Introduction to Cybersecurity Elisabeth Oswald

Symmetric Cryptography

2023

Overview

Some exercises are based on paper and pencil. Unless you use a tablet (with a pencil) you must bring actual paper and pencil along.

For some parts of this lab you need to program in Python, and we expect that you have a reasonable level of proficiency in Python already. We also expect that you have Python installed on your system. You will also need the Cryptography library as well as PyCryptoDome.

We do not expect that you can finish all exercises within the 3hr lab session. We also do not require that you do all exercises. The exercises are design to help you finding the answers to the corresponding Moodle quiz.

What you should get out of this.

Today's session's main goal is that you "use" a range of symmetric primitives in a somewhat "hands-on-manner" in order to deepen your understanding about them. The emphasis is here on "using schemes" rather than deep mathematical understanding: you are more likely in any job later on to become a "user of cryptography" than a cryptographer. As a user, sometimes this role is also called "Security Archtitect" your job is to design and specify how cryptography is used to achieve certain security goals. Therefore you need to understand what guarantees schemes give you, and you need to be aware that "small/innocent looking" modifications to schemes (standards) can have a devastating impact on security.

Assessment

There is no required submission for this lab, but doing the tasks will enable you to answer questions in a small Moodle quiz which is due at the end of next week.

Paper and Pencil Exercises

Exercise 1: Increasing key size

Sometimes older ciphers are fine in terms of the cryptographic strength of their round function, as well as the number of rounds, but they use keys that are too short in relation to brute force key search. To keep the same cipher, one must then consider ways of adding more key material, without changing the interface to the cipher. Consider the two following suggestions:

$$EncA_{k,k_1}(M) = Enc_k(M) \oplus k_1 \text{ and}$$
$$EncB_{k,k_1}(M) = Enc_k(M \oplus k_1).$$

The size of k is given by $|k| = 2^n$ and the size of k_1 is given by $|k_1| = 2^m$. What are the expected worst case complexities of EncA and EncB when attempting a brute force search of the key? You may assume that you have a moderate number of plaintext-ciphertext pairs, $C_i = EncA_{k,k_1}(M_i)$ or $C_i = EncB_{k,k_1}(M_i)$.

Exercise 2: Double encryption

One variant of triple encryption with DES using two rather than three 56 bit keys is defined in the following way:

$$C = DES(K_2, DES(K_1, DES(K_1, M)))$$

What is the worst case complexity of an exhaustive search against this variant? Show that this cipher succumbs to a meet-in-the-middle-attack (write down the sets/lists you compute and between which you look for an overlap). How much time/space does the attack need?

Exercise 3: Security notions for encryption

Explain, using your own words, what "IND", "OW", "PASS", "CPA", and "CCA" mean. Explain, again using your own words, why IND represents a strong notion of security than OW, and CCA is stronger than CPA. Finally, show that the CBC mode is not OW-CCA.

Exercise 4: Nonce vs IV?

In various modes of operation, as well as in the context of MACs and AE schemes the terms "nonce" and "IV" are used. I want you to clarify exactly what these terms mean, because they refer to different ways of incorporating "something" into a scheme.

- What is the difference between a nonce and an IV?
- Would a nonce based CBC mode for encryption of arbitrary length messages be secure?

Programming Exercises

Exercise 5: Modes of Operations

For this task we will be using the `PyCryptoDome` library. Open the script "ImageEncryption.py". It uses a number of libraries to work with files and do cryptography. I already included import statements that you will require when you work with other modes of operation. But as they are not used in the script as it is, they are greyed out in PyCharm.

The script takes a `.bmp` image that represents the Linux mascot "Tux". BMP is a simple image format, and I have produced a few lines of code, that open `tux.bmp`, and separate out the header information and the actual image. You don't need to really look into the details of how this works. What matters is that the actual plaintext that we want to encrypt is in the variable `msg`.

Encryption is implemented via PyCryptoDome's API. I instantiate an encryption box that uses ECB Mode (exactly how their documentation describes it), and then I produce the ciphertext. I produce an "encrypted" image by writing this back into a `.bmp` file. You can open the file and check out the resulting encrypted image. Is there something that you notice with respect to this image? Can you explain why it looks the way it does? Now use other modes of operations, and examine how the encrypted image looks like.

Exercise 6: MAC based Hash

You work on a project where cryptography has to be deployed on small embedded systems. As a consequence, one has to consider "savings" even in relation to code size. The token already must implement ECBC-MAC using a block cipher with block size 128: this is CBC mac using a second key k_2 to encrypt the final block again (see slide 19 in the slide deck under C vs I vs A).

Your team realise that not only a MAC is required to implement the necessary functionalities, but also a hash function must be available. To save on code size, the idea is born to use the ECBC-MAC as a hash function by fixing both keys k_1 and k_2 to be a constant value (and for simplicity this constant value is set to be 0).

Implement this construction using either the Cryptography or the PyCryptoDome library and show by implementing a concrete attack/example (for messages of at least 2 blocks) that it is not even pre-image resistant.

Exercise 7: One-time passwords based on hash functions

You are asked to help out with the implementation of a one-time password generator. The generator must work on a token that has a simple display. The display only has the encodings for 64 characters: "a-zA-Z0-9.," (so the entire Latin alphabet, plus the Arabic digits, plus two punctuation characters), and it can only display 6 such characters.

The one-time password system works as follows. Upon it's first power-on (this happens when the battery is inserted by the user), it samples 6 randomly chosen characters based on an in-built true random number generator. This is the inital password ot_0, and it is hashed using SHA-1 leading to a 128 bit hash value h_1. The hash value h_1 is truncated to the leadings 36 bits, which are then used to select the corresponding six characters from the set "a-zA-Z0-9.," to form the next password ot_1. The password ot_1 is displayed. The display will power off until the next use. When the token is activated again by the user, ot_1 is hashed to produce h_2, which is truncated to produce ot_2, etc.

Write the one-time password system down more formally, and implement it using either the Cryptography or PyCryptoDome library. You can choose how to "simulate" the user interaction. You should implement two functions: `Initialise` (sampling 6 random characters from the set "a-zA-Z0-9.,") and `GenNextPwd` (taking as input a 6 character password and producing a new 6 character password per your specification).

After approximately how many iterations do you expect that passwords will start to reappear with probability ≥ 0.5?

Find two different initial passwords ot_0 that produce a collision (in any of the password generated by `GenNextPwd`).