Information and Organisational Security

Guides for Practical Classes

João Paulo Barraca and Vitor Cunha

Department of Electronics, Telecommunications and Informatics
University of Aveiro

2020-2021

3

Applied Cryptography

3.1 Introduction

In this guide we will develop programs that use cryptographic methods, relying in the Python3 Cryptography module. The module can be installed using the typical package management methods (e.g, `apt install python3-cryptography`), or using the `pip3` tool (e.g. `pip3 install cryptography`). The Virtual Machine provided should already contains the necessary components and nothing else is required.

It will be useful to visualize and edit files in binary format. For that purpose, if you are using Linux, you may install `GHex` or `hexedit` from the repositories.

IMPORTANT: The exercises of this lab are extremely important for the practical project. Implement all code as reusable functions and they may be applicable to the project.

3.2 Symmetric Cryptography

Symmetric cryptography is used by creating an object that represents a given cipher, with some parameters specifying the mode, as well as a key. The cipher object presents an `encryptor` method, that is applied (`update`) to the text in chunks (may require alignment with the cipher block size). After the text is ciphered, a `finalize` method may be used. Decryption is done in a similar way.

It should be noticed that we will directly use the base cryptographic primitives, which imply that the ciphers are constructed with reasonable parameters.

For more information please check the documentation available at the Cryptography.io Hazmat section.

3.2.1 Symmetric key generation

Before a cipher is used, it is required the generation of proper arguments. These arguments are the key, the cipher mode, and potentially the Initialization Vector (IV). The cipher mode is chosen at design time, and the IV should always be a large random number (size similar to the block size or key) that is never repeated. This lab will discuss the IVs in the next sections.

The key can be obtained from good sources of random numbers, or generated from other primitive material such as a password. When choosing the last source (a password), it is imperative to transform the user text into a key of the correct complexity. While there are many methods, we will consider the Password Based Key Derivation Function 2 (PBKDF2), which takes a key, a random value named salt, a digest algorithm, and a number of iterations (you should use several thousands). The algorithm will iterate the digest algorithm in a chain starting in the concatenation of the salt and key, for the specified number of iterations. Using Secure Hash Algorithm 2 (SHA-2), the result is at least 256 bits, which can be used as a key.

Exercise: Construct a small function that generates and returns a symmetric key from a password. The algorithm name for the symmetric key and the file name should be provided as an argument. For testing purposes, the algorithm should be provided by the user as a argument and you can use the following algorithms: Triple Data Encryption Standard (3DES), Advanced Encryption Standard 128 bits (AES-128) and Salsa20 variant ChaCha20 (ChaCha20).

To provide the password, check the `getpass` Python3 module.

3.2.2 File encryption

Create a function to encrypt the contents of a file, whose name should be provided by the user. The key should be provided by the previous function. The user must provide (as program parameters or by request or any other suitable method): (i) the name of the file to encrypt, (ii) the name of the file to store the cryptogram, (iii) the name of the encryption algorithm. The key must be requested when the program is executing.

If you need to save multiple fields to the same file (e.g, salt, cryptogram), save these as fixed length fields to the beginning of the file. A simpler alternative is to use Base64 to convert the objects to text (`base64` module), and then use a delimited such as `\n`.

Note 1: Take in consideration that data may need to be encrypted in blocks, and the last block may require padding. Please see the Public Key Cryptography Standards: Cryptographic Message Syntax Standard (PKCS#7) padding method in the documentation.

3.2.3 File decryption

Alter your program by adding a function that decrypts a user file. For this functionality, the user must provide the following (as program parameters or by request or any other suitable method): (i) the name of the file to decrypt, (ii) the name of the file to store the decryption result. The key must be requested.

Take care of removing the padding (if present!)

3.3 Cipher modes

3.3.1 Initialization Vector

Some cipher modes use feedback to add more complexity to the cryptogram, namely Cipher-FeedBack (CFB), Output-FeedBack (OFB) and CFB. Feedback implies the use of an **IV** which must be provided when initializing an object for one of such cipher modes. Note that the IV used to encrypt some data must also be provided when decrypting it, therefore, the IV is usually sent in clear text.

Alter your program so the user, when requesting an encryption operation, also indicates the cipher mode to be used. This should be applied both to encryption and decryption.

Note that the IV is only used on cipher modes with feedback, which is not the case of Electronic Code Block (ECB). Your program must be able to handle encryption using cipher modes both with and without feedback.

Hint: Use the `secrets` module to obtain securely random IVs.

3.3.2 Patterns

In this exercise we will analyze the impact of ECB and Cipher Block Chaining (CBC) cipher modes in the reproduction of patterns in the original document into the encrypted document. For this, we are going to encrypt an image in the bit map (BitMaP image file (BMP)) file format, after which we are going to visualize the contents of the obtained encrypted file and compare it with the original image. In order we can visualize the contents of the encrypted file we must replace the first 54 bytes with the first 54 bytes from the original file (these bytes constitute the header of BMP formatted files, which is necessary so the file can be recognized as a BMP formatted file).

Use the program you developed in the previous sections to encrypt the file `p_ori.bmp` using the ECB cipher mode and a cryptographic algorithm of your choice. Using the `dd` application copy the copy the first 54 bytes from the original image file into the first 54 bytes of the encrypted file:

```
$ dd if=p_ori.bmp of=p_enc.bmp ibs=1 count=54 conv=notrunc
```

Using a program to visualize images, open the original image and the encrypted image and compare them. What do you observe?

Repeat all the above operations, now using the CBC cipher mode instead of ECB cipher mode, and using the same algorithm. Then, compare the original image with the obtained encrypted image. What do you observe?

Repeat the experience, using the same cipher modes, but varying the algorithm. What do you conclude?

3.3.3 Corruption in the cryptogram

In this exercise we are going to analyze the impact in a decrypted text caused by errors in the cryptogram, when using ECB, CBC, OFB and CFB cipher modes.

Using the program developed in previous sections, encrypt the `p_ori.bmp` file using the ECB cipher mode and an algorithm of your choice. Using a binary file editor, change the value of a single bit in some byte of the encrypted image (notice that the first 54 bytes are not part of the image but rather part of the header of the file), for example the byte in position `0x60`.

Decrypt the file with the corrupted bit, using the same cipher mode and algorithm you used to encrypt. Using an binary file editor, open the original file and the decrypted file and compare them. What are your conclusions regarding the impact in the decrypted file produced by the corruption of a single bit in the cryptogram?

Repeat the experience for the remaining cipher modes and, for each of them, analyze the impact on the decrypted image produced by an error in a single bit of the cryptogram. Try to determine which are the cipher modes that produce a bigger impact and those that produce a smaller impact in the decrypted file.

3.3.4 Padding

A block cipher requires input blocks of a fixed size that equals the algorithm block size. However, its improbable that a file to encrypt as a number of bytes that is multiple of the block size of the algorithm to be used, i.e., frequently, the number of bytes that remain for the last block do not equals the block size. To solve this problem, extra bytes are added to have a block with the correct size. These extra bytes are then removed when in the decryption operation.

There are several standards for padding. PKCS#7 is one of them. The objective for this exercise is that you demonstrate the existence of padding in a encryption operation, and that it is according the PKCS#7 standard.

Using a binary file editor and the program you developed, idealize an experiment involving encryption and decryption operations with ECB cipher mode, Public Key Cryptography Standards: Password-based Encryption Standard (PKCS#5) padding and an algorithm of your choice, that shows the presence of padding and how PKCS#7 padding is made.

Hint: Do not use padding when decrypting a cryptogram with padding

3.4 Cryptographic hash functions

Create a program to produce the hash of the contents of a file. The user must provide the following input to the program: (i) the name of the file with the data to create the hash, and (ii) the name of the cryptographic hash function to use (Message Digest 5 (MD5), SHA-2 256 bits (SHA-256), SHA-2 384 bits (SHA-384), SHA-2 512 bits (SHA-512) and BLAKE-2 (BLAKE-2)). The program must print the hash in the screen, in hexadecimal format.

3.4.1 Avalanche effect

A very important requirement for cryptographic hash functions is that a small change in a text must produce a completely different hash – avalanche effect. In this exercise we are going to verify this requirement.

Select or create a file to calculate the hash. Using your program to obtain the hash of the file you just selected and save it in a file. Repeat the operation to obtain the hash of the same file, but using different cryptographic hash functions, and save the result hash in separated files.

Change one single bit of data in the source file you used above. Using this altered source file, obtain the hash produced by each of the cryptographic functions used in the previous step, and save them in separate files.

Compare the similarity of each pair of hashes produced by the same cryptographic hash function, and using source files differing only in a single bit.

3.4.2 Statistical analysis of avalanche effect

Based on the program you developed to calculate cryptographic hashes in section 3.4, create a new program to calculate the statistical distribution of the differences in the hashes of a set of messages that differ in one single bit from an original message. The program must receive from the user the following input: (i) the name of the file with the source message and (ii) the number of messages (N), differing one bit from the original message, to calculate the hash. The creation of single bit altered messages must use a random number generator to calculate the position of the bit to alter.

After the calculation of the hashes of all the N one-bit altered messages, evaluate the difference between each of these hashes and the hash of the original message, in terms of number of bits (Hamming distance, in bits). Comment the distribution of the differences that you obtained.

Hint: Use the XOR operation to detect the number of different bits between between two hashes (number of bits with the value 1 in the result of XOR operation).

3.5 Asymmetric cryptography

3.5.1 Key pair generation

Create a small program to generate anRivest–Shamir–Adleman (RSA) key pair, with a key length specified by the user, that could be one of the following values: 1024, 2048, 3072 and 4096. The program must save the key pair in two files, one for the private key and the other for the public key, whose names should also be specified by the user. The keys can be saved as binary blobs, but probably, the Privacy-Enhanced Mail (PEM) format will be more appropriate.

Run the program, several times, varying the key length. What do you think of using 4096 bit keys by default?

Hint: See `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/` and `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa`

3.5.2 RSA encryption

Create a small program to encrypt a file using theRSAalgorithm. The user must indicate the following data: (i) the name of the original file to encrypt, (ii) the name of the file with the public key, (iii) the name for the encrypted file.

Note: Pay attention to the size of the original file, i.e., the file to encrypt. Using the default configuration, the block size is equal to the key size minus eleven bytes (eleven bytes for padding). So, using 1024 bitsRSA key (128 bytes) the block size is 117 bytes (128 - 11).

3.5.3 RSA decryption

Create a small program to decrypt the contents of a file, whose name is provided by the user, using theRSA algorithm. The user must also indicate (i) the name of the file containing the public key to use, and (ii) the name for the file to save the decrypted content.

3.5.4 Different cryptograms?

Run theRSA encryption program you developed, to encrypt a file. Run again the program to encrypt the same file, using the same key, and save the encrypted content in another file. Using a binary file editor, open the two encrypted files you just generated and compare them. Are they similar or exactly the same?

Now use theRSA decryption program you developed, and decrypt the two encrypted files you generated. What is the result? Are the decrypted files equal to the original file?

Can you explain the reason for what you observed in this exercise?

3.5.5 How to encrypt a very big file?

Has you know, by now,RSA encryption is not efficient and is not used to encrypt data bigger than its block size. If you wish to see how inefficient it is, just take some data and measure the time it takes to encrypt and decrypt that data withRSA and AES.

Imagine you have a big file (500MBytes, for example) that you want to send to some person, with the guarantee that only that person can decrypt the file. More, you have the public key of that person, and you are not able to contact the person before sending the file. So, you have a big file to transmit to a person, from which you know the public key, but the process will be very slow if you encrypt the file using RSA. However, you can use any other encryption algorithm to encrypt the file, but remember you want that only the receiver must be able to decrypt the file.

Can you imagine some combination of encryption technologies that allows you to efficiently send the file to the other person, with the guarantee that only that person can decrypt the file?

A typical solution is called Hybrid Encryption, which combines two ciphers, a symmetric to encrypt the file with a random key, and a asymmetric to encrypt the key used in the previous step. Implement a function that applies this method.

3.6 Bibliography

- Python Cryptography: `https://cryptography.io/en/latest`
- Evgeny Milanov, "TheRSAAlgorithm": `https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf`
- Blake2: `https://blake2.net/`

- PKCS#5: Password-Based Cryptography Specification Version 2.0, RFC2898s: `https://tools.ietf.org/html/rfc2898`