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 using the `RAND_bytes` openSSL library function (See the next section to see how to use `RAND_bytes`). Instead of the salt's binary representation, you will store the hexadecimal representation of the salt. For this you will use the utility function `byte_array_to_hex_array`. It is described in the next section. 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 is shown in the next section. Please use 20000 as the iteration value for generating hashed passwords.

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`$(u, p, pFile)$ takes as input a user name u, a password p, 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. If the password does not match then 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. Your program should following as command line arguments (in the following order): the master password, the password file name, and date and time of last edit of the system. The date and time will be used to check the integrity of the master filelist file. You will be provided a new skelton file for Part 2 and Part 3 of the project.

Master keys. For the next two parts of the project, you will be required to create a 128-bit master encryption-decryption key and a 128-bit master hmac-key. Both the master encryption-decryption key and the master hmac-key will be derived from a master password which you cannot save (*e.g.*, you cannot save it in a file or hardcode in your source file). For generating the master encryption-decryption key and the master hmac-key from the master password you will be required to use the key derivation function from openSSL (*i.e.*, `PKCS5_PBKDF2_HMAC_SHA1`). An example usage of this function for the purpose of key derivation can be found in the following url: `http://www.mail-archive.com/openssl-users@openssl.org/msg54143/pkcs5.c`. For your convenience, the example key derivation program is shown below.

```
#include <string.h>
#include <openssl/x509.h>
#include <openssl/evp.h>
#include <openssl/hmac.h>
int print_hex(unsigned char *buf, int len){
        int i, n;
        for(i=0,n=0;i<len;i++){
                if(n > 7){
                        printf("\n");
                        n = 0;
                }
                printf("0x%02x, ",buf[i]);
                n++;
        }
        printf("\n");
        return(0);
}
int main(){
        char *pass = "password";
        char *salt = "12340000";
        int iteration_count = 1;
        unsigned char buf[1024];
        iteration_count = 1;
        PKCS5_PBKDF2_HMAC_SHA1(pass, strlen(pass), (unsigned char*)salt, strlen(salt),
            iteration_count, 32+16, buf);
        printf("PKCS5_PBKDF2_HMAC_SHA1(\"%s\", \"%s\", %d)=\n", pass, salt,
            iteration_count);
        print_hex(buf, 32+16);
        return(0);
}
```

The example usage shows how to derive a 32-byte key and a 16-byte IV from a master password and salt. Note that, your program does not take the salt as input. Hence, for the salt value you are required to use the following fixed 32 byte string (without the quotes): "PURDUECS526CLASSFALL2015WLINUSA0". You will use the `PKCS5_PBKDF2_HMAC_SHA1` function to derive a 256-bit (*i.e.*, 32-byte) key. The first 128 bits (*i.e.*, 16 bytes) of the derived key will be used as the master encryption-decryption key whereas the remaining 128 bits (*i.e.*, 16 bytes) will be used as the master hmac-key. For the purpose of key derivation, use the iteration count to be 1. Note that, for generating hashed passwords, you are however required to use the iteration count to be 20000. This is to slow down the adversary from cracking the passwords.

Other keys. Except the master key all the other (encryption-decryption or hmac) keys should be generated randomly. For generating random keys use the `RAND_bytes` function from openSSL. The function takes two arguments: pointer to the container where to hold the random data (*e.g.*, `*ptr`) and the number of random bytes it should generate (*e.g.*, `num_bytes`). Note that, you have to make sure the container has enough space to hold `num_bytes` number of random bytes. An example usage is shown below.

```
....
# include <openssl/rand.h>
.....
int getRandBytes(unsigned char * ptr, unsigned num_bytes){
        int ret = RAND_bytes(ptr, num_bytes) ;
```

```
        if(ret != 1) return -1 ;
        return ret ;
}
```

User authentication. For user authentication, you will use the functions you have developed for the Part 1 of the project. For creating the users and their passwords (resp., changing user passwords, deleting user passwords) use the standalone program you have written for the Part 1 of the project. For the current part of the project, when your program starts it will take the password file as an argument and use the functions you have written for Part 1 to check user authentication.

Outputing binary data. For outputing binary data in the file or in the console, please convert the binary data into hexadecimal format first. This is convenient in many ways. For instance, hexadecimal numbers only contains characters from the class "0-9A-F" hence you can use separators such as ':' to separate different fields. For binary data, you would require to pad it to a specific length and do involved length and offset calculations. For outputing the binary data, you can use the following `byte_array_to_hex_array` function.

```
unsigned char to_hex_digit(unsigned int val){
        if(val >= 0 && val <= 9)
                return (unsigned char) (val + '0') ;
        else if(val >= 10 && val < 16)
                return (unsigned char) (val - 10 + 'A') ;
        else{
                assert(0);
                return 255;
        }
}
//REQUIRED: Both byte_array and hex_string are already allocated with enough space
//hex_string size should be twice the size of byte_array
//If byte_array size is 5, then allocate (10+1) bytes for hex_string
// The additional byte is for safety, this is not required.
//RETURNS: the length of the hex_string (-1 on error)
int byte_array_to_hex_array(unsigned char * byte_array, unsigned int num_bytes,  unsigned
     char* hex_string)
{
        if(!byte_array){
                printf("ERROR: Byte String not allocated ....");
                return -1 ;
        }
        if(!hex_string){
                printf("ERROR: Hex String not allocated ....");
                return -1 ;
        }
        int i , j = 0;
        unsigned char buff[1024];
        unsigned char mask = 0xf0;
        for(i=0;i<num_bytes;++i){
                hex_string[2*i]=to_hex_digit((byte_array[i] & mask)>>4);
                hex_string[2*i+1]=to_hex_digit(byte_array[i]&(mask>>4));
        }
        hex_string[2*i] = (unsigned char)0 ;
        return (2*i)  ;
}
```

Adversary model. The adversary we consider for the project is assumed to have the following capabilities. He can access the file system when the SEFS is not operational multiple times. You can view the adversary model as your SEFS and the adversary take turns read and write the same disk. When the adversary has access, he can read from any file or write to any file of the system.

The **confidentiality goal** of the project is to ensure that the adversary does not learn (any part of) the raw content (*i.e.*, plaintext content) of the files. The adversary is allowed to know the lengths of the files, and which parts of a file have been updated.

The **integrity goal** of the project is to ensure that if the adversary modifies something, then your system will be available to detect it. More precisely, suppose that one invokes the method to read from the SEFS content, and the result become different due to the adversary's action, then the method should report error.

Re-encrypting data. Whenever you are reencrypting any data, please make sure you generate a new random initialization vector (IV). **Do not use the IV you have already used**.

Temporary files. You cannot generate any temporary files containing sensitive information in plaintext. Sensitive information here refers to any information that you do not want the adversary to know, *e.g.*, encryption-decryption key, hmac-key, passwords, file contents, file meta information, *etc.*

Simplified SEFS description.

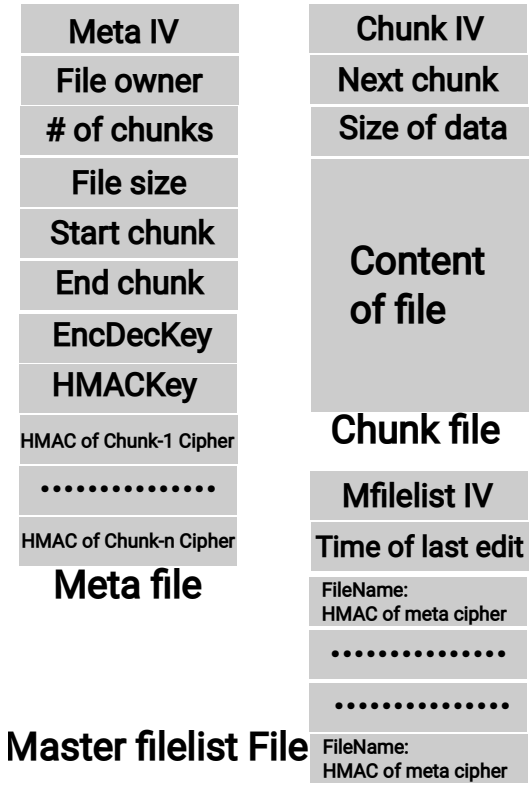

Figure 1: Format of meta file and chunk file

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. For encrypting the content, you should use a random key of 128 bit length for encryption. The initialization vector (IV) (128-bit in size) 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 it will contain the hexadecimal representation of the ciphertext AES_CTR(IV, Key, NULL||18||AnEmailFromMeToYou).

A plaintext representation of the chunk file is shown in Figure 1. An example usage of the `HMAC` function is shown below.

```
|| HMAC(EVP_sha256(), HMAC_key, HMAC_KEY_SIZE_IN_BYTES, data, data_len_in_bytes, NULL, NULL)
```

For taking HMAC of a large data, you can do it incrementally using the following functions. See the following website for details: `https://www.openssl.org/docs/manmaster/crypto/hmac.html`.

- `void HMAC_CTX_init(HMAC_CTX *ctx);`
- `int HMAC_Init(HMAC_CTX *ctx, const void *key, int key_len, const EVP_MD *md);`
- `int HMAC_Update(HMAC_CTX *ctx, const unsigned char *data, int len);`
- `int HMAC_Final(HMAC_CTX *ctx, unsigned char *md, unsigned int *len);`
- `void HMAC_cleanup(HMAC_CTX *ctx);`

Meta file format. The meta file (*e.g.*, **email.meta**) contains auxiliary information regarding the encrypted file (*e.g.*, **random.chunk**). Note that, the meta file will also be encrypted with the master encryption-decryption key. A plaintext format of the meta file is shown in Figure 1. For this part of the project, the meta file will contain 9 lines. Line 1 of the meta file will contain the hexadecimal representation of the IV used to encrypt this file. Line 2 of the meta file will contain the username who owns the file. Line 3 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 4 will contain the size of the plaintext file (*e.g.*, **email**) in bytes. Line 5 will contain the name of the starting chunk of the file. Line 6 will contain the name of the ending chunk of the file. For this part of the project, both line 5 and 6 will contain the same value. For the example above, it will contain the value **random.chunk**. The line 7 & 8 contains the encryption-decryption key used to encrypt the chunk file and the hmac-key which is used to take the hmac of the chunk file (in its encrypted form), respectively. Finally, line 9 will contain the hmac of the chunk file (in its encrypted form) using the hmac-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 meta IV. An example of AES in CTR mode is shown below. The code can be downloaded from `http://www.gurutechnologies.net/blog/aes-ctr-encryption-in-c/` and `http://stackoverflow.com/questions/3141860/aes-ctr-256-encryption-mode-of-operation-on-openssl`. Note that, the following piece of code can only encrypt or decrypt when the file size is a multiple of `AES_BLOCK_SIZE` (*i.e.*, 128 bits). Make sure you take that into consideration.

```
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/buffer.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
struct ctr_state{
        unsigned char ivec[AES_BLOCK_SIZE];
        unsigned int num;
        unsigned char ecount[AES_BLOCK_SIZE];
};
FILE *readFile; FILE *writeFile; AES_KEY key;
int bytes_read, bytes_written;
unsigned char indata[AES_BLOCK_SIZE]; unsigned char outdata[AES_BLOCK_SIZE];
unsigned char iv[AES_BLOCK_SIZE]; struct ctr_state state;
int init_ctr(struct ctr_state *state, const unsigned char iv[16]){
    /* aes_ctr128_encrypt requires 'num' and 'ecount' set to zero on the
     * first call. */
    state->num = 0;
    memset(state->ecount, 0, AES_BLOCK_SIZE);
    /* Initialise counter in 'ivec' to 0 */
    memset(state->ivec + 8, 0, 8);
    /* Copy IV into 'ivec' */
    memcpy(state->ivec, iv, 8);
}
void fencrypt(char* read, char* write, const unsigned char* enc_key){
```

```
    if(!RAND_bytes(iv, AES_BLOCK_SIZE)){
        fprintf(stderr, "Could not create random bytes.");
        exit(1);
    }
    readFile = fopen(read,"rb"); // The b is required in windows.
    writeFile = fopen(write,"wb");
    if(readFile==NULL){
        fprintf(stderr, "Read file is null.");
        exit(1);
    }
    if(writeFile==NULL){
        fprintf(stderr, "Write file is null.");
        exit(1);
    }
    fwrite(iv, 1, 8, writeFile); // IV bytes 1 - 8
    fwrite("\0\0\0\0\0\0\0\0", 1, 8, writeFile); // Fill the last 4 with null bytes 9 -
        16
        //Initializing the encryption KEY
    if (AES_set_encrypt_key(enc_key, 128, &key) < 0){
        fprintf(stderr, "Could not set encryption key.");
        exit(1);
    }
    init_ctr(&state, iv); //Counter call
    //Encrypting Blocks of 16 bytes and writing the output.txt with ciphertext
    while(1){
        bytes_read = fread(indata, 1, AES_BLOCK_SIZE, readFile);
        AES_ctr128_encrypt(indata, outdata, bytes_read, &key, state.ivec, state.ecount, &
            state.num);
        bytes_written = fwrite(outdata, 1, bytes_read, writeFile);
        if (bytes_read < AES_BLOCK_SIZE) break;
    }
    fclose(writeFile);
    fclose(readFile);
}
void fdecrypt(char* read, char* write, const unsigned char* enc_key){
        readFile=fopen(read,"rb"); // The b is required in windows.
        writeFile=fopen(write,"wb");
        if(readFile==NULL){
                fprintf(stderr,"Read file is null.");
                exit(1);
        }
        if(writeFile==NULL){
                fprintf(stderr, "Write file is null.");
                exit(1);
        }
        fread(iv, 1, AES_BLOCK_SIZE, readFile);
        //Initializing the encryption KEY
        if (AES_set_encrypt_key(enc_key, 128, &key) < 0){
                fprintf(stderr, "Could not set decryption key.");
                exit(1);
        }
        init_ctr(&state, iv);//Counter call
        //Encrypting Blocks of 16 bytes and writing the output.txt with ciphertext
        while(1){
                bytes_read = fread(indata, 1, AES_BLOCK_SIZE, readFile);
                AES_ctr128_encrypt(indata, outdata, bytes_read, &key, state.ivec, state.
                    ecount, &state.num);
                bytes_written = fwrite(outdata, 1, bytes_read, writeFile);
                if (bytes_read < AES_BLOCK_SIZE) break;
        }
        fclose(writeFile);
        fclose(readFile);
}
int main(int argc, char *argv[]){
```

```
        fencrypt("encme.txt", "enced.enc", (unsigned const char*)"1234567812345678");
        fdecrypt("enced.enc", "unenced.txt", (unsigned const char*)"1234567812345678");
        printf("FINISHED DOING ENCRYPTION AND DECRYPTION\n");
        return 0;
}
```

Master filelist format. Finally, we also assume that the simplified SEFS has an encrypted file named **master.filelist**. The plaintext version of this file contains multiple lines. The format of the file in shown in Figure 1. This is a variable length file. The first line of the master filelist file will have the hexadecimal representation of the IV used to encrypt the file in plaintext. In the plaintext format, the next line of the file will contain the date and time the file was last edited. Everytime you re-encrypt the master filelist file, you will read the system date and time and encrypt it. When you decrypt the master filelist file you will then check the input date and time with the one in the file. This protects the file to be replaced by an older version of the file. Then the file will contain (possibly) several lines each containing information about a single file that has been encrypted with the simplified SEFS. Each line is of format $x : y$ where ':' is the separator. x represents a file name whereas y is the hexadecimal representation of the hmac of x's encrypted meta file (*i.e.*, hmac of the ciphertext). For performing the hmac of a file's encrypted meta file, you will use the master hmac-key. The whole master filelist file is then encrypted with the master key and the IV in the first line. Note that, even in the encrypted file the IV is kept in plaintext which will be used for decryption.

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 with the same chunk encryption-decryption key but with a different IV. Each chunk can contain a maximum of 1024 bytes of the plaintext file content. The last chunk can possibly have less than 1024 bytes. In which case, you should pad it with the character 0 (ASCII value 0) to make it 1024 bytes.

Extension to the meta file and the chunk file. The chunks of a file can be viewed as a linked list. 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. The last chunk file will contain the value "NULL". Additionally, instead of having only one line containing chunk_hash_mac in the meta file, the meta file can possibly contain multiple lines of 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.*, 9 for the simplified SEFS in part 2). Additionally, in the number of chunk field in the meta file, we can have a value other than 1.

Space efficiency restriction. You are required to use space efficiently. The overall goal is to use the minimum number of chunks for representing a file. After each operation you have to ensure that all chunks except the last one has 1024 bytes of plaintext data (*i.e.*, padding character is not considered part of the plaintext data).

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 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 Implementation

For compilation, I have given you several `Makefile`s. Invoke the `make` command in the terminal after you have accessed the appropriate folder. If the compilation is successful, it will generate a binary file (*e.g.*, `phasher`) in the same folder. To execute the binary `phasher`, you should type the following in the terminal (without quotes): '.\`phasher`'. For compilation, please use "-lcrypto" flag. **Please make sure your code compiles with `gcc`. Codes that do not compile with `gcc`, will not be graded.** You are given an account on the `river.cs.purdue.edu` machine. You should have already received an email from your TA with the username and password to access the machine. The TA has installed the openSSL library. For starters, try to compile one of the codes, you are provided as an example. Please contact the TA, if one of the sample codes do not compile in the `river` machine.

Input-output function. In the skeleton codes, you are provided, uses `scanf` as the function to take input from the console. As you have been taught, `scanf` does not perform bound check, hence it is susceptible to buffer overflow vulnerabilities. Use functions you think are appropriate. The same goes for `sscanf` and `sprintf`.

Changing the skeleton code. Please feel free to make any organizational change in the skeleton file. A skeleton code was given to you to show how the program interface or the console expects input. As long as you follow the same format, it is okay (*i.e.*, same command names, take arguments in the same order, take inputs in the same order, *etc.*). You are not bound use the same organization as the skeleton code as long as it operates in the same fashion.

7 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 16 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.

8 Potential Pitfalls

Memory leaks. A lot of the operations of the project require pointer manipulation, make sure to free the pointer after usage. Please be careful to not double free a pointer, not access a pointer with allocating, not access a pointer after deallocating, *etc.*

File operations. File operations in C are complicated, you cannot write in the middle of a file without overwriting the content. You have to manually move the following content and then write something. Otherwise, you can end up overwriting the content. Note that, while opening a file using `fopen` function, if you open it with the "wb" mode, then it will overwrite an existing file. Please make sure that you use `fopen` with appropriate access mode.

Error checking. A lot of errors can potentially happen during the operation and it is paramount that you do handle these errors gracefully. Make sure your program does not crash because of a wrong input. Recall that an adversary can write to your file, hence it might not follow the format explained. You program should not crash and rather state that the operation failed because of wrong format. Do not assume inputs are well-formed. Perform input validation when applicable. A lot of my test inputs are basically malformed files. **Specifically for integrity checking, you have to make sure that if the decryption fails then you raise an error flag. Hence it is essential that you have a way to detect decryption failure.**