CS 4640: Computer Security **Spring 2017**

Programming Project 1
Total Points 100%

This homework will contribute **15%** to your total grade

Due date & time: 11:59 pm on April 23rd, 2017. Submit the Programming Project through ICON. Your submission folder will be a directory with the name ⟨your-last-name⟩-⟨your-first-name⟩-homework-3.

Late Policy: You have three extra days in total for all your homework assignment and projects. Any portion of a day used counts as one day; that is, you have to use integer number of late days each time. If you exhaust your three late days, any late homework won't be graded.

Objective: For your previous homework, you have implemented the random one-time pad and its approximation (i.e., stream cipher) RC4. I have mentioned in class that it is often futile to design or implement cryptographic constructs. This is a sound advice, for most of us, because often in-house designed cryptographic constructs do not have the theoretically proven, rigorous, and well-vetted security assurances that the existing well studied constructs have. Furthermore, implementing an existing construct adds another level of challenge because the security of some of the existing constructs requires carefully chosen parameter values which are not often mentioned in the initial specification. Overall, it is always a good idea to use a well-established cryptographic library which has been vetted by experts and also attackers. **The overarching goal of this project is to give you hands-on experience on using cryptographic constructs and libraries**.

Participants: This is a group project. I **highly recommend** doing it in a group of two people. If you do it alone, please note that the workload may be high and you will be responsible for the consequences. You can choose your group members.

Problem: *This project requires you to design and develop a* ***simple but secure*** *password manager with 256-bit security.* Please read the security requirement and adversary model section to understand what I mean by a secure password manager.

Note that your password manager does not need to auto-fill your password fields in the browser. You will run your password manager from the command line or terminal.

Adversary Model: You can assume the adversary can read the file your password manager uses to store the username, domain, and password of each account. You can also assume that the adversary may modify the password file (i.e., passwd_file) in an arbitrary fashion. You can safely assume that the adversary will not delete the files.

Deliverables: You are going to provide 3 electronic documents along with your full source code. The first document will state the compilation process of your source code from the terminal or command prompt. The second document will detail your design and your choice of implementation-level functions and constructs with necessary information. It should also argue how your design provides the security guarantees expected. For instance, what is the format of your password file, and what is your IV length and which functions did you use to generate them? The final document will precisely state the work division among the group members.

Programming Languages allowed: For this programming project, you can use C/C++, Java, and Python. Please note that the instructor is not proficient in Python. Hence, the instructor may not be able to provide language level assistance if you are using Python.

Cryptographic Libraries to use: Depending on the language of your choice use one of the following cryptographic libraries: C/C++ (OpenSSL), Java (BouncyCastle), Python (OpenSSL). For groups using C/C++/Python on MAC OSX, please use the Common Crypto Library (`https://opensource.apple.com//source/CommonCrypto/`) as it is often difficult to install OpenSSL for MAC OSX.

Cryptographic Constructs to use: For all your encryption purposes, please use AES with CTR (counter) mode with 256-bit block size. For hashing, you should use SHA512. For HMAC or MAC, you can use any built-in functions provided by the respective cryptographic libraries. When generating keys and IVs, please make sure you use a secure pseudo-random number generators (PRNG) provided by the cryptographic libraries. Please do not use the PRNGs provided by the standard libraries of the language you are using; they are often not secure.

Functionality expected from your password manager: The password manager you are required to design will interact with the user through the standard input. When you first run your password manager, it is going to look for two files named "passwd_file" and "master_passwd," respectively, in the same directory, the executable is executing. If it cannot find any of those files, your program will assume that the user has not registered yet, and the password manager is running for the first time. In that case, the user will be given a prompt to submit a master password. After the master password has been obtained from the user, the password manager will generate two files: one file named "passwd_file" where it will store all the encrypted account information; the other file named "master_passwd" where it will store a 256-byte salt followed by the master password hashed with the salt.

After this registration phase, every time you execute the password manager, it is going to first prompt the user for the master password. It will then check whether the password matches according to the salted hash. If it does not match, your program should output the following error message without quote signs in the standard output: "WRONG MASTER PASSWORD!\n". *Note that your program should not recreate either the passwd_file or master_passwd file after the registration phase.*

It will then perform an integrity check as to verify that no one has modified the passwd_file in an unauthorized fashion. If it fails the integrity check, your program should output the following error message without the quote signs in the standard output: "INTEGRITY CHECK OF PASSWORD FILE FAILED!\n". If in the case the integrity check passes, your program should not print anything to the standard output.

Then your password manager will wait for input commands from the user. The user can pose one of the following commands. When your program receives one of the following commands, it will perform the functions as described below. For processing commands, you may need to create `check_integrity`: Your program should check the integrity of the password file and output either "PASSED!\n" or "FAILED!\n" in the standard output without the quotes.

`register_account`: Your program should take as input the username, password, and the domain name, and should store it in the passwd_file encrypted. You can safely assume that username, password, and the domain name will have a length less than 80 characters. Note that when you register for a new account, you should re-encrypt the whole passwd_file with a new random IV. If you are registering for an account that already exists (if all of the following matches: username, domain), your program should output the following message without quotes in the standard output: "USER ACCOUNT ALREADY EXISTS!\n". You do not need to print anything when the operation is successful.

`delete_account`: Your program should take as input the username, password, and the domain name to delete, and should delete it from the passwd_file. Note that when you delete an existing account, you should re-encrypt the whole passwd_file with a new random IV. If you are trying to delete an account that does not exist, your program should output the following message without quotes in the standard output: "USER ACCOUNT DOES NOT EXIST!\n". You do not need to print anything when the operation is successful.

`change_account`: Your program should take as input the username, the old password, the new password, and the domain name to delete, and should change the old_password with the new_password in the passwd_file. Note that when you change an existing account, you should re-encrypt the whole passwd_file with a new random IV. If you are trying to change the password for an account that does not exist, your program should output the following message without quotes in the standard output: "USER ACCOUNT DOES NOT EXIST!\n". You do not need to print anything when the operation is successful.

`get_password`: Your program should take as input the domain name, and should print the username and the password for that domain obtained from the passwd_file. Your program should print the username, password in the following format without quotes in the standard output: "username<SPACE><username><SPACE>password<SPACE><password>\n" where <username>, <password>, and <SPACE> are the actual username, password, and the space character, respectively. If you are trying to retrieve the password for an account that does not exist, your program should output the following message without quotes in the standard output: "USER ACCOUNT DOES NOT EXIST!\n".

Hint and Other Information: There is a question of how would you connect a master password to an encryption key. There are secure algorithms for generating encryption keys from textual passwords. For instance, you can use PKCS5_PBKDF2_HMAC from OpenSSL (`https://wiki.openssl.org/index.php/Manual:PKCS5_PBKDF2_HMAC(3)`). Such functions are available for other libraries too.

Please do not use any insecure cryptographic constructs or constructs which have been proven to be insecure, e.g., SHA1, MD5, RC4.

Assume all the passwords, domain names, and also the usernames that can be submitted by the user can contain all printable ASCII characters except the space character and '!'.

Security guarantees expected from your design: The first security guarantee expected from your design and implementation is the confidentiality guarantee. Precisely, anyone without the correct master password should not be able to know any one of the following information: (1) Which accounts you have registered for; (2) Username of any of the accounts you have registered for; (3) Passwords of any of the registered accounts. This security guarantee dictates that the file in which your password manager stores the password should be encrypted. Note that we want a 256-bit security of our system.

The second security requirement your design and implementation should provide is integrity guarantees. More precisely, if anyone has modified your file, you should be able to detect it. This security guarantee dictates that you should use some form of message authentication code (MAC) or Merkle hash tree.

Your design does not need to protect the integrity of the contents of the master_passwd file. You can assume there is some other way to protect this file.

Note that your design does not need to provide freshness guarantees. A password manager with freshness guarantee ensures that if the adversary replaces the current version of your password file with an old version, it will be able to detect it.