notrunc
```

Open both files with any *BMP* viewer, and compare the results. In alternative, you can also just copy the first 54 bytes directly to the destination file without applying any cipher.

Repeat the same operation, with the same algorithm, over the file `security.bmp`, but using the `CBC` mode. You should produce a file named `security-cbc.bmp`. Restore the header, view both images and compare the result.

Repeat the above steps for other algorithms and cipher modes. What can you conclude?

2.3 Error propagation

In this exercise we will analyze the impact of errors in the ciphertext. That is, the effect of modifications to one or more bits in the ciphertext, and then deciphering the ciphertext into the clear text, when using `ECB`, `CBC`, `OFB` and `CFB`.
Using the program developed, with any cipher, and the `ECB` cipher mode, cipher the image that was provided with this assignment.
Using an hex editor, such as `ghex`, select a random byte and take notice of this byte. Then flip one bit to the opposite value. As an example, you can use address `0xec00` which encodes the lower right pixel of the dot in the exclamation mark, after the word RSA.

Decipher the file you just modified using the same cipher and cipher mode. View both the original image, and the one you just obtained. Then compare their content using an hex editor. In particular, focus in the byte that you just changed, and the surrounding bytes. You can also use the `cmp -bl firstFile secondFile` command.
Repeat these steps with the remaining cipher modes, and for each find what is the impact of errors in the ciphertext. Also, define which cipher modes are

more and less sensible to errors in the ciphertext (considering the amount of errors in the final image).

3 Padding

As you should have noticed, these cipher algorithms operate with blocks of information, and the clear text should have the size of the cipher block, or at least it should be a multiple of the block size. However when ciphering texts, there is no assurance regarding the size of the text to cipher. Therefore, a mechanism is required in order to pad the text up to the desired size. This is typically named as padding, and is applied to the last block of text to cipher. When deciphering the ciphertext, this extra pad is removed and doesn't appear on the resulting clear text.

There are several standards for padding, one of them is named `PKCS#5`. Using an image, and the hex editor, create and experiment demonstrating that additional padding is being added to the text. Also, determine that this padding follows the `PKCS#5` standard (IETF RFC2898).

Tip: use `NoPadding` when deciphering and `PKCS5Padding` when ciphering.

4 Triple DES

The `DES` algorithm uses keys with 56 bits, and was considered insecure some time after its creation. However, it was created a method to increase its security by doubling or tripling the key size. This method is frequently called `TripleDES` or `DESede`. What this method introduces is the notion of multiple operations over the text, using different keys.

When using two keys (112 bits), `TripleDES` is implemented by calculating:

$$E_{k1}(D_{k2}(E_{k1}(text))$$

When using three keys (168 bits), `TripleDES` is implemented by calculating:

$$E_{k3}(D_{k2}(E_{k1}(text))$$

Where k_i is a key, D is a decipher operation, and E is a cipher operation. This method is based on the fact that deciphering a text with a wrong key, is equivalent to cipher it again.

Implement a program which applies this method to the `DES` cipher. Please take in consideration that, when ciphering, only the first cipher operation should use padding! When deciphering, the last operation should use padding[1].

5 Cipher Performance

An important aspect of the different ciphers is their performance in common hardware, which varies by a great amount. Taking in consideration the ciphers available in Java (`Blowfish`, `AES`, `DES`, `RC2`, `RC4`, `ARCFOUR`, and `DESede`) implement a program to benchmark each cipher. Consider blocks with size ranging from 16 bytes to 8192 bytes. In order to run the benchmark, consider the method `System.currentTimeMillis()` and see how many time it takes to do 10000000 cipher operations.

References

- `http://docs.oracle.com/javase/tutorial/security/index.html`
- `http://docs.oracle.com/javase/7/docs/technotes/guides/security/index.html`
- `http://tools.ietf.org/html/rfc2898`

[1] Java supports `DESede` natively, but you should not use it unless you wish to test your implementation.