Instituto Superior Técnico, Universidade de Lisboa
Network and Computer Security

Lab Guide:
Java Cryptographic Mechanisms

Revised on 2016-09-24

Goals

- Use the cryptographic mechanisms available in the Java platform.
- Perform attacks exploiting vulnerabilities introduced by the bad use of cryptography.

1. Introduction

This laboratory assignment uses Java Development Kit (JDK) version 7 or later, running over Linux. The Java platform strongly emphasizes security, including language safety, cryptography, public key infrastructure, secure communication, authentication and access control.

The Java Cryptography Architecture (JCA), which is a major piece of the Java platform, includes a large set of application programming interfaces (APIs), tools, and implementations of commonly-used security algorithms, mechanisms, and protocols. It provides a comprehensive security framework for writing applications and also provides a set of tools to securely manage applications.

The JCA APIs include abstractions for secure random number generation, key generation and management, certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), message digests (hashes), and digital signatures. Some examples are the `MessageDigest`, `Signature`, `KeyFactory`, `KeyPairGenerator`, and `Cipher` classes.

Implementation independence, in the Java platform, is achieved using a *provider*-based architecture. The term Cryptographic Service Provider (CSP) refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key conversion services. A program may simply request a particular type of object, e.g., a `MessageDigest` object, implementing a particular service, e.g., the SHA1 digest algorithm, and get an implementation from one of the installed providers. A program may instead request, if necessary, an implementation from a specific provider.

To obtain a security service from an underlying provider, applications rely on the relevant `getInstance()` method. The message digest creation, for example, represents one type of service available from providers. To obtain an implementation of a specific message digest algorithm, an application invokes the `getInstance()` method in the `java.security.MessageDigest` class.

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

Optionally, by indicating the provider name, the program may request an implementation from a specific provider as in the following:

```
MessageDigest md = MessageDigest.getInstance("SHA1", "MyProvider");
```

Providers may be updated transparently to the application when faster or more secure versions are available. In the Java platform, the `java.security.Provider` class is the base class for all security providers. Each CSP contains an instance of this class which contains the provider's name and lists all of the security services/algorithms it implements. Multiple providers may be configured at the same time, and are listed in order of preference. The highest priority provider that implements that service is selected when a security service is requested.

For more information, please read:
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Introduction

2. Cryptographic mechanisms

Extract the `JavaCrypto.zip` package into `/tmp/sirs`. Please notice that all steps that follow expect that this was done, so you must change commands according to an alternative location if used.

Note: For every java command, please write `$java pt.ulisboa.tecnico.meic.sirs.RandomImageGenerator` instead of just `$java RandomImageGenerator`. The package name is omitted for brevity. For the argument files provide the complete path names.

In the directory `intro/inputs`, you can find 3 different images:

- Tecnico: `*.png`, the IST logo
- Tux: `*.png`, Tux, the Linux penguin
- Glider: `*.png`, the hacker emblem (http://www.catb.org/hacker-emblem/)

Each one is presented with three different dimensions: 480x480, 960x960, and 2400x2400. The resolution number is part of the file name. The `ImageMixer` class is available to facilitate the operations on images. Many examples are available, such as the `RandomImageGenerator`, `ImageXor`, and `ImageAESCipher` classes.

2.1 One-Time Pads (Symmetric stream cipher)

When correctly used, one-time pads provide perfect security. One of the constraints to make them work as expected is that the key stream must never be reused. The following steps visual illustrate what happens if they are reused, even if just once:

1. Generate a new 480x480 random image

```
$ java RandomImageGenerator intro/outputs/otp.png 480 480
```

2. Perform the bitwise eXclusive OR operation (XOR) with the generated key

```
$ java ImageXor intro/inputs/tecnico-0480.png intro/outputs/otp.png
intro/outputs/encrypted-tecnico.png
```

3. XOR tux-0480.png with the same generated key

```
$ java ImageXor intro/inputs/tux-0480.png intro/outputs/otp.png
intro/outputs/encrypted-tux.png
```

4. Watch the images `encrypted-tecnico.png` and `encrypted-tux.png`. Switch between them, and see the differences.
5. Make the differences obvious: XOR them together:

```
$ java ImageXor intro/outputs/encrypted-tecnico.png
intro/outputs/encrypted-tux.png intro/outputs/tecnico-tux.png
```

You can see that the reuse of a one-time pad (or any stream cipher key at all) considerably weakens (or completely breaks) the security of the information. In the following, C stands for cipher-text, M for plain-text and K for key: $C1 = M1 \oplus K$ $\quad C2 = M2 \oplus K$

$$C1 \oplus C2 = M1 \oplus M2$$

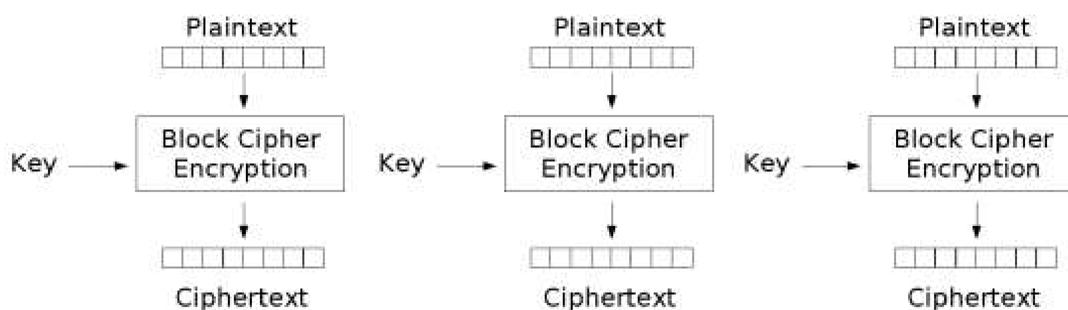
The result you get is the XOR of the images. Feel free to experiment with other images and sizes.

2.2 Block cipher modes

Now that you know that keys should never be reused, remember that the way you use them is also important. You are about to use a symmetric-key encryption algorithm in modes ECB (Electronic Code Book), CBC (Cipher Block Chaining) and OFB (Output FeedBack), to encrypt the pixels from an image.

2.2.1 ECB (Electronic Code Book)

In the ECB mode, each block $m[i]$ is encrypted with key k independently: $c[i] = E_k(m[i])$



1. Begin by generating a new AES Key.

```
$ java AESKeyGenerator w intro/outputs/aes.key
```

2. Then, encrypt the glider image with it:

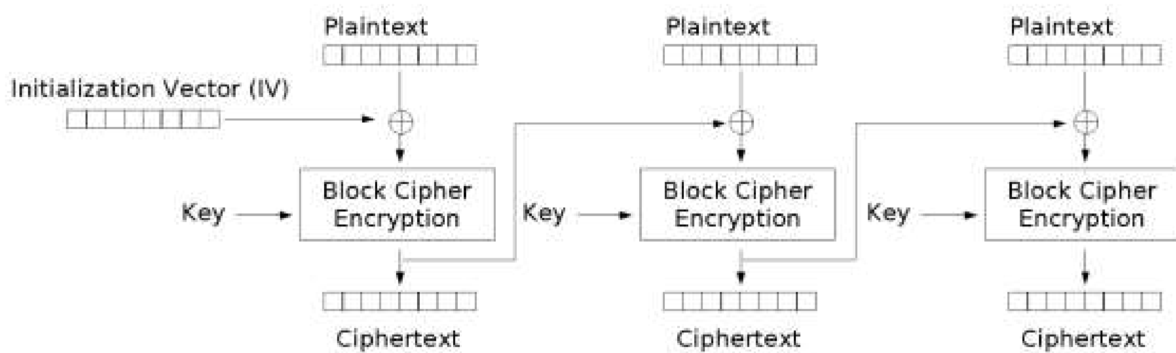
```
$ java ImageAESCipher intro/inputs/glider-0480.png intro/outputs/aes.key
ECB intro/outputs/glider-aes-ecb.png
```

3. Watch the output image. Remember what you have just done: encrypted the image with AES, using ECB mode, and a key you generated yourself.
4. Feel free to try the same thing with the other images (especially with other sizes).

5. Try using Java providers to generate a new DES key, by creating a `DESKeyGenerator` based on the `AESKeyGenerator` class. What is necessary to change in the code for that to happen?
6. Repeat all the previous steps for the new key.
7. Compare the results obtained using ECB mode with DES with the previous ones. What are the differences between them?

2.2.2 CBC (Cipher Block Chaining)

In the CBC mode, each block $m[i]$ is XORed with the ciphertext from the previous block, and then encrypted with key k: $c[i] = E_k(m[i] \oplus c[i-1])$.



2.2.2.1 The encryption of the first block can be performed by means of a random and unique value known as the Initialization Vector (IV).

a. The AES key will be the same from the previous step.
b. Encrypt the glider image with it, this time replacing ECB for CBC:

```
$ java ImageAESCipher intro/inputs/glider-0480.png intro/outputs/aes.key
CBC intro/outputs/glider-aes-cbc.png
```

c. Watch the file `glider-aes-cbc.png`. See the difference it made, changing only the mode of operation.

2.2.2.2 Still in the CBC mode, you might have wondered why the IV is needed in the first block. Consider what happens when you encrypt two different images with similar beginnings, and with the same key k: the initial cipher text blocks will also be similar!

The `ImageAESCipher` class provided has been deliberately weakened: instead of randomizing the IV, it is always the same.

a. This time, encrypt the other two images with AES/CBC, still using the same AES key:

```
$ java ImageAESCipher intro/inputs/tux-0480.png intro/outputs/aes.key CBC
intro/outputs/tux-aes-cbc.png
```

```
$ java ImageAESCipher intro/inputs/tecnico-0480.png intro/outputs/aes.key
CBC intro/outputs/tecnico-aes-cbc.png
```

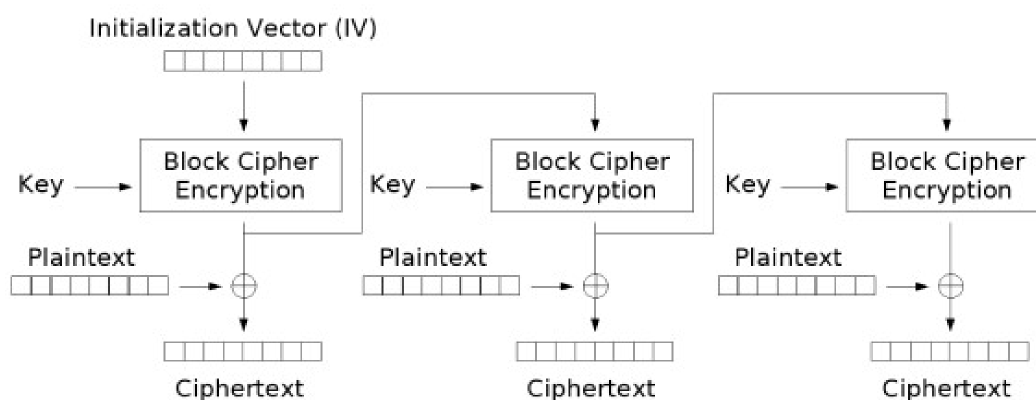
b. Now watch the images `glider-aes-cbc.png`, `tux-aes-cbc.png`, and `tecnico-aes-cbc.png`

Look to the first lines of pixels. Can you see what is going on?

c. Generate a new file by concatenating two png image files. Cipher this new image file. What did you obtain? Now, try doing the previous with two jpeg image files.

2.2.3 OFB

In the OFB mode, the IV is encrypted with the key to make a keystream that is then XORed with the plaintext to make the cipher text.



In practice, the keystream of the OFB mode can be seen as the one-time pad that is used to encrypt a message. This implies that in OFB mode, if the key and the IV are both reused, there is no security.

1. Encrypt the images with OFB:

```
$ java ImageAESCipher intro/inputs/glider-0480.png intro/outputs/aes.key
OFB intro/outputs/glider-aes-ofb.png
$ java ImageAESCipher intro/inputs/tux-0480.png intro/outputs/aes.key OFB
intro/outputs/tux-aes-ofb.png
$ java ImageAESCipher intro/inputs/tecnico-0480.png intro/outputs/aes.key
OFB intro/outputs/tecnico-aes-ofb.png
```

2. Remember that the `ImageAESCipher` implementation has been weakened, by having a null IV, and you are reusing the same AES key. Watch the generated images, and switch quickly between them.
3. Take 2 images (e.g., image1 and image2) and cipher them both. XOR image1 with the ciphered image2. What did you obtain? Why?
4. What is more secure to use: CBC or OFB?

2.3 Asymmetric ciphers

2.3.1 Generating a pair of keys with OpenSSL

1. Private key

```
$ openssl genrsa -out server.key
```

2. Public key:

```
$ openssl rsa -in server.key –pubout > public.key
```

3. Generating a self-signed certificate with these keys:
   - Certificate Signing Request, using same key:

```
$ openssl req -new -key server.key -out server.csr
```

   - Self-sign:

```
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out
server.crt
```

   - In order for our certificate to be able to sign other certificates, OpenSSL requires that a database exists (a .srl file). Create it:

```
$ echo 01 > server.srl
```

4. Then, generating a key to a user is basically repeating the same steps (see commands above), except that the self-sign no longer happens and is replaced by:

```
$ openssl x509 -req -days 365 -in user.csr -CA server.crt -CAkey server.key
-out user.crt
```

5. Sign the file grades.txt with the user certificate:

```
$ openssl dgst -sha256 grades/inputs/grades.txt > grades.sha256
$ openssl rsautl -sign -inkey user.key -keyform PEM -in grades.sha256 >
grades.sig
```

   - Verify the signature:

```
$ openssl rsautl -verify -in grades.sig -inkey server.key
```

The expected output is:

```
SHA256(/tmp/sirs/grades/inputs/grades.txt)=
770ddfe97cd0e6d279b9ce780ff060554d8ccbe4b8eccaed364a8fc6e89fd34d
```

and should always match this:

```
$ openssl dgst -sha256 grades/inputs/grades.txt
   SHA256(/tmp/sirs/grades/inputs/grades.txt)=
770ddfe97cd0e6d279b9ce780ff060554d8ccbe4b8eccaed364a8fc6e89fd34d
```

- Verify the user certificate:

```
$ openssl verify -CAfile server.crt user.crt
```

user.crt: OK

2.3.2 Reading the generated pair of keys with Java

In order to read the generated keys in Java it is necessary to convert them to the right format.

- Convert the private key to PKCS8

```
$ openssl pkcs8 -topk8 -inform PEM -outform DER -in server.key -nocrypt >
server_pkcs8.key
```

- Read the key files using the following command:

```
java RSAKeyGenerator r server_pkcs8.key server.crt
```

2.3.3 Generating a pair of keys with Java

1. Generate a new pair of RSA Keys.

```
$ java RSAKeyGenerator w intro/outputs/priv.key intro/outputs/pub.key
```

2. Based on the `ImageAESCipher` class create `ImageRSACipher` and `ImageRSADecipher` classes.
3. Encrypted the image with the public key and then decrypt it with the private key.

Feel free to try the same thing with the other images - especially with other sizes, like 2400x2400.

3. Additional exercise (tampering with a file)

In the directory `grades/inputs`, you can find the file `grades.txt`, the plaintext of a file with the grades of a course. This flat-file database has a rigid structure: 64 bytes for name, and 16 bytes for each of the other fields, number, age and grade. Unfortunately, you happen to be *Mr. Thomas S. Cook*, and your grade was not on par with the rest of your class because you forgot to study...

1. Begin by encrypting this file into `grades.ecb.aes`. For this example, we will still reuse the AES key generated above and ECB mode.

```
$java FileAESCipher grades/inputs/grades.txt intro/outputs/aes.key ECB
grades/outputs/grades.ecb.aes
```

2. Keeping in mind how the mode operations work, and without using the secret key, try to change your grade to 21 in the encrypted files or give everyone in class a 20. Your goal here is to hack, show that the system is vulnerable. Did you succeed? Did your changes have side effects? If so, which ones, and in which block modes?

3. Now try to attack `grades.cbc.aes` and `grades.ofb.aes`. For this example, we will still reuse the AES key generated above but use the CBC and OFB modes.

```
$java FileAESCipher grades/inputs/grades.txt intro/outputs/aes.key CBC
grades/outputs/grades.cbc.aes
$java FileAESCipher grades/inputs/grades.txt intro/outputs/aes.key OFB
grades/outputs/grades.ofb.aes
```

4. Since the inputs and outputs of cryptographic mechanisms are byte arrays, in many occasions it is necessary to represent encrypted data in text files. A possibility is to use Base64 encoding that, for every binary sequence of 6 bits, assigns a predefined ASCII character. Execute the following to create a Base64 representation of files previously generated.

```
$java Base64Encode grades/outputs/grades.cbc.aes
grades/outputs/grades.cbc.aes.b64
```

5. Decode them:

```
$java Base64Decode grades/outputs/grades.cbc.aes.b64
grades/outputs/grades.cbc.aes.b64.decoded
```

6. Check if they are similar using the `diff` command (or `fc /b` command on Windows).

```
$diff grades/outputs/grades.cbc.aes
grades/outputs/grades.cbc.aes.b64.decoded
```

It should not return anything.

7. Check the difference on the file sizes. Can you explain it? In percent, how much is it? Does Base64 provide any kind of security? If so, how?

8. Use Java to generate the message authentication code (MAC) and digital signature of the grades file. By performing these operations which security requirements are guaranteed?

Acknowledgments

This set of exercises is based on the work developed by the student Valmiky Arquissandas within the scope of his project work for this course, and later refined by the teaching staff.