Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Security systems

Assignment No.2

Ομάδα LAB41446304

Τρίμας Χρήστος	***2016030054***

Purpose of Exercise:

Develop a symmetric encryption tool in C, using the OpenSSL development api. This assignment was more of documentation work, since the lines of code needed to complete it, were very few. The tool consists of five functions, each one implementing a different task.

Files:

In the **assign_1.c** file, one can find five functions that implement the tool, three helping functions that were created and implemented by me and finally the main function.

The **Makefile**, compiles and also runs the tool.

There is also a shell script, testing.sh, that helps the Makefile to run in a much more fancy way (I hate the fact that in makefiles the commands get printed in the console).

Implementation:

The assign_1.c file starts with the **keygen** function that generates a key using the users password. To do so, two functions were used, the EVP_sha1() and EVP_BytesToKey().

EVP_sha1(): The message digests SHA1. The SHA1 algorithm is a cryptographic hash function, which takes an input and produces a hash value, known as digest message.

EVP_BytesToKey(): It is a password based encryption routine. To be more specific, the function derives a key (and an iv) using the cipher, a hash function, the user's password and the length of that password.

The **encryption** uses the function encrypt(). First, it defines a new ctx using EVP_CIPHER_CTX_new(). Then, the EVP_EncryptInit_ex(), EVP_EncryptUpdate(), EVP_EncryptFinal_ex() does all the symmetric encryption for us.

EVP_CIPHER_CTX_new(): This function creates a cipher context. For success the function returns a pointer to a newly created EVP_CIPHER_CTX otherwise, it returns NULL.

EVP_EncryptInit_ex(): sets up cipher context for encryption using context, the cipher and the key.

EVP_EncryptUpdate(): encrypts bytes from the input buffer and writes the encrypted version to the output.

EVP_EncryptFinal_ex(): since I am using default padding, this function encrypts the final data that is any data that remained in a partial block.

For the **decryption** phase I am using the same commands, except that I am decrypting the input this time. Functions such as:

EVP_DecryptInit_ex(): same as Encrypt but for the decryption phase.

EVP_DecryptUpdate(): same as above, but for the decryption phase.

EVP_DecryptFinal_ex(): again same as above but for decryption.

The tool also **generates** a **CMAC** using the function gen_cmac(). After I created the context, I simply use the CMAC_Init(), CMAC_Update() and CMAC_Final() to generate successfully the cmac.

CMAC_Init(): selects the given block cipher for use by context.

CMAC_Update(): processes bytes (size of input length) of data input.

CMAC_Final(): stores the length of the message authentication code in bytes, which equals to the chipper block size.

For the verification process, I simply use the memcmp() function to compare the n bytes of cmac1 with the n bytes of cmac2. Where n is the pre-defined BLOCK_SIZE.

Finally, the three helping functions that I developed, are:

readText() and writeText(): both functions were developed in COMP101 to read and write files, nothing impressive here.

bit_check(): Depending on the bit mode, the function returns and EVP_CIPHER struct that is encrypted using the aes 128 bit or 256.

Run:

For easy testing I also created a bash script that produces all the files for the TASK F, by typing the AM of a student. To run that do the following:

- $: make delete
- $: make test

This runs the tests.sh shell script and creates all the test files.

For Task F.4, neither of the files were successfully verified therefore, no files were created. In an attempt to check what was wrong, I let the verification to pass and then compared the

files using the cmp and diff shell commands. The files were different in byte size and of course in content.