CS526: Information Security **Fall 2015**

Project #1 - Simulated Encrypted File System (SEFS)

Due date & time: Email the source code and the report to the TA (`chen623@purdue.edu`) by **11:59pm (evening) on October 15, 2015**.

Late Policy: You have three extra days in total for all your 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 project won't be graded.

Additional Instructions: (1) This project worths 10% of the course grade, and you can work together as a team of two. (2) The submitted report must be typed. Using LaTeXis recommended, but not required.

1 Summary

In a traditional file system, files are usually stored on disks in plaintext (*i.e.*, unencrypted). When the disks are stolen by someone, contents of those files can be easily recovered by the perpetrator. Files, belonging to large corporations, being stolen by hackers are very commonplace now-a-days (*e.g.*, Sony Hack, Target data breach). Encrypted File Systems (EFS) are developed to protect individuals and organizations from such leakages. In an EFS, files on disks are all encrypted, nobody can decrypt the files without the information of the required secret. Therefore, even if a EFS disk is stolen, or if otherwise an adversary can read files stored on the disk, its files and their contents are kept confidential. EFS has been implemented in a number of operating systems, such as Solaris, Windows NT, and Linux.

In this project, you are asked to implement a simulated version of encrypted file system (SEFS) in C. It is evident from the description so far that you will need to use cryptographic libraries to encrypt the files in the disk. One of the takeaway messages during the cryptography lectures was: *Do not write your own cryptographic library and instead use well-known cryptographic libraries that underwent heavy scrutiny from the academic community and practioner.* The current project puts this principle into practice. One very well-known cryptographic library, that is battle hardened, is OpenSSL (`https://www.openssl.org/`). One of the primary goals of this project is to familiarize you with the OpenSSL library and how to effectively use it to do different cryptographic operations (*e.g.*, encryption, decryption, calculating message digests, calculating HMACs).

2 Project components and Points division

Project components. This project is divided into 3 parts. In the part 1, you are required to implement a password management system for user authentication. In part 2 of the project, you are required to implement a simplified version of SEFS where you use the password management system developed in part 1 for the user authentication mechanism. Finally, in part 3 of the project, you will be required to built a more elegant version of SEFS by generalizing the simplified SEFS developed in part 2 of the project.

Points distribution. The part 1 of the project (and, the accompanying questions if any) will be worth 20% of the whole points of the project. The part 2 of the project (and, the accompanying questions if any) will be worth 30% of the whole points of the project. The final part of the project (and, the accompanying questions if any) will be worth 45% of the whole project points. Finally, 5% of the project points will be given for inter-operability. More precisely, if you use your tool to encrypt a file then the other groups' correct implementation of the tool should be able to decrypt the file and vice versa. This is to mimic the software development environment where your tool should operate properly with existing tools. For instance, A JPEG file exported by Photoshop should be viewable and editable in GIMP. Additionally, this is to increase the interaction between your classmates.

3 Description: Part 1 – User Authentication using Passwords

In this part of the project, you are required to develop a user password management system. You can assume there exists a password file in the disk with name "**passwd**".

For this part, we assume username should have a length greater than 5 but less than 32 characters. The allowed characters in the username are: "a-zA-Z0-9". The password generated by the user should have a length greater than 8 but less than 32 characters. The allowed characters in the password are: "a-zA-Z0-9@#$%&*()-+=". Each line of the **passwd** file contains information about the password of one user. More precisely, each line has the form: "username:salt:hPassword". The character ':' is used as the separator. username is kept in plaintext. salt is the random salt for hashing the password. For each user the salt will be generated randomly. A utility function will be provided to generate a random salt. Instead of the salt's binary representation, we will store the hexadecimal representation of the salt. A utility function will be provided that will convert a byte array to its hexadecimal representation. The hPassword represents the hexadecimal representation of the hashed password of the user with respect to the salt. For generating the hashed password with respect to the salt, we will use the `PKCS5_PBKDF2_HMAC_SHA1` library function of OpenSSL. A sample usage of the function will be given to you.

A standalone skeleton program in C will be given to you. You will be asked to fill out the following functions.

1. **Register a user.** The function `register_user`$(u, p, pFile)$ takes as input a user name u, a password p, and a password file name $pFile$ and tries to create an entry (*i.e.*, username:salt:hPassword) for user u with the password p in the file $pFile$. If the user is already present, then your function will return '-1' (meaning error). In case, the user name or the password contains invalid characters or the password file is not present then your function will also return -1. When the user and its associated password can be successfully created, then your function should return 1.
2. **Delete a user.** The function `delete_user`$(u, pFile)$ takes as input a user name u and a password file name $pFile$ and tries to delete an entry (*i.e.*, username:salt:hPassword) for user u in the file $pFile$. If the user is **not** present, then your function will return '-1' (meaning error). In case, the user name contains invalid characters or the password file is not present then your function will also return -1. When the user is successfully deleted, then your function should return 1.
3. **Is user valid.** The function `is_user_valid`$(u, pFile)$ takes as input a username u and a password file $pFile$, and checks to see whether the user is present in the password file. If the user u is present, your function should return 1 or for any error it should return -1.
4. **Match user-password.** The function `match_user`$(u, p, pFile)$ takes as input a username u, a password p, and a password file $pFile$, and checks to see whether the user u has the password p. On success your function should return 1 or otherwise return -1.
5. **Change user-password.** The function `change_user_password`$(u, p_c, p_n, pFile)$ takes as input a username u, his current password p_c, his new password p_n, and a password file $pFile$, and tries to replace u's current password p_c with the new password p_n. On success your function should return 1 or otherwise return -1.

4 Description: Part 2 – Simplified SEFS

You are required to design a simplified SEFS for this part of the project. For user authentication, you will use the user-password management functions written in the Part 1 of the project. You can assume the simplified SEFS has access to a 128-bit master key and 256 bit master initialization vector (IV). For testing you will be provided a sample master key file containing the hexadecimal representation of a master key and an IV. You will be also given a skeleton file for this part of the project which takes as input a master key file and a password file. The skeleton file (or, program) loads the master key and the IV to the memory. You

will also be given master key file generator which generates a random 128-bit key and 256 bit IV and writes their hexadecimal representations to a file of your choice.

For each plaintext file f, after encryption your simplified SEFS will generate two files named "**f.meta**" and "**r.chunk**" where r is a name of your choice only containing alphanumeric characters (*i.e.*, a-zA-Z0-9) and can have maximum 20 characters in length. For example, if you encrypt a plaintext file named **email**, it will generate two files: **email.meta** and **random.chunk**. Let us assume the file **email** contains a single line "AnEmailFromMeToYou". To encrypt the file and generate the **random.chunk**, you should use AES in the CTR mode. You should use a random key of 128 bit length for encryption. The initialization vector (IV) used for encryption should be randomly generated. The first line of the encrypted file (*i.e.*, **random.chunk**) will contain the hexadecimal representation of the IV in plaintext. From next line on it will contain the hexadecimal representation of the ciphertext of (next_chunk||size_of_data||content_of_file). In the example, size_of_data is 18. For this part of the project, we will assume next_chunk only contains the value "NULL". Hence, the file **random.chunk** will contain the hexadecimal representation of IV in the first line and then from the second line will contain the hexadecimal representation of the ciphertext AES_CTR(IV, Key, NULL||18||AnEmailFromMeToYou).

The meta file (*e.g.*, **email.meta**) contains auxiliary information regarding the encrypted file (*e.g.*, **random.chunk**). For this part of the project, the meta file will contain 6 lines. Line 1 of the meta file will contain the username of the owner of the file. Line 2 will contain the value of number of chunks that the encrypted file has been splitted into. For this part of the project, the value to use is only 1. For the next part of the project, it can contain other values. Line 3 will contain the size of the plaintext file (*e.g.*, **email**) in bytes. Line 4 will contain the name of the starting chunk of the file. Line 5 will contain the name of the ending chunk of the file. For this part of the project, both line 4 and 5 will contain the same value. For the example above, it will contain the value **random.chunk**. The line 6 contains three values chunk_name, chunk_key, chunk_hash_mac, separated by the space character (*i.e.*, ASCII character 32). The chunk_name field contains the chunk name of the encrypted file which for the above example is **random.chunk**. chunk_key is the hexadecimal representation of a random 128-bit key which is used to encrypt the contents of the file **random.chunk** (*i.e.*, next_chunk, size_of_data, and plaintext_file_content). chunk_hash_mac is the hashmac of the contents of **random.chunk** with respect to the chunk_key. For performing hashmac we will use the OpenSSL's `HMAC` function with sha256 (*i.e.*, `EVP_sha256()`) as the choice of the hash function. The meta file will be encrypted with AES in the CTR mode with the master key and IV.

Finally, we also assume that the simplified SEFS has an encrypted file named **master.filelist**. The plaintext version of this file contains multiple lines. Each line contains a file name, currently secured by the simplified SEFS, and the sha256 digest of the file's (encrypted) meta-file contents.

You are required to implement the following file operations in the skeleton file.

1. **File creation.** The function `create_file`$(u, p, fName)$ takes as input a user name u, his password p, a file name $fName$, and tries to create an encrypted file with name $fName$. On success the function returns 1 and returns -1 on errors. Example errors include a file with the same name already exists.
2. **File deletion.** The function `delete_file`$(u, p, fName)$ takes as input a user name u, his password p, a file name $fName$, and tries to delete an encrypted file with name $fName$. On success the function returns 1 and returns -1 on errors. Example errors include a file with the same name does not exist or the user u is not the owner of the file.
3. **Encrypting a plaintext file.** The function `encrypt_file`$(u, p, fName)$ takes as input a user name u, his password p, a plaintext file name $fName$, and tries to create an encrypted version of the file with name $fName$. On success the function returns 1 and returns -1 on errors. Example errors include such a plaintext file with the name does not exist or the user-password does not match up.
4. **Decrypting an encrypted file.** The function `decrypt_file`$(u, p, fName, pFName)$ takes as input a user name u, his password p, an encrypted file name $fName$, a plaintext fileName $pFName$ and tries to create a plaintext file with name $pFName$ by decrypting the encrypted file $fName$. On success the function returns 1 and returns -1 on errors. Example errors include such an encrypted file with the name does not exist or the user is not the owner of the file.

5. **Read from an encrypted file.** The function `read_from_file`$(u, p, fName, position, len)$ takes as input a user name u, his password p, an encrypted file name $fName$, an offset *position*, a length *len*, and it tries to read the *len* number of characters from byte position *position* from the plaintext content of the file. It then returns those characters. On error it returns a NULL character pointer.
6. **Write to an encrypted file.** The function `write_to_file`$(u, p, fName, position, nContent)$ takes as input a user name u, his password p, an encrypted file name $fName$, an offset *position*, a character pointer *nContent*, and it tries to write the contents of the character pointer *nContent* from byte position *position* in the file $fName$. On success it returns 1 and on errors it returns -1. **While writing the new data, a new chunk key and IV should be generated and used for encryption.**
7. **Size of a file.** The function `file_size`$(u, p, fName)$ takes as input a user name u, his password p, an encrypted file name $fName$, and tries to return the size of the file in bytes. On error, the function returns -1. Example errors include the user is not the owner of the file or such a file does not exist.
8. **File integrity checking.** The function `file_integrity_check`$(u, p, fName)$ takes as input a user name u, his password p, an encrypted file name $fName$, and tries to check the integrity of the file. On success the function returns 1 and on errors it returns -1.
9. **System health checking.** The function `system_health_check`() tries to check the integrity of all the files currently managed by the SEFS system. On success the function returns 1 and on errors it returns -1.

5 Description: Part 3 – SEFS

This part of the project requires extending the simplified the SEFS developed in the part 2 of the project. The main extension is that instead of storing the whole file in one chunk, we will split the file into multiple chunks. Each chunk will be encrypted separately with a different random key and IV. Each chunk can contain a maximum of 1024 bytes of the plaintext file content. In case, for a chunk the remaining plaintext file content is less than 1024 bytes, then you should pad it with the character 0 (ASCII value 0).

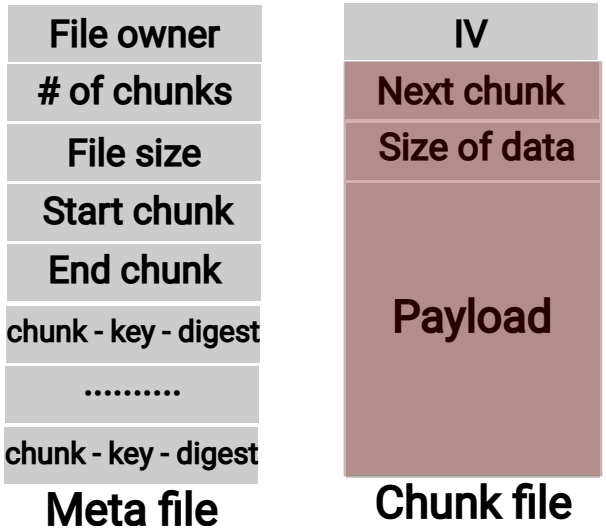

Figure 1: Format of meta file and chunk file

Extension to the meta file and the chunk. The chunks of a file can be viewed as a linked list (or, simply a chain). The start and end chunks are stored in the meta file. Each chunk's next_chunk field contains the name of the next chunk in the chain. Additionally, instead of having only one line containing chunk_name, chunk_key, chunk_hash_mac in the meta file, the meta file can possibly contain multiple lines of chunk_name, chunk_key, and chunk_hash_mac, one for each chunk. Hence, the meta file for this part of the project can possibly have variable number of lines instead of a fixed number of lines (*e.g.*, 6 for the simplified SEFS in part 2).

Space efficiency restriction. You are required to use space efficiently. For instance, if you have a chunk which contains 512 bytes of plaintext data and the user issued a write command that asks you to write new 200 bytes of plaintext data after the 512 bytes, you cannot just create a new chunk and write the new 200 bytes in that new chunk. Instead you have to write the new 200 bytes in the same chunk containing the 512 bytes. The overall goal is to use the minimum number of chunks for representing a file.

You are required to implement the following file operations in the skeleton file with the new requirements of Part 3.

1. **File creation.** The function `create_file`$(u, p, fName)$ takes as input a user name u, his password p, a file name $fName$, and tries to create an encrypted file with name $fName$. On success the function returns 1 and returns -1 on errors. Example errors include a file with the same name already exists.
2. **File deletion.** The function `delete_file`$(u, p, fName)$ takes as input a user name u, his password p, a file name $fName$, and tries to delete an encrypted file with name $fName$. On success the function returns 1 and returns -1 on errors. Example errors include a file with the same name does not exist or the user u is not the owner of the file.
3. **Encrypting a plaintext file.** The function `encrypt_file`$(u, p, fName)$ takes as input a user name u, his password p, a plaintext file name $fName$, and tries to create an encrypted version of the file with name $fName$. On success the function returns 1 and returns -1 on errors. Example errors include such a plaintext file with the name does not exist or the user-password does not match up.
4. **Decrypting an encrypted file.** The function `decrypt_file`$(u, p, fName, pFName)$ takes as input a user name u, his password p, an encrypted file name $fName$, a plaintext fileName $pFName$ and tries to create a plaintext file with name $pFName$ by decrypting the encrypted file $fName$. On success the function returns 1 and returns -1 on errors. Example errors include such an encrypted file with the name does not exist or the user is not the owner of the file.
5. **Read from an encrypted file.** The function `read_from_file`$(u, p, fName, position, len)$ takes as input a user name u, his password p, an encrypted file name $fName$, an offset *position*, a length *len*, and it tries to read the *len* number of characters from byte position *position* from the plaintext content of the file. It then returns those characters. On error it returns a NULL character pointer.
6. **Write to an encrypted file.** The function `write_to_file`$(u, p, fName, position, nContent)$ takes as input a user name u, his password p, an encrypted file name $fName$, an offset *position*, a character pointer *nContent*, and it tries to write the contents of the character pointer *nContent* from byte position *position* in the file $fName$. On success it returns 1 and on errors it returns -1. **While writing the new data, new chunk key(s) and IV(s) should be generated and used for encryption.**
7. **Size of a file.** The function `file_size`$(u, p, fName)$ takes as input a user name u, his password p, an encrypted file name $fName$, and tries to return the size of the file in bytes. On error, the function returns -1. Example errors include the user is not the owner of the file or such a file does not exist.
8. **File integrity checking.** The function `file_integrity_check`$(u, p, fName)$ takes as input a user name u, his password p, an encrypted file name $fName$, and tries to check the integrity of the file. On success the function returns 1 and on errors it returns -1.
9. **System health checking.** The function `system_health_check`() tries to check the integrity of all the files currently managed by the SEFS system. On success the function returns 1 and on errors it returns -1.

6 Notes

- username can only contain characters from the class "a-zA-Z0-9". The length of username will be in the range [6,31].
- password can only contain characters from the class "a-zA-Z0-9@#$%&*()-+=". The length of password will in the range [9,31].
- salt must be randomly generated and must be of length 256 bits (*i.e.*, 32 bytes).
- For generating hashed password use the function `PKCS5_PBKDF2_HMAC_SHA1` with iteration value 20000.

- The master key is of size 128 bits (*i.e.*, 16 bytes) and will be given to your program.
- All keys except the master key should be randomly generated. They should be 128 bits in length.
- For encryption, use the AES in the CTR mode.
- Initialization vectors (IV) should be randomly generated and must be 32 bytes in length.
- Chunk names can be randomly generated and cannot contain the space character in it.
- For hash mac use the `HMAC` function with sha256 (*i.e.*, `EVP_sha256()`).
- For any kind of padding use the character with ASCII value 0.

Supplementary materials. Example usages of different cryptographic primitives of OpenSSL relevant to this project, tutorials of OpenSSL, the skeleton source codes, Makefiles, and the different utility functions (*e.g.*, generating random bytes, converting byte array to its hexadecimal representation, converting hexadecimal representation to byte array) will be provided through piazza.