Exercise Report

General Description

Multitool Encryptor was supposed to be a file encryption program that would allow file encryption using different crypto-toolkit APIs, namely EVP (OpenSSL), Nettle and libsodium. Sadly, debugging a bug encountered in the EVP decryption method ended up taking so much time that only OpenSSL was finished, and I was never able to get rid of the bug, despite hours of trying. The program does compile though, and as I used Qt to develop it, compiling with qmake is recommended. After compiling, the program can be used to encrypt/decrypt a file using the command line, in the following manner:

./MultitoolEncryptor [encrypt/decrypt] [filepath]

The program then asks the user for a password. The password is used to generate the key and the IV using PKCS5_PBKDF2_HMAC with SHA-256. The algorithm used for encryption is AES-256-CBC. The encrypted file (named Encrypted.enc) is saved in the same directory as where the program is run from. The original file is unchanged. Decryption creates a new file in the directory, called Decrypted. The key and IV used are shown in the terminal during encryption/decryption, though this is because it was helpful for debugging, and it stayed to show that the decryption-bug isn't caused by a faulty key/IV.

Structure of the Program

The main-function is written in C++, while the other source files are written in C. C was chosen mostly because I wanted to learn more about it, as I have barely used it before. The actual encryption/decryption-functionality is in multitool_evp.c. multitool_c_util.h and .c introduce structures used in the encryption process, and a cleanup-function used to free memory in the case of errors during encryption/decryption.

Secure Programming Solutions

The C-code extensively checks for failures, and will exit safely if failures are encountered. The password is retrieved with fgets to avoid buffer overflows. The main-function is very selective about it's input, so misuse potential should be minimal.

Security Testing

Manual security testing didn't reveal any issues. The selective main function ensures that the program doesn't do anything unless proper parameters are given. The program also doesn't delete or alter the original file as it performs an encryption. Clang was used as a static analyzer, but it didn't report any issues.

Security Issues

The salt used for generating the key and the IV is hard-coded. While this isn't recommended, it does prevent using pre-computed rainbow-tables. User input while getting the password isn't sanitized, which could lead to security issues.

Lessons Learned

Better time management at least. Also, my Virtual Machine ended up breaking (more specifically, my Ubuntu installation ended up in a boot loop), which did result in lost progress, though that could have been avoided with proper use of git. One interesting thing I happened to learn while trying to debug the decryption, was how to properly utilize a salt in the key derivation process. The salt is supposed to be randomly generated for each encryption process, which leads to unique keys and IVs even if the same password is used. However, since the key derivation function needs to be able to generate the same key/IV for decrypting a file, it also needs the same salt used for encrypting said file. A good method of handling this wasn't immediately obvious, but since the salt doesn't need to stay secret, it can be included in the header of the encrypted file, from where it can then be retrieved to compute the decryption key/IV. I believe this is how OpenSSL handles this, though I was unable to find any sources.