Assignment 2

Assigned: 8/11/2021
Due: 18/11/2021

- You should have already installed and start getting familiar with the **Ubuntu Mate 20.04 Operating System(OS) 20.04.** You can do this either:
  (1) with dual boot i.e. if you already have an OS such as MS Windows, MAC OS, or another linux distribution, just install Ubuntu Mate 20.04 as a secondary OS, or,
  (2) by installing first a Virtual machine (e.g. the Oracle VM VirtualBox is freely available) in your existing OS, and then within the VirtualBox installing the Ubuntu Mate 20.04, or,
  (3) by installing Ubuntu Mate 20.04 as your primary OS.
- If you prefer install the **Ubuntu 20.04 OS** instead of the Ubuntu Mate.
- You should have already installed the gcc compiler:

```
sudo apt update
sudo apt install build-essential
```

- Develop you C code using one of your favorite editors such as gedit, pluma or vi.

Introduction

In this assignment you are going to develop a symmetric encryption tool in C, using the OpenSSL toolkit https://www.openssl.org/. The purpose of this assignment is to provide you the opportunity to get familiar with the very popular general-purpose cryptography toolkit and acquire hands-on experience in implementing simple cryptographic applications. The tool will provide encryption, decryption, CMAC signing and CMAC verification functionality.

More specifically, you are going to use the EVP API, found in the OpenSSL toolkit, which provides a high-level interface to cryptographic functions. The cryptographic algorithm you are going to use is AES in Electronic Code Book (ECB) mode with both 128 and 256 bit modes.

Task A

[Key Derivation Function (KDF)]

In this task you have to implement a key derivation function. The function derives the symmetric key from a user-defined string (password). In order to generate the key, you will use the appropriate functions from the EVP API and the SHA1 cryptographic hash function. The KDF requires as arguments the password and the desired key size (128 or 256 bits) and generates a symmetric key of the appropriate size.

Task B

[Data Encryption]

Develop a function that provides AES-ECB encryption functionality, using 128-bit and 256-bit keys. This function reads the data of an input file and encrypts them using AES-ECB with the

key generated by the KDF described in Task A. Then, it stores the ciphertext to an output file. Use the EVP API in order to develop the encryption functionality.

Task C

[Data Decryption]

Using the EVP API, implement a function that reads a ciphertext from an input file and decrypts it with AES-ECB using 128-bit or 256-bit keys. The key will be generated using the KDF described in Task A. When the decryption is over, the function stores the plaintext in an appropriate output file.

IMPORTANT: In order to successfully decrypt the data, you have to generate the same key as the one used in order to encrypt them. For this reason the keys must be of the same size and derived from the same password.

Task D

[Data Signing (CMAC)]

For this task you have to implement a Cipher-based Message Authentication Code (CMAC) generation function, using the EVP and CMAC APIs. This function reads the plaintext data from an input file and encrypts them using the encryption function (Task B), using the key generated by the KDF (Task A), and then generates the CMAC. After the ciphertext and the CMAC are generated, it stores the ciphertext concatenated with the CMAC in an appropriate output file. Note_1: to generate CMAC you should use the plaintext. Note_2:

Task E

[Data Verification (CMAC)]

In this task you have to implement a CMAC verification function. This function reads the ciphertext concatenated with its CMAC from an input file. Then, it separates the ciphertext from the CMAC and decrypts the ciphertext. Using the plaintext obtained by the decryption, it generates its CMAC and compares it to the one that came with the ciphertext. The function returns TRUE if the CMAC is successfully verified and stores the plaintext in an appropriate file. Otherwise, it just returns FALSE.

IMPORTANT: In order to successfully decrypt the data, you have to generate the same key as the one used in order to encrypt them. For this reason the keys must be of the same size and derived from the same password. The same thing applies to CMAC generation and verification.

Task F

[Using the tool]

Once you have implemented all the above functionality for your tool, use it to do the following operations on the txt files provided:

1. Encrypt the file “encryptme_256.txt” using a 256-bit key. The key should be generated using TUC<AM> as password. For example “TUC2018123456”. The output file should be called “decryptme_256.txt”

2. Decrypt the file “hpy414_decryptme_128.txt” using a 128-bit key derived by the password “hpy414”. Store the plaintext in a file named “hpy414_encryptme_128.txt”

3. Sign the file “signme_128.txt” using a 128-bit key derived from a password in the format TUC<AM>. Store the ciphertext concatenated with its CMAC in a file named “verifyme_128.txt”

4. Verify the files “hpy414_verifyme_256.txt” and “hpy414_verifyme_128.txt” using the appropriate key size, as the filename specifies. The keys should be derived by the password “hpy414”. Write in your README.txt file which of those is successfully verified, if any.

Tool Specifications

To assist you in the development of this tool, we provide a basic skeleton of the tool. We also provide some helper functions, used to print the plaintext as a string and the bytes of the ciphertext, key and CMAC in a human readable form. Study carefully this basic skeleton in order to understand it first, then make the changes that are needed. The tool will receive the required arguments from the command line upon execution as such:

Options:

`-i`	`path`	Path to input file
`-o`	`path`	Path to output file
`-p`	`psswd`	Password for key generation
`-b`	`bits`	Bit mode (128 or 256 only)
`-d`		Decrypt input and store results to output
`-e`		Encrypt input and store results to output
`-g`		Encrypt+generate CMAC sign and store results to output
`-v`		Decrypt+verify input and store results to output
`-h`		This help message

The arguments “i”, “o”, “p” and “b” are always required. Moreover one of the “d”, “e”, “g” and “v” arguments should be provided.
Using -i and a path the user specifies the path to the input file.
Using -o and a path the user specifies the path to the output file.
Using -p and a password the user defines the string from which the AES key will be derived.
Using -b and the numbers 128 or 256 the user defines the desired key size.
Using -d the user specifies that the tool should read the ciphertext from the input file, decrypt it and then store the plaintext to the output file.
Using -e the user specifies that the tool should read the plaintext from the input file, encrypt it and store the ciphertext to the output file.
Using -g the user specifies that the tool should read the plaintext from the input file, encrypt it, generate its CMAC and then store the ciphertext concatenated with its CMAC to the output file.

Using -v the user specifies that the tool should read the ciphertext concatenated with its CMAC from the input file, decrypt it and store the plaintext to the output file only if the CMAC is successfully verified. Otherwise, it will inform the user that the ciphertext is not verified.

Example:
./assign_2 -i plaintext.txt -o ciphertext.txt -p hpy414 -b 256 -e

The tool will generate a 256-bit key from the password “hpy414” and use this key to encrypt the data found in “plaintext.txt” and then store the ciphertext to “ciphertext.txt”

IMPORTANT

Even if you chose to redesign your tool from scratch and not use the provided skeleton, you have to support the exact command line options described above.

Important notes

1. Install the `libssl-dev` package using your package manager.
2. You need to submit all the source code of your tool, a **“Makefile”**, all the files you used or generated in **Task F**, and a **“README.txt”** file that explains your implementation and what you didn’t implement and why. Please place all these files in a folder named <yourlastnameAM>_assign2, and then compress it as a .zip file that you will upload to eclass. For example: christodoulou2018123456_assign2.zip.
3. The README.txt file is important to submit, and it should describe briefly your tool and contain the answer of Task F.4.
4. Very important: execute the command `gcc --version` and write whatever the output is into your README.txt file, e.g. “gcc (Ubuntu 9.3.0-10ubuntu2~20.04)”
5. Study carefully the EVP and CMAC APIs (you can find many info online) and do **not** bother with asymmetric encryption.
6. Do **not** copy-paste code from online examples, we will know ;)
7. The tool’s skeleton provided with this assignment is just an example. Feel free to define your own functions or change the signatures of the given functions. You can even re-design it from the scratch.
8. The size of the CMAC you are going to generate is 16 bytes. Keep that in mind when separating the CMAC from the ciphertext upon verification.
9. The ciphertext, key and CMAC are just bytes. That means that they do not terminate with \0. Don’t try to print them as strings, use the provided function.
10. The ciphertext may be bigger than the plaintext due to padding. The AES block size is 16 bytes. Keep that in mind when allocating plaintext buffers.
11. Submitted code will be tested using plagiarism-detection software.
12. Make sure you support the exact command line options contained in the basic skeleton.