MIECT: Security 2019-20

Practical Exercise: Symmetric Key Cryptography

October 16, 2019 Due date: no date

Changelog

- v1.4 - Added tips relatively to the use of STDIN and STDOUT.
- v1.3 - Python programs fixed.
- v1.2 - Fixed imports in Java snippets.
- v1.1 - Python programs fixed, Python module name fixed.
- 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.

Because you will need to visualize binary files, the `ghex` application may also be useful. It is available for installation in the Linux repositories using `apt-get install ghex`.

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

1 Symmetric Key Cryptography in Java

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

1.1 Creation of a symmetric key

Develop a program to generate a symmetric key for the `AES` algorithm, and save it to a file. The key should be generated from a password, a process known as Password-Based Encryption (PBE). The password should be provided as the first argument of the program and the file name as the second.

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

The program should generate 128-bit keys, as this is required by the AES algorithm. For dealing with PBE we will use a password-to-key transformation function known as PBKDF2 (Password-Based Key

Derivation Function 2). This is a generic, multi-iteration hashing function, which can be parametrized with another keyed hashing function; we will use HMAC with SHA-1.

The following Java code exemplifies how to convert a textual password, stored in the string `pwd`, into a byte array (named `keyData`) with PBKDF2 (with HMAC and SHA-1), and how to use that byte array to generate an AES `SecretKey`.

```java
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.SecretKey;

// PBKDF2 specifications: password, salt, iterations and output size
//
// salt: a byte array intended to produce different outputs for the same password
// iterations: the number of times the PBKDF2 iterates internally over a generator function
// output size: the number of bits we want to generate from the password (128 for AES)

byte[] salt = new byte[1];
salt[0] = 0; // We need to provide one salt ...

// First, we create a specifications object with all the PBKDF2 parameters we want

PBEKeySpec pbeKeySpec = new PBEKeySpec(pwd.toCharArray(), salt, 1000, 128);

// Then, we create key material (a byte array) using PBKDF2 + HMAC(SHA-1)

SecretKeyFactory skf = SecretKeyFactory.getInstance( "PBKDF2WithHmacSHA1" );
byte[] keyData = skf.generateSecret( pbeKeySpec ).getEncoded();

// Now, we generate an AES key from the key material

SecretKey secretKey = new SecretKeySpec( keyData, "AES" );

// Write secretKey.getEncoded() in a file
```

The following code does the same in Python;

```python
import sys
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend

# The PBKDF2 generator of Python receives as inpout the number of byes to generate,
# instead of bits

salt = b'\x00'
kdf = PBKDF2HMAC( hashes.SHA1(), 16, salt, 1000, default_backend() )
secretKey = kdf.derive( bytes(pwd, 'UTF-8') )

# Write secretKey in a file
```

1.2 Ciphering a file using the AES algorithm

Write a program to encrypt the contents of a file using the AES cipher and a key priviously generated and stored in a file. The program should accept three parameters: the key file, the input and the output files.

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

In Java, ciphers can be explored through the javax.crypto.**Cipher** class, as depicted in the following example for a CBC cipher mode with a PKCS #5 padding:

```java
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.SecretKey;
import java.security.SecureRandom;
import javax.crypto.Cipher;

// Read encoded key from key file (into variable encodedKey)
...

// Setup AES secretKey with the key to encrypt

SecretKey secretKey = new SecretKeySpec( encodedKey, "AES" );

// Create a cipher engine given the algorithm (AES), the encryption mode (CBC)
// and the padding (PKCS \#5)

Cipher c = Cipher.getInstance( "AES/CBC/PKCS5Padding" );

// Set the IV parameters (required for CBC) to a random value.
// The IV must have a size equal to the cipher's block size

byte[] iv = new byte[c.getBlockSize()];
SecureRandom random = new SecureRandom();
random.nextBytes( iv );

// Set the cipher engine to encrypt with the intended key and IV

c.init( Cipher.ENCRYPT_MODE, secretKey, new IvParameterSpec( iv ) );

// Open input file for reading and output file for writing
...

// Write the IV in the output file
...

// Read chunks of the input file, encrypt them and
// write the result to the output file

while (...) { // Cicle to repeat while there is data left on the input file
    // Read a chunk of the input file to the plaintext variable
    ...

    // The length of the plaintext data should be stored in pLen

    ciphertext = c.update( plaintext, 0, pLen );

    // Store the ciphertext in the output file
    ...
}

// Perform the encryption of the last plaintext contents + the padding

ciphertext = c.doFinal();

// Store the ciphertext in the output file
...
```

Check the size of the resulting encrypted files for each input file. Explain why they are always bigger and the size increment.

Change the padding mode to `NoPadding` and observe the result. You may find that you are constrained in the size of the files that you are able to cipher. In other words, your program will not work properly with input files with a size that is not a multiple of 16 bytes.

The following code does the same in Python;

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.backends import default_backend

def main():
    # Read key from key file and setup secretKey with the key to encrypt

    # Setup cipher: AES in CBC mode, w/ a random IV and PKCS #7 padding (similar to PKCS #5)

    iv = os.urandom( algorithms.AES.block_size // 8 );
    cipher = Cipher( algorithms.AES( secretKey ), modes.CBC( iv ), default_backend())
    encryptor = cipher.encryptor()
    padder = padding.PKCS7( algorithms.AES.block_size ).padder()

    # Open input file for reading and output file for writing
    ...

    # Write the contents of iv in the output file
    ...

    while True: # Cicle to repeat while there is data left on the input file

        # Read a chunk of the input file to the plaintext variable

        if not plaintext:
            ciphertext = encryptor.update( padder.finalize() )

            # Write the contents of ciphertext in the output file

            break
        else:
            ciphertext = encryptor.update( padder.update( plaintext ) )

            # Write the ciphertext in the output file
```

1.3 Deciphering a file using AES

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

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. The change should be minimal.

2 Cipher modes

2.1 Initialization Vector

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

Modify the previous programs to accept as argument a file name, which should contain a definition of the cipher algorithm, the cipher mode and an IV (assume a default padding when required). The content of this file should be used to initialize the javax.crypto.**Cipher** object when deciphering a cryptogram. The IV in this file should be a random value generated when ciphering.

Note: you do not need to store the IV anymore in the beginning of the encrypted file.

Take in consideration that only the ECB cipher mode does not require the use of an IV.

2.2 Propagation of patterns

In this exercise the goal is to analyze the impact of using ECB and CBC from the perspective of the propagation of patterns between the plaintext and the corresponding cryptogram.

The approach followed will be to use a BMP file, cipher it, and visualize the resulting cryptogram. The BMP format is very simple and as long as the first 54 bytes (the header) are kept unchanged, the remaining content will be shown as an image.

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

The following command can be used to fix the header from the encrypted contents to the original ones:

```
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.

Repeat the same operation with the same algorithm and over the same file `security.bmp`, but now 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. Do not restore the header!

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 sensitive to errors in the ciphertext (considering the amount of errors in the final image).

3 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 `3DESede`. What this method introduces is the notion of multiple operations over the text, using different keys and is a good example of a cipher reinforcement method.

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

$$E_{k1}\left(D_{k2}\left(E_{k1}\left(text\right)\right)\right)$$

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

$$E_{k3}\left(D_{k2}\left(E_{k1}\left(text\right)\right)\right)$$

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 cryptogram 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].

4 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 `3DESede`) 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 $10,000,000$ cipher operations.

References

- The Java Tutorial: Security Features in Java SE
  `https://docs.oracle.com/javase/tutorial/security/index.html`
- Java Documentation: Security
  `http://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`
- PKCS #5: Password-Based Cryptography Specification Version 2.0
  (`https://tools.ietf.org/html/rfc2898`)
- Welcome to pyca/cryptography
  (`https://cryptography.io`)

[1] Java supports `3DESede` natively, but you should not use it unless you wish to test your implementation.