TCFS

0.2

Generated on Mon Nov 27 2023 17:03:19 for TCFS by Doxygen 1.9.8

Mon Nov 27 2023 17:03:19

Chapter 1

TCFS - Transparent Cryptographic Filesystem

TCFS is a transparent cryptographic filesystem designed to secure files mounted on a Network File System (NFS) server. It is implemented as a FUSE (Filesystem in Userspace) module along with a user-friendly helper program. TCFS ensures that files are encrypted and decrypted seamlessly without requiring user intervention, providing an additional layer of security for sensitive data.

1.1 Disclamer

Note: This project is currently in an early development stage and should be considered as an alpha version. This means there may be many missing features, unresolved bugs, or unexpected behaviors. The project is made available in this phase for testing and evaluation purposes and should not be used in production or for critical purposes. It is not recommended to use this software in sensitive environments or to store important data until a stable and complete version is reached. We appreciate any feedback, bug reports, or contributions from the community that can help improve the project. If you decide to use this software, please **don't do it**. Thank you for your interest and understanding as we work to improve the project and make it stable and complete.

1.2 Features

- Transparent Encryption: TCFS operates silently in the background, encrypting and decrypting files on-the-fly as they are accessed or modified. Users don't need to worry about managing encryption keys or performing manual cryptographic operations.
- FUSE Integration: TCFS leverages the FUSE framework to create a virtual filesystem that integrates seamlessly with the existing file hierarchy. This allows users to interact with their files just like any other files on their system.
- Secure Data Storage: Files stored on an NFS server can be vulnerable during transit or at rest. TCFS addresses these security concerns by ensuring data is encrypted before it leaves the client system, offering end-to-end encryption for your files.
- Transparency: No modifications to the NFS server are required.

1.3 Getting Started

1.3.1 Prerequisites

- FUSE: Ensure that FUSE and FUSE-dev are installed on your system. You can usually install it using your system's package manager (e.g., apt, yum, dnf, ecc).
- OpenSSl: Install OpenSSL and its development package.

1.3.2 Build

- Clone the TCFS repository to your local machine:

  ```
  git clone  https://github.com/carloalbertogiordano/TCFS
  ```

- Compile: Run the Makefile in the userspace-module directory (Only the FUSE module is avilable at the moment, the whole project has not been implemented yet)

  ```
  make all
  ```

#

1.4 Usage

1.4.0.1 Mount an NFS share using TCFS:

First, mount the NFS share to a directroy, this directory will be called sourcedir. This will be done by the helper program in a future release.

```
    ./build-fs/tcfs-fuse-module/tcfs -s /fullpath/sourcedir -d /fullpath/destdir -p "your passwor
```

Access and modify files in the mounted directory as you normally would. TCFS will handle encryption and decryption automatically. NOTE: This behaviour will be changed in the future, the kernel module will handle your password.

1.4.0.2 Unmount the NFS share when you're done:

```
    fusermount -u /fullpath/destdir
```

then unmount the NFS share.

1.4.1 Contributing

Contributions to TCFS are welcome! If you find a bug or have an idea for an improvement, please open an issue or submit a pull request on the TCFS GitHub repository.

1.4.2 License

This project is licensed under the GPLv3 License - see the LICENSE file for details.

1.4.3 Acknowledgments

TCFS is inspired by the need for secure data storage and transmission in NFS environments. Thanks to the FUSE project for providing a user-friendly way to create custom filesystems.

Inspiration from TCFS (2001): This project draws substantial inspiration from an earlier project named "TCFS" that was developed around 2001. While the original source code for TCFS has unfortunately been lost over time, we have retained valuable documentation and insights from that era. In the "TCFS-2001" folder, you can find historical documentation and design concepts related to the original TCFS project. Although we are unable to directly leverage the source code from the previous project, we have taken lessons learned from its design principles to inform the development of this current TCFS implementation. We would like to express our gratitude to the creators and contributors of TCFS for their pioneering work, which has influenced and inspired our efforts to create a modern TCFS solution. Thank you for your interest in this project as we continue to build upon the foundations set by the original TCFS project.

1.4.4 Roadmap

- Key management:
  - ~~Store a per-file key in the extended attributes and use the user key to decipher it.~~
  - Implement a kernel module to rebuild the private key to decipher the files. This module will use a certificate and your key to rebuild the private key
  - Implement key recovery.
- Implement threshold sharing files.

Chapter 2

Todo List

Member handle_termination (int signum)

: Implement remove_queue() to clear and delete the queue

Member main ()

: The brief description is basically false advertisement. It only spawn a thread and hangs infinitely

: Remove the thread that spawns handle_outgoing_messages. This must not make it into final release

File tcfs_daemon.c

: Enable forking

Run the daemon via SystemD

Member terminate

: Implement logic to make this work

Member terminate_mutex

: implement logic to make this work

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

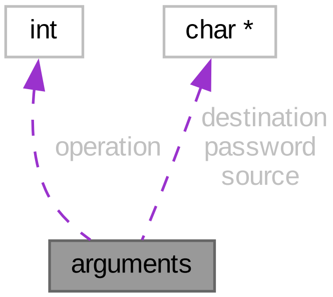
Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 arguments Struct Reference

Collaboration diagram for arguments:



Public Attributes

- int operation
- char ∗ source
- char ∗ destination
- char ∗ password

5.1.1 Detailed Description

Definition at line 20 of file user_tcfs.c.

5.1.2 Member Data Documentation

5.1.2.1 destination

```
char* arguments::destination
```

Definition at line 668 of file tcfs.c.

5.1.2.2 operation

```
int arguments::operation
```

Definition at line 21 of file user_tcfs.c.

5.1.2.3 password

```
char* arguments::password
```

Definition at line 669 of file tcfs.c.

5.1.2.4 source

```
char* arguments::source
```

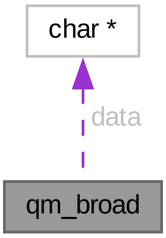
Definition at line 667 of file tcfs.c.

The documentation for this struct was generated from the following files:

- user/user_tcfs.c
- userspace-module/tcfs.c

5.2 qm_broad Struct Reference

Collaboration diagram for qm_broad:



Public Attributes

- char ∗ data

5.2.1 Detailed Description

Definition at line 40 of file common.h.

5.2.2 Member Data Documentation

5.2.2.1 data

```
char∗ qm_broad::data
```

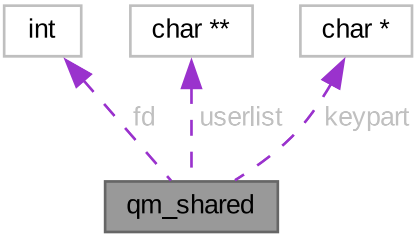
Definition at line 41 of file common.h.

The documentation for this struct was generated from the following file:

- daemon/daemon_utils/common.h

5.3 qm_shared Struct Reference

Collaboration diagram for qm_shared:



Public Attributes

- int fd
- char ∗∗ userlist
- char ∗ keypart

5.3.1 Detailed Description

Definition at line 34 of file common.h.

5.3.2 Member Data Documentation

5.3.2.1 fd

```
int qm_shared::fd
```

Definition at line 35 of file common.h.

5.3.2.2 keypart

```
char* qm_shared::keypart
```

Definition at line 37 of file common.h.

5.3.2.3 userlist

```
char** qm_shared::userlist
```

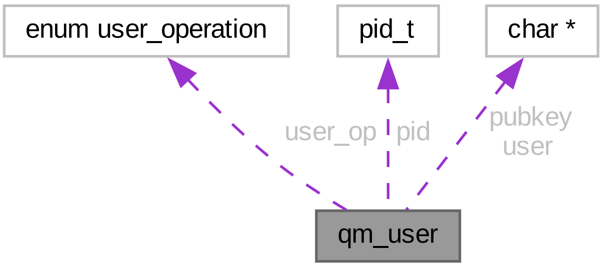
Definition at line 36 of file common.h.

The documentation for this struct was generated from the following file:

- daemon/daemon_utils/common.h

5.4 qm_user Struct Reference

Collaboration diagram for qm_user:



Public Attributes

- user_operation user_op
- pid_t pid
- char ∗ user
- char ∗ pubkey

5.4.1 Detailed Description

Definition at line 27 of file common.h.

5.4.2 Member Data Documentation

5.4.2.1 pid

```
pid_t qm_user::pid
```

Definition at line 29 of file common.h.

5.4.2.2 pubkey

```
char* qm_user::pubkey
```

Definition at line 31 of file common.h.

5.4.2.3 user

```
char* qm_user::user
```

Definition at line 30 of file common.h.

5.4.2.4 user_op

```
user_operation qm_user::user_op
```

Definition at line 28 of file common.h.

The documentation for this struct was generated from the following file:

- daemon/daemon_utils/common.h

Chapter 6

File Documentation

6.1 common.h

```c
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <mqueue.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define MAX_QM_SIZE 512 //Max size of a message
#define MAX_QM_N 100 //Max number of messages that can be enqueued

#ifndef QUEUE_STRUCTS
#define QUEUE_STRUCTS

typedef enum qm_type{
    USER = 0,
    SHARED = 1,
    BROADCAST = 2,
    QM_TYPE_UNDEFINED = -1,
} qm_type;

typedef enum user_operation{
    REGISTER = 0,
    UNREGISTER = 1,
} user_operation;

typedef struct qm_user {
    user_operation user_op;
    pid_t pid;
    char *user;
    char *pubkey;
} qm_user;

typedef struct qm_shared {
    int fd;
    char **userlist;
    char *keypart;
} qm_shared;

typedef struct qm_broad {
    char *data;
} qm_broad;

#endif
```

6.2 redis.c

```c

#include "redis.h"
#include "../print/print_utils.h"
#include "../json/json_tools.h"
#include <hiredis/hiredis.h>

const char HOST[] = "127.0.0.1";
```

```
#define PORT 6380

redisContext *context;

void print_all_keys() {
    redisReply *keys_reply = (redisReply *)redisCommand(context, "KEYS *");
    if (keys_reply) {
        if (keys_reply->type == REDIS_REPLY_ARRAY) {
            for (size_t i = 0; i < keys_reply->elements; ++i) {
                print_msg("\tKey: %s", keys_reply->element[i]->str);
            }
        } else {
            print_msg("Error retrieving keys: %s", keys_reply->str);
        }
        freeReplyObject(keys_reply);
    } else {
        print_msg("Error executing KEYS command");
    }
}
int init_context()
{
    //Do not reinit the context
    if (context != NULL)
        return 1;

    context = redisConnect(HOST, PORT);
    if (context->err) {
        print_err("Connection error: %s", context->errstr);
        return 0;
    }
    return 1;
}
void free_context()
{
    redisFree(context);
}
qm_user *json_to_qm_user(char *json)
{
    print_msg("DEBUG: Converting %s", json);
    qm_type type;
    //Redis return the value as json:{actual json} so we need to eliminate the json: from the string
    char *res = strchr(json, ':');
    res++; //Skip the : char
    qm_user *user = (qm_user *)string_to_struct(res, &type);
    return user;
}
qm_user *get_user_by_pid(pid_t pid) {
    qm_user *user = NULL;
    // Retrieve the JSON data from Redis hash
    print_msg("EXECUTING \"GET pid:%d\"", pid);
    redisReply *luaReply = (redisReply *)redisCommand(context, "GET pid:%d", pid);
    if (luaReply) {
        if (luaReply->type == REDIS_REPLY_STRING) {
            user = json_to_qm_user(luaReply->str);
            if (user) {
                print_msg("Successful retrieval! PID: %d, User: %s", user->pid, user->user);
            } else {
                print_err("Error converting JSON to struct");
            }
        } else {
            print_err("Reply type error %d -> executing HGET\n\tErrString: %s",
                      luaReply->type, luaReply->str,context->errstr);
        }
        freeReplyObject(luaReply);
    } else {
        print_err("Reply error executing HGET\n\tErrString: %s", context->errstr);
    }
    return user;
}
qm_user *get_user_by_name(const char *name) {
    qm_user *user = NULL;
    // Retrieve the JSON data from Redis hash
    print_msg("EXECUTING \"GET name:%d\"", name);
    redisReply *luaReply = (redisReply *)redisCommand(context, "GET name:%d", name);
    if (luaReply) {
        if (luaReply->type == REDIS_REPLY_STRING) {
            user = json_to_qm_user(luaReply->str);
            if (user) {
                print_msg("Successful retrieval! PID: %d, User: %s", user->pid, user->user);
            } else {
                print_err("Error converting JSON to struct");
            }
        } else {
            print_err("Reply type error %d -> executing HGET\n\tErrString: %s",
                      luaReply->type, luaReply->str,context->errstr);
        }
        freeReplyObject(luaReply);
```

```c
    } else {
        print_err("Reply error executing HGET\n\tErrString: %s", context->errstr);
    }
    return user;
}
int insert(qm_user *user)
{
    // Convert the structure to JSON
    const char *json = struct_to_json(USER, user);
    if (!json)
    {
        print_err("Error converting qm_user to JSON");
        return 0;
    }
    // Save to Redis with key "pid_str"
    print_msg("\tDB: \"SET pid:%d json:%s\"", user->pid, json);
    redisReply *reply_pid =(redisReply *) redisCommand(context, "SET pid:%d json:%s", user->pid, json);
    if (!reply_pid)
    {
        print_err("Error saving to Redis (pid)");
        free((void *)json);
        return 0;
    }
    freeReplyObject(reply_pid);

    // Save to Redis with key "user"
    redisReply *reply_user =(redisReply *) redisCommand(context, "SET user:%s json:%s", user->user, json);
    if (!reply_user)
    {
        print_err("Error saving to Redis (user)");
        free((void *)json);
        return 0;
    }
    freeReplyObject(reply_user);
    // Free the allocated JSON memory
    free((void *)json); //Discard qualifier
    return 1;
}
int remove_by_pid(pid_t pid)
{
    qm_user *user_tmp = get_user_by_pid(pid);
    // Remove the structure by PID
    print_msg("\tDB: \"DEL pid:%d\"", pid);
    redisReply *reply_pid =(redisReply *) redisCommand(context, "DEL pid:%d", pid);
    if (!reply_pid) {
        print_err("Error removing structure by PID");
        return 0;
    }
    freeReplyObject(reply_pid);
    // Also remove the corresponding key by name
    print_msg("\tDB: \"DEL user:%s\"", user_tmp->user);
    redisReply *reply_name =(redisReply *) redisCommand(context, "DEL user:%s", user_tmp->user);
    if (!reply_name) {
        print_err("Error removing key by name");
        return 0;
    }
    free(user_tmp);
    freeReplyObject(reply_name);
    return 1;
}
int remove_by_user(char *name)
{
    qm_user *user_tmp = get_user_by_name(name);
    // Remove the structure by name
    char key_name[64]; // Adjust the size as needed
    snprintf(key_name, sizeof(key_name), "user:%s", name);
    redisReply *reply_name =(redisReply *) redisCommand(context, "DEL %s", key_name);
    if (!reply_name) {
        print_err("Error removing structure by name");
        return 0;
    }
    freeReplyObject(reply_name);
    // Also remove the corresponding key by PID
    redisReply *reply_pid =(redisReply *) redisCommand(context, "DEL %d", user_tmp->pid);
    if (!reply_pid) {
        print_err("Error removing key by PID");
        return 0;
    }
    freeReplyObject(reply_pid);
    return 1;
}
```

6.3 redis.h

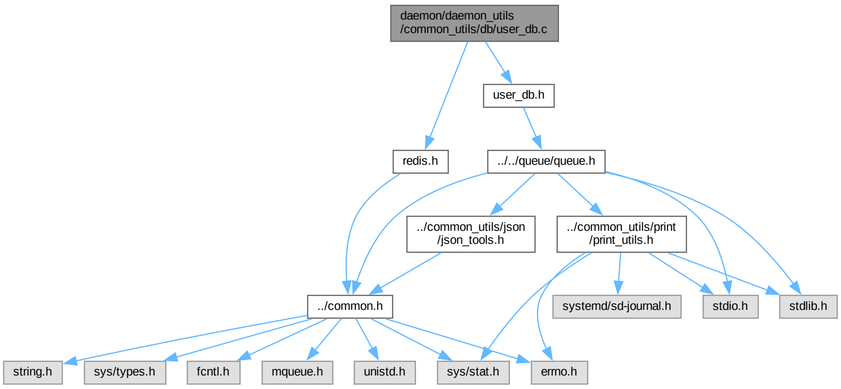
```
#include "../../common.h"

void print_all_keys();

int init_context();

qm_user *json_to_qm_user(char *json);

qm_user *get_user_by_pid(pid_t pid);

qm_user *get_user_by_name(const char *name);

int insert(qm_user *user);

int remove_by_pid(pid_t pid);

int remove_by_user(char *name);

void free_context();
```

6.4 daemon/daemon_utils/common_utils/db/user_db.c File Reference

This file contains the functions to interact with the database.

```
#include "user_db.h"
#include "redis.h"
```

Include dependency graph for user_db.c:



Functions

- int register_user (qm_user *user_msg)

  Register or update a user in the db, this relies on the redis.c file.
- int unregister_user (pid_t pid)

  Remove a user from the DB.
- void disconnect_db (void)

  Free the context of the DB.

6.4.1 Detailed Description

This file contains the functions to interact with the database.

Definition in file user_db.c.

6.4.2 Function Documentation

6.4.2.1 disconnect_db()

```
void disconnect_db (
            void  )
```

Free the context of the DB.

Parameters

void	

Returns

void

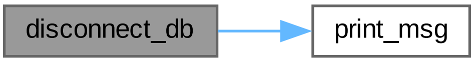
Note

If this fails no errors will be printed and no errno will be set, you are on your own :(

Definition at line 41 of file user_db.c.

References print_msg().

Here is the call graph for this function:



6.4.2.2 register_user()

```
int register_user (
            qm_user * user_msg )
```

Register or update a user in the db, this relies on the redis.c file.

Parameters

qm_user∗	A pointer to the allocated qm_user∗ struct

Returns

1 if successful, 0 otherwise

Definition at line 14 of file user_db.c.

References print_msg().

Here is the call graph for this function:

register_user
print_msg

6.4.2.3 unregister_user()

```
int unregister_user (
            pid_t pid )
```

Remove a user from the DB.

Parameters

pid↩_t	pid the key

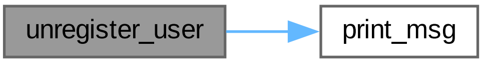
Returns

1 if successful, 0 otherwise

Definition at line 29 of file user_db.c.

References print_msg().

Here is the call graph for this function:



6.5 user_db.c

Go to the documentation of this file.

```
#include "user_db.h"
#include "redis.h"

int register_user(qm_user *user_msg)
{
    print_msg("Registering new user");
    if (init_context() == 0)
        return 0;
    print_all_keys();
    if (insert(user_msg) == 0)
        return 0;
    return 1;
}
int unregister_user(pid_t pid)
{
    print_all_keys();
    print_msg("Removing user");
    return remove_by_pid(pid);
}
void disconnect_db(void)
{
    print_msg("Freeing context...");
    free_context();
}
```

6.6 user_db.h

```
#include "../../queue/queue.h"

int register_user(qm_user *user_msg);
int unregister_user(pid_t pid);
void disconnect_db(void);
```

6.7 json_tools.cpp

```
#include "../../common.h"
#include <iostream>
#include <string.h>
#include <vector>
#include <cstring> // For strcpy
#include <cstdlib> // For malloc and free
#include "/usr/include/nlohmann/json.hpp" // Assuming you're using nlohmann's JSON library
#include "../print/print_utils.h"

char* struct_to_json(qm_type qmt, void* q_mess) {
    nlohmann::json json_obj;

    switch (qmt) {
        case USER: {
            qm_user* user = static_cast<qm_user*>(q_mess);
            if (user->user_op == REGISTER)
                print_msg("Register");
            if (user->user_op == UNREGISTER)
                print_msg("Unregister");
            json_obj["user_op"] = user->user_op;
            json_obj["pid"] = user->pid;
            json_obj["user"] = user->user;
            json_obj["pubkey"] = user->pubkey;
            break;
        }
        case SHARED: {
            qm_shared* shared = static_cast<qm_shared*>(q_mess);
            json_obj["fd"] = shared->fd;

            // Converti la matrice di stringhe in un array di stringhe JSON
            nlohmann::json userlist_array = nlohmann::json::array();
            for (size_t i = 0; shared->userlist[i] != nullptr; ++i) {
                userlist_array.push_back(shared->userlist[i]);
            }
            json_obj["userlist"] = userlist_array;

            json_obj["keypart"] = shared->keypart;
            break;
        }
        case BROADCAST: {
            qm_broad* broad = static_cast<qm_broad*>(q_mess);
            json_obj["data"] = broad->data;
            break;
        }
    }
    // Cast Json obj to string
    std::string json_str = json_obj.dump();
    // Allocate memory for result
    char* result = (char*)malloc(json_str.size() + 1);
    if (result) {
        strcpy(result, json_str.c_str());
    }
    print_msg("JSONIFIED: %s", result);
    return result;
}

void* string_to_struct(const char* json_string, qm_type* type) {
    try {
        nlohmann::json json_obj = nlohmann::json::parse(json_string);

        if (json_obj.contains("user_op")) {
            *type = USER;
            qm_user* user = static_cast<qm_user*>(std::malloc(sizeof(qm_user)));
            user->user_op = json_obj["user_op"];
            user->pid = json_obj["pid"];
            user->user = strdup(json_obj["user"].get<std::string>().c_str());
            user->pubkey = strdup(json_obj["pubkey"].get<std::string>().c_str());
            return user;
        } else if (json_obj.contains("fd")) {
            *type = SHARED;
            qm_shared* shared = static_cast<qm_shared*>(std::malloc(sizeof(qm_shared)));
            shared->fd = json_obj["fd"];

            // Populate userlist array
            std::vector<std::string> userlist = json_obj["userlist"];
            shared->userlist = static_cast<char**>(std::malloc((userlist.size() + 1) * sizeof(char*)));
            for (size_t i = 0; i < userlist.size(); ++i) {
                shared->userlist[i] = strdup(userlist[i].c_str());
            }
            shared->userlist[userlist.size()] = nullptr;

            shared->keypart = strdup(json_obj["keypart"].get<std::string>().c_str());
            return shared;
        } else if (json_obj.contains("data")) {
            *type = BROADCAST;
            qm_broad* broad = static_cast<qm_broad*>(std::malloc(sizeof(qm_broad)));
            broad->data = strdup(json_obj["data"].get<std::string>().c_str());
            return broad;
        } else {
            *type = QM_TYPE_UNDEFINED;
            return nullptr;
        }
    } catch (const std::exception& e) {
        std::cerr « "Error parsing JSON: " « e.what() « std::endl;
        return nullptr;
    }
}
```

6.8 json_tools.h

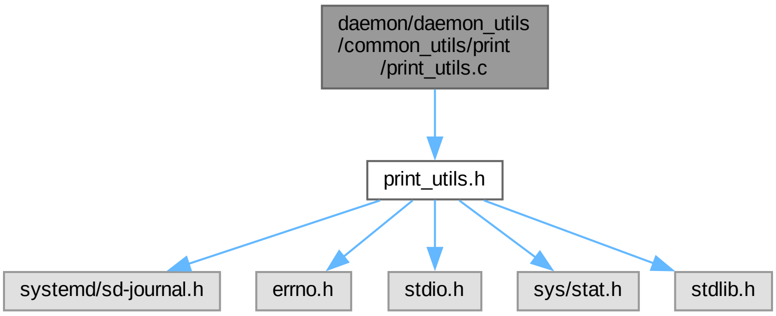
```
#include "../../common.h"

extern const char *struct_to_json(qm_type qmt, void *q_mess);
extern void* string_to_struct(const char* json_string, qm_type* type);
```

6.9 daemon/daemon_utils/common_utils/print/print_utils.c File Reference

This file defines some QoL functions.

```
#include "print_utils.h"
```

Include dependency graph for print_utils.c:



Functions

- void print_err (const char ∗format,...)

  Format and print data as an error.
- void print_msg (const char ∗format,...)

  Format and print data as a message.
- void print_warn (const char ∗format,...)

  Format and print data as a waring.
- void print_debug (const char ∗format,...)

  Format and print data as a debug.

Variables

- int cleared = 0

6.9.1 Detailed Description

This file defines some QoL functions.

Definition in file print_utils.c.

6.9.2 Function Documentation

6.9.2.1 print_debug()

```
void print_debug (
            const char ∗ format,
             ... )
```

Format and print data as a debug.

Parameters

const	char ∗format the string that will formatted and printed
[ARGUMENTS]...	Print optional ARGUMENT(s) according to format

Returns

void

Note

Will also log using systemD

"DEBUG=" will be prepended to format

Definition at line 131 of file print_utils.c.

6.9.2.2 print_err()

```
void print_err (
             const char * format,
              ... )
```

Format and print data as an error.

Parameters

const	char ∗format the string that will formatted and printed
[ARGUMENTS]...	Print optional ARGUMENT(s) according to format

Returns

void

Note

Will also log using systemD

"ERROR=" will be prepended to format

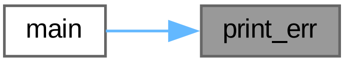
"Err_Numebr:d" will be appended to the formatted string describing the error number

after Err_Number "-> s" will be appended printing the std-error

Definition at line 69 of file print_utils.c.

Referenced by main().

Here is the caller graph for this function:



6.9.2.3 print_msg()

```
void print_msg (
             const char * format,
              ... )
```

Format and print data as a message.

Parameters

const	char ∗format the string that will formatted and printed
[ARGUMENTS]...	Print optional ARGUMENT(s) according to format

Returns

void

Note

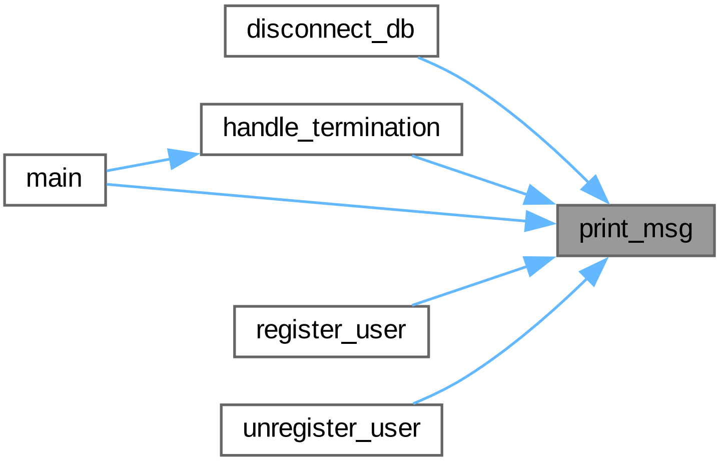
Will also log using systemD

"MESSAGE=" will be prepended to format

Definition at line 89 of file print_utils.c.

Referenced by disconnect_db(), handle_termination(), main(), register_user(), and unregister_user().

Here is the caller graph for this function:



6.9.2.4 print_warn()

```
void print_warn (
             const char * format,
              ...  )
```

Format and print data as a waring.

Parameters

const	char ∗format the string that will formatted and printed
[ARGUMENTS]...	Print optional ARGUMENT(s) according to format

Returns

void

Note

Will also log using systemD

"WARNING=" will be prepended to format

Definition at line 110 of file print_utils.c.

6.9.3 Variable Documentation

6.9.3.1 cleared

```
int cleared = 0
```

Definition at line 13 of file print_utils.c.

6.10 print_utils.c

Go to the documentation of this file.

```c
#include "print_utils.h"

int cleared = 0;

void log_message(const char *log){
    printf("%s\n", log);
    // Path of the log folder and log file
    const char *logFolder = "/var/log/tcfs";
    const char *logFile = "/var/log/tcfs/log.txt";

    // Check if the folder exists, otherwise create it
    struct stat st;
    if (stat(logFolder, &st) == -1) {
        mkdir(logFolder, 0700);
    }

    FILE *file;
    if (cleared == 0)
    {
        cleared = 1;
        file = fopen(logFile, "w");
    } else {
        file = fopen(logFile, "a");
    }

    // Open the log file in append mode
    if (file == NULL) {
        perror("Error opening the log file");
    }

    // Write the message to the log file
    fprintf(file, "%s\n", log);

    // Close the file
    fclose(file);
}

void print_err(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    char buffer[1024];
    vsnprintf(buffer, sizeof(buffer), format, args);
    va_end(args);

    log_message(buffer);

    sd_journal_print(LOG_ERR, "ERROR=%s Err_Number:%d -> %s", buffer, errno, strerror(errno));
}
void print_msg(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    char buffer[1024];
    vsnprintf(buffer, sizeof(buffer), format, args);
    va_end(args);

    log_message(buffer);

    sd_journal_send("MESSAGE=%s", buffer, NULL);
}

void print_warn(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    char buffer[1024];
    vsnprintf(buffer, sizeof(buffer), format, args);
    va_end(args);

    log_message(buffer);

    sd_journal_print(LOG_WARNING, "WARNING=%s", buffer, NULL);
}

void print_debug(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    char buffer[1024];
    vsnprintf(buffer, sizeof(buffer), format, args);
    va_end(args);

    log_message(buffer);

    sd_journal_print(LOG_DEBUG, "DEBUG=%s", buffer, NULL);
}
```

6.11 print_utils.h

```
#include <systemd/sd-journal.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>

void print_err(const char *format, ...);
void print_msg(const char *format, ...);
void print_warn(const char *format, ...);
void print_debug(const char *format, ...);
```

6.12 tcfs_daemon_tools.c

```
#include "tcfs_daemon_tools.h"
#include "../message_handler/message_handler.h"

void *handle_incoming_messages(void *queue_id)
{
    qm_type qmt;
    qm_user *user_msg;
    qm_shared *shared_msg;
    qm_broad *broadcast_msg;


    print_msg("Starting handler for incoming messages");
    void *tmp_struct;
    while (1) {
        tmp_struct = dequeue(*(mqd_t *) queue_id, &qmt);
        switch (qmt) {
            case USER:
                print_msg("Handling user message");
                user_msg = (qm_user *) tmp_struct;
                handle_user_message(user_msg);
                break;
            case SHARED:
                print_msg("Handling shared message");
                shared_msg = (qm_shared *) tmp_struct;
                //handle_shared_message()
                break;
            case BROADCAST:
                print_msg("Handling broadcast message");
                broadcast_msg = (qm_broad *) tmp_struct;
                //handle_broadcast_message()
                break;
            case QM_TYPE_UNDEFINED:
                print_err("Received un unknown message type, skipping...");
                break;
        }
        free(tmp_struct);
    }
    return NULL;
}

void *handle_outgoing_messages(void *queue_id)
{
    print_msg("Handling outgoing messages");
    //sleep(1);

    char s1[] = "TEST";
    char s2[] = "pubkey";

    struct qm_user test_msg;
    test_msg.user_op = REGISTER;
    test_msg.pid = 104;
    test_msg.user = s1;
    test_msg.pubkey = s2;

    print_msg("Enqueueing test registration...");
    int res = enqueue(*(mqd_t *)queue_id, USER, (void *)&test_msg);
    print_msg("TEST message send with result %d", res);

    if (res != 1){
        print_err("enqueue err ");
    }

    struct qm_user test_msg2;
    test_msg2.user_op = UNREGISTER;
    test_msg2.pid = 104;
    test_msg2.user = "";
    test_msg2.pubkey = "";

    sleep(3);

    print_msg("Enqueueing test remove...");
    res = enqueue(*(mqd_t *)queue_id, USER, (void *)&test_msg2);
    print_msg("TEST message send with result %d", res);

    if (res != 1){
        print_err("enqueue err ");
    }

    return NULL;
}

/*
 *
void* monitor_termination(void* queue_id) {
    while (1) {
        pthread_mutex_lock(&terminate_mutex);
        if (terminate) {
            pthread_mutex_unlock(&terminate_mutex);
            break;
        }
        pthread_mutex_unlock(&terminate_mutex);
        sleep(1);
    }
    print_err("Terminating threads");
    remove_empty_queue(*(int *)queue_id);
    return NULL;
}*/
```

6.13 tcfs_daemon_tools.h

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdbool.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/stat.h>
#include <pthread.h>
#include <signal.h>
#include "../queue/queue.h"
#include "../message_handler/message_handler.h"

// Condition variable & mutex
extern volatile int terminate;
extern pthread_mutex_t terminate_mutex;

void *handle_incoming_messages(void *queue_id);
void *handle_outgoing_messages(void *queue_id);
void *monitor_termination(void *queue_id);
void cleanup_threads(pthread_t thread1, pthread_t thread2);
```

6.14 message_handler.c

```
#include "message_handler.h"
#include "../common_utils/json/json_tools.h"
#include "../common_utils/db/user_db.h"
#include "../common_utils/print/print_utils.h"

int handle_user_message(qm_user *user_msg)
{
    if (user_msg->user_op == REGISTER){
        register_user(user_msg);
    } else if (user_msg->user_op == UNREGISTER)
    {
        unregister_user(user_msg->pid);
        //TODO: next line is a test, remove it
        free_context();
    } else
    {
        print_err("Unknown user operation %d", user_msg->user_op);
        return 0;
    }

    return 1;
}
```

6.15 message_handler.h

```
#include "../common.h"
#include "../common_utils/print/print_utils.h"

int handle_user_message(qm_user *user_msg);
```

6.16 queue.c

```
#include "queue.h"

mqd_t init_queue(char *queue)
{
    struct mq_attr attr;
    mqd_t mq;

    // Initialize queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = MAX_QM_N; // Maximum number of messages in the queue
    attr.mq_msgsize = MAX_QM_SIZE; // Maximum size of a single message
    attr.mq_curmsgs = 0;

    // Create the message queue
    mq = mq_open(queue, O_CREAT | O_RDWR /*| O_RDONLY | O_NONBLOCK*/, 0777, &attr); //TODO: Better define permissions
    printf("mqopen %d\n", mq);
    if (mq == (mqd_t)-1) {
        print_err("mq_open cannot create que in %s %d %s", queue, errno, strerror(errno));
        print_msg("mq_open cannot create que in %s %d %s", queue, errno, strerror(errno));
        return 0;
    }
    printf("Message queue created successfully at %s!\n", queue);
    return mq;
}
int enqueue(mqd_t queue_d, qm_type qmt, void *q_mess)
{
    const char *qm_json = struct_to_json(qmt, q_mess);

    if (mq_send(queue_d, qm_json, strlen(qm_json)+1, 0) == -1)
    {
        print_err("mq_send %s", qm_json);
        free((void *)qm_json);
        return 0;
    }
    print_msg("Message sent successfully!\n");
    free((void *)qm_json);
    return 1;
}
void *dequeue(mqd_t queue_d, qm_type *qmt)
{
    char *qm_json = (char *)malloc(sizeof(char ) * MAX_QM_SIZE);

    if (mq_receive(queue_d, qm_json, MAX_QM_SIZE, 0) == -1)
    {
        free((void *)qm_json);
        print_err("mq_rec %d %s", errno, strerror(errno));
        return NULL;
    }

    print_msg("Dequeued %s", qm_json);
    void *tmp_struct = string_to_struct(qm_json, qmt);

    free((void *)qm_json);
    return tmp_struct;
}
```

6.17 queue.h

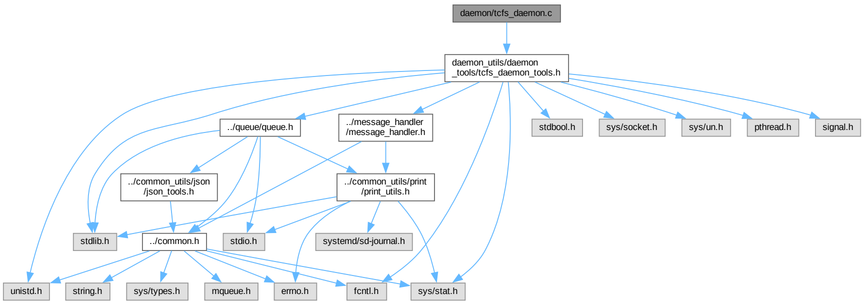
```
#include "../common.h"
#include "../common_utils/print/print_utils.h"
#include "../common_utils/json/json_tools.h"
#include <stdio.h>
#include <stdlib.h>

#define MESSAGE_BUFFER_SIZE 256
#define MQUEUE_N 3;



mqd_t init_queue(char *queue);
int enqueue(mqd_t queue_d, qm_type qmt, void *q_mess);
void *dequeue(mqd_t queue_d, qm_type *qmt);
```

6.18 daemon/tcfs_daemon.c File Reference

This is the core of the daemon.

`#include "daemon_utils/daemon_tools/tcfs_daemon_tools.h"`

Include dependency graph for tcfs_daemon.c:



Functions

- void handle_termination (int signum)

    Handle the termination if SIGTERM is received.
- int main ()

    main function of the daemon. This will daemonize the program, spawn a thread to handle messages and handle unexpected termination of the thread

Variables

- volatile int terminate = 0

    If the spawned threads terminate abruptly they should set this to 1, so that the daemon can terminate.
- pthread_mutex_t terminate_mutex = PTHREAD_MUTEX_INITIALIZER

    Mutex needed to set the var terminate to 1 safely.
- const char MQUEUE [ ] = "/tcfs_queue"

    the queue file location

6.18.1 Detailed Description

This is the core of the daemon.

Note

Forking is disable at the moment, this meas it will run as a "normal" program

the main function spawns a thread to handle incoming messages on the queue

Todo : Enable forking

Run the daemon via SystemD

Definition in file tcfs_daemon.c.

6.18.2 Function Documentation

6.18.2.1 handle_termination()

```
void handle_termination (
             int signum )
```

Handle the termination if SIGTERM is received.

Parameters

int	signum Integer corresponding to SIGNUM

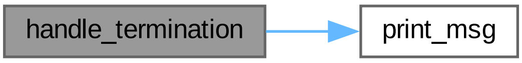
Todo : Implement remove_queue() to clear and delete the queue

Definition at line 36 of file tcfs_daemon.c.

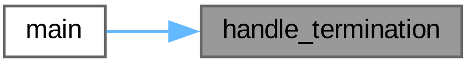
References print_msg().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



6.18.2.2 main()

```
int main ( )
```

main function of the daemon. This will daemonize the program, spawn a thread to handle messages and handle unexpected termination of the thread

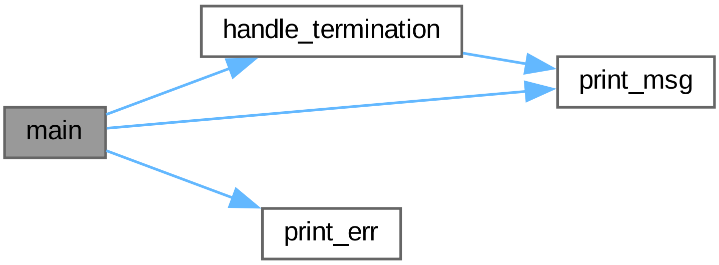
Todo : The brief description is basically false advertisement. It only spawn a thread and hangs infinitely

: Remove the thread that spawns handle_outgoing_messages. This must not make it into final release

Definition at line 47 of file tcfs_daemon.c.

References handle_termination(), MQUEUE, print_err(), print_msg(), and terminate.

Here is the call graph for this function:



6.18.3 Variable Documentation

6.18.3.1 MQUEUE

```
MQUEUE = "/tcfs_queue"
```

the queue file location

Definition at line 29 of file tcfs_daemon.c.

Referenced by main().

6.18.3.2 terminate

```
volatile int terminate = 0
```

If the spawned threads terminate abruptly they should set this to 1, so that the daemon can terminate.

Todo : Implement logic to make this work

Definition at line 17 of file tcfs_daemon.c.

Referenced by main().

6.18.3.3 terminate_mutex

```
pthread_mutex_t terminate_mutex = PTHREAD_MUTEX_INITIALIZER
```

Mutex needed to set the var terminate to 1 safely.

Todo : implement logic to make this work

Definition at line 23 of file tcfs_daemon.c.

6.19 tcfs_daemon.c

Go to the documentation of this file.

```
00001 #include "daemon_utils/daemon_tools/tcfs_daemon_tools.h"
00002
00017 volatile int terminate = 0;
00023 pthread_mutex_t terminate_mutex = PTHREAD_MUTEX_INITIALIZER;
00024
00029 const char MQUEUE[] = "/tcfs_queue";
00030
00036 void handle_termination(int signum) {
00037     print_msg("TCFS TERMINATED.\n");
00038     //remove_empty_queue(queue_id);
00039     exit(0);
00040 }
00041
00047 int main() {
00048     signal(SIGTERM, handle_termination);
00049
00050     print_msg("TCFS daemon is starting");
00051
00052     /*pid_t pid;
00053
00054     // Fork off the parent process
00055     pid = fork();
00056
00057     // An error occurred
00058     if (pid < 0)
00059         exit(EXIT_FAILURE);
00060
00061     // Success: Let the parent terminate
00062     if (pid > 0)
00063         exit(EXIT_SUCCESS);
00064
00065     // On success: The child process becomes session leader
00066     if (setsid() < 0)
00067         exit(EXIT_FAILURE);
00068
00069     // Catch, ignore and handle signals
00070     signal(SIGCHLD, SIG_IGN);
00071     signal(SIGHUP, SIG_IGN);
00072
```

```
    // Fork off for the second time
    pid = fork();

    // An error occurred
    if (pid < 0)
        exit(EXIT_FAILURE);

    // Success: Let the parent terminate
    if (pid > 0)
        exit(EXIT_SUCCESS);

    // Set new file permissions
    umask(0);

    // Change the working directory to the root directory
    // or another appropriated directory
    chdir("/");

    // Close all open file descriptors
    int x;
    for (x = sysconf(_SC_OPEN_MAX); x>=0; x--)
    {
        close (x);
    }*/

    pthread_t thread1, thread2;

    mqd_t queue_id = init_queue((char *)MQUEUE);
    printf("TEST %d", (int)queue_id);
    if (queue_id == 0)
    {
        print_err("Cannot open message queue in %s", (char *)MQUEUE);
        unlink(MQUEUE);
        return -errno;
    }

    if (pthread_create(&thread1, NULL, handle_incoming_messages, &queue_id) != 0) {
        print_err("Failed to create thread1");
        mq_close(queue_id);
        unlink(MQUEUE);
        return -errno;
    }

    if (pthread_create(&thread2, NULL, handle_outgoing_messages, &queue_id) != 0) {
        print_err("Failed to create thread1");
        mq_close(queue_id);
        unlink(MQUEUE);
        return -errno;
    }

    while (!terminate) {}

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    mq_close(queue_id);
    unlink(MQUEUE);


    print_err("TCFS daemon threads returned, this should have never happened");

    return -1;
}
```

6.20 tcfs_kmodule.c

```
/*
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");

static char *key = NULL;
static size_t key_size = 0;

SYSCALL_DEFINE2(putkey, char __user *, user_key, size_t, size)
{
char *new_key = kmalloc(size, GFP_KERNEL);
if (!new_key)
return -ENOMEM;

if (copy_from_user(new_key, user_key, size)) {
kfree(new_key);
return -EFAULT;
}

kfree(key);
key = new_key;
key_size = size;

return 0;
}

SYSCALL_DEFINE2(getkey, char __user *, user_key, size_t, size)
{
if (size < key_size)
return -EINVAL;

if (copy_to_user(user_key, key, key_size))
return -EFAULT;

return key_size;
}
*/
```

6.21 tcfs_helper_tools.c

```
#include "tcfs_helper_tools.h"

#define PASS_SIZE 33

int handle_local_mount();
int handle_remote_mount();
int handle_folder_mount();

int do_mount()
{
    int choice = -1;
    do
    {
        printf("Chose between:\n"
               "\t1. Network FS\n"
               "\t2. Local FS\n"
               "\t3. Local folder");
        scanf("%d", &choice);
        if (choice != 1 && choice != 2 && choice != 3)
            printf("Err: Select 1 or 2\n");
    } while (choice != 1 && choice != 2 && choice != 3);
    printf("You chose %d\n", choice);

    if (choice == 1)
    {
        return handle_remote_mount();
    } else if (choice == 2)
    {
        return handle_local_mount();
    } else if (choice == 3)
    {
        return handle_folder_mount();
    }
    printf("Unrecoverable error\n");
    return 0;
}

int generate_random_string(char *str)
{
    if (str == NULL)
        return 0;
    for (int i = 0; i < 10; i++)
        str[i] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"[rand() % 62];
    str[10] = '\0';
    return 1;
}

int directory_exists(const char *path) {
    struct stat sb;
    return stat(path, &sb) == 0 && S_ISDIR(sb.st_mode);
}

char *setup_env()
{
    printf("SETUP ENV\n");
    char *home = getenv("HOME");
    printf("$HOME=%s\n",home);

    char *tcfs_path = malloc((strlen(home) + strlen("/.tcfs\0")) * sizeof(char));
    char rand_path_name[11];
    char *new_path = NULL;

    if (home == NULL)
    {
        perror("Could not get $HOME\n");
        return 0;
    }

    if (tcfs_path == NULL)
    {
        perror("Could not allocate string tcfs_path");
        return 0;
    }
    sprintf(tcfs_path, "%s/%s", home, ".tcfs");

    //$HOME/.tcfs does not exist if this is true
    if (directory_exists(tcfs_path) == 0)
    {
        if (mkdir(tcfs_path, 0770) == -1)
        {
            perror("Cannot create .tcfs directory");
            return 0;
        }
    }
    //Create a folder to mount the source to
    //Generate a random path name
    if (generate_random_string(rand_path_name) == 0)
    {
        fprintf(stderr, "Err: Name generation for temp folder failed\n");
        return 0;
    }
    //Build the path from / to the generated path
    new_path = malloc((strlen(rand_path_name) + strlen(tcfs_path) + 1) * sizeof(char));
    if (new_path == NULL)
    {
        perror("Cannot allocate new memory for path name");
        return 0;
    }
    sprintf(new_path, "%s/%s", tcfs_path, rand_path_name);
    if (mkdir(new_path, 0770) == -1)
    {
        perror("Cannot create the tmp folder inside .tcfs");
        return 0;
    }

    printf("New path %s\n", new_path);
    free(tcfs_path);
    return new_path;
}

void get_pass (char *pw) {
    struct termios old, new;
    int i = 0;
    int ch = 0;

    // Disable character echo
    tcgetattr(STDIN_FILENO, &old);
    new = old;
    new.c_lflag &= ~ECHO;
    tcsetattr(STDIN_FILENO, TCSANOW, &new);

    printf("Please enter a password exactly %d characters long:\n", PASS_SIZE);

    while (strlen(pw)*sizeof(char) < (PASS_SIZE-1)*sizeof(char))
    {
        while (1)
        {
            ch = getchar();
            if (ch == ’\r’ || ch == ’\n’ || ch == EOF) {
                break;
            }
            if (i < PASS_SIZE - 1)
            {
                pw[i] = ch;
                pw[i + 1] = ’\0’;
            }
            i++;
        }
    }

    // Restore terminal settings
    tcsetattr(STDIN_FILENO, TCSANOW, &old);
    printf("\nPassword successfully entered!\n");
}
```

```

void get_source_dest(char *source, char *dest)
{
    printf("Please type the path to the source\n");
    scanf("%s", source);

    printf("Please type where it should be mounted\n");
    scanf("%s", dest);
}

char *create_tcfs_mount_folder()
{
    char *tmp_path = NULL;

    //Create a folder to mount it to
    srand(time(NULL));
    char random_string[11];
    if (generate_random_string(random_string) == 0)
    {
        fprintf(stderr, "Err: cannot generate a folder to mount to\n");
        return 0;
    }
    tmp_path = setup_env();
    if (tmp_path == NULL)
    {
        fprintf(stderr, "Err: could not get temp path\n");
        return 0;
    }
    printf("Creating dir: %s\n", tmp_path);
    return tmp_path;
}

int mount_tcfs_folder(char *tmp_path, char *destination)
{
    char pass[PASS_SIZE] = "\0";
    struct termios old, new;

    // Disable character echo
    tcgetattr(STDIN_FILENO, &old);
    new = old;
    new.c_lflag &= ~ECHO;
    tcsetattr(STDIN_FILENO, TCSANOW, &new);

    get_pass(pass);
    if (pass[0] == '\0')
    {
        tcsetattr(STDIN_FILENO, TCSANOW, &old);
        fprintf(stderr, "Could not get password\n");
        return 0;
    }

    //Mount tmpfolder to the destination
    char *tcfs_command = malloc((strlen("tcfs -s ")+strlen(tmp_path)+ strlen(" -d ")+
                                 strlen(destination)+ strlen(" -p ")+strlen(pass)));
    sprintf(tcfs_command, "tcfs -s %s -d %s -p %s", tmp_path, destination, pass);

    int status_tcfs_mount = system(tcfs_command);
    if (!(WIFEXITED(status_tcfs_mount) && WEXITSTATUS(status_tcfs_mount) == 0))
    {
        tcsetattr(STDIN_FILENO, TCSANOW, &old);
        perror("Could not execute the command");
        return 0;
    }
    free(tcfs_command);
    tcsetattr(STDIN_FILENO, TCSANOW, &old);
    return 1;
}

int handle_local_mount()
{
    char source[PATH_MAX];
    char destination[PATH_MAX];
    char *tmp_path = NULL;

    get_source_dest(source, destination);

    tmp_path = create_tcfs_mount_folder();
    if (tmp_path == NULL)
    {
        printf("Err: could not get tmp folder path\n");
        return 0;
    }

    //Mount block device to temp folder
    char *command = malloc((strlen("mount ") + strlen(source) + strlen(" ") + strlen(tmp_path)) * sizeof(char));
    if (command == NULL)
    {
        perror("cannot allocate memoty for the command");
        return 0;
    }
    sprintf(command, "sudo mount -o umask=0755,gid=1000,uid=1000 %s %s", source, tmp_path);
    printf("executing: %s\n", command);
    int status_tmp_mount = system(command);
    if (!(WIFEXITED(status_tmp_mount) && WEXITSTATUS(status_tmp_mount) == 0)) {
        perror("Could not execute the command");
        return 0;
    }

    int res = mount_tcfs_folder(tmp_path, destination);
    if (res == 0) return 0;

    free(tmp_path);
    free(command);
    return 1;
}

int handle_folder_mount()
{
    char source[PATH_MAX];
    char destination[PATH_MAX];

    get_source_dest(source, destination);
    if (source[0] == '\0' || destination[0] == '\0')
    {
        printf("Err: Could not get source or destination\n");
        return 0;
    }
    printf("Source:%s\tdestination:%s\n", source, destination);

    int res = mount_tcfs_folder(source, destination);
    if (res == 0) return 0;

    return 1;
}

void clearKeyboardBuffer() {
    int ch;
    while ((ch = getchar()) != EOF && ch != '\n');
}

int handle_remote_mount()
{
    char source[PATH_MAX] = "\0";
    char destination[PATH_MAX] = "\0";
    char command[100] = "\0";

    printf("WARN: This function is not complete, I don't know how many remote FileSystems support extended "
           "attributes, please mount it manually. "
           "\nEX:sudo mount -t nfs -o umask=0755,gid=1000,uid=1000 10.10.10.10:/NFS /mnt\n");


    clearKeyboardBuffer();
    printf("Enter the command: ");
    int ch;
    int loop = 0;
    while (loop < 99 && (ch = getc(stdin)) != EOF && ch != '\n') {
        command[loop] = ch;
        ++loop;
    }
    command[loop] = '\0'; // Null-terminate the string

    printf("Command: %s\n", command);
    int status= system(command);
    if (!(WIFEXITED(status) && WEXITSTATUS(status) == 0)) {
        perror("Could not execute the command");
        return 0;
    }

    printf("Where has it been mounted? ");
    loop = 0;
    while (loop < PATH_MAX - 1 && (ch = getc(stdin)) != EOF && ch != '\n') {
        source[loop] = ch;
        ++loop;
    }
    source[loop] = '\0'; // Null-terminate the string

    printf("Source: %s\n", source);

    printf("Where should TCFS mount it? ");
    loop = 0;
    while (loop < PATH_MAX - 1 && (ch = getc(stdin)) != EOF && ch != '\n') {
        destination[loop] = ch;
        ++loop;
    }
    destination[loop] = ’\0’; // Null-terminate the string

    printf("Destination: %s\n", destination);


    int res = mount_tcfs_folder(source, destination);
    return res;
}
```

6.22 tcfs_helper_tools.h

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <termios.h>

int do_mount();
```

6.23 user_tcfs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <argp.h>
#include "tcfs_helper_tools.h"

// Define the program documentation
const char *argp_program_version = "TCFS user helper program";
const char *argp_program_bug_address = "carloalbertogiordano@duck.com";
static char doc[] = "TCFS user accepts one of three arguments: mount, create-shared, or umount.";

// Define the accepted options
static struct argp_option options[] = {
        {"mount", ’m’, 0, 0, "Perform mount operation", -1},
        {"create-shared", ’c’, 0, 0, "Perform create-shared operation", -1},
        {"umount", ’u’, 0, 0, "Perform umount operation", -1},
        {NULL}
};

// Structure to hold the parsed arguments
struct arguments {
    int operation;
};

// Parse the arguments
static error_t parse_opt(int key, char *arg, struct argp_state *state) {
    (void )arg;

    struct arguments *arguments = state->input;
    switch (key) {
        case ’m’:
            arguments->operation = 1; // Mount
            break;
        case ’c’:
            arguments->operation = 2; // Create-shared
            break;
        case ’u’:
            arguments->operation = 3; // Umount
            break;
        default:
            return ARGP_ERR_UNKNOWN;
    }
    return 0;
}

// Define the argp object
static struct argp argp = {
        .options = options,
        .parser = parse_opt,
        .doc = doc,
        .args_doc = NULL,
        .children = NULL,
        .help_filter = NULL
};

int main(int argc, char *argv[]) {
    struct arguments arguments;
    arguments.operation = 0; // Default value

    // Parse the arguments
    argp_parse(&argp, argc, argv, 0, 0, &arguments);

    arguments.operation = 1; //TODO: option 1 is the only one implemented
    switch (arguments.operation) {
        case 1:
            printf("Mounting your FS, Please specify the location\n");
            int result = do_mount();
            if (result == 0)
            {
                fprintf(stderr, "An error occurred\n");
                exit(-1);
            }
            break;
        case 2:
            printf("You chose the ’create-shared’ operation.\n");
            // Add specific logic for ’create-shared’ here.
            break;
        case 3:
            printf("You chose the ’umount’ operation.\n");
            // Add specific logic for ’umount’ here.
            break;
        default:
            printf("Invalid argument. Choose from ’mount’, ’create-shared’, or ’umount’.\n");
            return 1;
    }

    return 0;
}
```

6.24 tcfs.c

```
#define FUSE_USE_VERSION 30
#define HAVE_SETXATTR

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

/* For pread()/pwrite() */
#if __STDC_VERSION__ >= 199901L
# define _XOPEN_SOURCE 600
#else
# define _XOPEN_SOURCE 500
#endif /* __STDC_VERSION__ */

#include <assert.h>
#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <linux/limits.h>
#include <dirent.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/xattr.h>
#include <fcntl.h>             /* Definition of AT_* constants */
#include <sys/stat.h>
#include <time.h>
#include <limits.h>
#include <argp.h>
#include <pwd.h>
#include "utils/tcfs_utils/tcfs_utils.h"
#include "utils/crypt-utils/crypt-utils.h"

char *root_path;
char *password;

static int tcfs_getxattr(const char *fuse_path, const char *name, char *value, size_t size);

static int tcfs_opendir(const char *fuse_path, struct fuse_file_info *fi)
{
    (void) fuse_path;
    (void) fi;
    printf("Called opendir UNIMPLEMENTED\n");
    /*int res = 0;
    DIR *dp;
    char path[PATH_MAX];

    *path = prefix_path(fuse_path);

    dp = opendir(path);
    if (dp == NULL)
        res = -errno;

    fi->fh = (intptr_t) dp;

    return res;*/
    return 0;
}

static int tcfs_getattr(const char *fuse_path, struct stat *stbuf)
{
    printf("Called getattr\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = stat(path, stbuf);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_access(const char *fuse_path, int mask)
{
    printf("Callen access\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = access(path, mask);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_readlink(const char *fuse_path, char *buf, size_t size)
{
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = readlink(path, buf, size - 1);
    if (res == -1)
        return -errno;

    buf[res] = ’\0’;
    return 0;
}

static int tcfs_readdir(const char *fuse_path, void *buf, fuse_fill_dir_t filler,
                       off_t offset, struct fuse_file_info *fi)
{
    (void) offset;
    (void) fi;

    printf("Called readdir %s\n", fuse_path);
    char *path = prefix_path(fuse_path, root_path);

    DIR *dp;
    struct dirent *de;

    dp = opendir(path);
    if (dp == NULL)
    {
        perror("Could not open the directory");
        return -errno;
    }

    while ((de = readdir(dp)) != NULL) {
        struct stat st;
        memset(&st, 0, sizeof(st));
        st.st_ino = de->d_ino;
        st.st_mode = de->d_type « 12;
        if (filler(buf, de->d_name, &st, 0))
            break;
    }

    closedir(dp);
    return 0;
}
```

```c

static int tcfs_mknod(const char *fuse_path, mode_t mode, dev_t rdev)
{
    printf("Called mknod\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    /* On Linux this could just be ’mknod(path, mode, rdev)’ but this
       is more portable */
    if (S_ISREG(mode)) {
        res = open(path, O_CREAT | O_EXCL | O_WRONLY, mode);
        if (res >= 0)
            res = close(res);
    } else if (S_ISFIFO(mode))
        res = mkfifo(path, mode);
    else
        res = mknod(path, mode, rdev);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_mkdir(const char *fuse_path, mode_t mode)
{
    printf("Called mkdir\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = mkdir(path, mode);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_unlink(const char *fuse_path)
{
    printf("Called unlink\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = unlink(path);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_rmdir(const char *fuse_path)
{
    printf("Called rmdir\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = rmdir(path);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_symlink(const char *from, const char *to)
{
    printf("Called symlink\n");
    int res;

    res = symlink(from, to);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_rename(const char *from, const char *to)
{
    printf("Called rename\n");
    int res;

    res = rename(from, to);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_link(const char *from, const char *to)
{
    printf("Called link\n");
    int res;

    res = link(from, to);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_chmod(const char *fuse_path, mode_t mode)
{
    printf("Called chmod\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = chmod(path, mode);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_chown(const char *fuse_path, uid_t uid, gid_t gid)
{
    printf("Called chown\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = lchown(path, uid, gid);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_truncate(const char *fuse_path, off_t size)
{
    printf("Called truncate\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = truncate(path, size);
    if (res == -1)
        return -errno;

    return 0;
}

//#ifdef HAVE_UTIMENSAT
static int tcfs_utimens(const char *fuse_path, const struct timespec ts[2])
{
    printf("Called utimens\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;
    struct timeval tv[2];

    tv[0].tv_sec = ts[0].tv_sec;
    tv[0].tv_usec = ts[0].tv_nsec / 1000;
    tv[1].tv_sec = ts[1].tv_sec;
    tv[1].tv_usec = ts[1].tv_nsec / 1000;

    res = utimes(path, tv);
    if (res == -1)
        return -errno;

    return 0;
}
//#endif

static int tcfs_open(const char *fuse_path, struct fuse_file_info *fi)
{
    printf("Called open\n");
    char *path = prefix_path(fuse_path, root_path);
    int res;

    res = open(path, fi->flags);
    if (res == -1)
```

```
        return -errno;

    close(res);
    return 0;
}

static inline int file_size(FILE *file) {
    struct stat st;

    if (fstat(fileno(file), &st) == 0)
        return st.st_size;

    return -1;
}

static int tcfs_read(const char *fuse_path, char *buf, size_t size, off_t offset, struct
fuse_file_info *fi)
{
    (void) size;
    (void) fi;

    printf("Calling read\n");
    FILE *path_ptr, *tmpf;
    char *path;
    int res;

    //Retrieve the username
    char username_buf[1024];
    size_t username_buf_size = 1024;
    get_user_name(username_buf, username_buf_size);

    path = prefix_path(fuse_path, root_path);

    path_ptr = fopen(path, "r");
    tmpf = tmpfile();

    //Get key size
    char* size_key_char = malloc(sizeof(char) * 20);
    if (tcfs_getxattr(fuse_path, "user.key_len", size_key_char, 20) == -1)
    {
        perror("Could not get file key size");
        return -errno;
    }
    ssize_t size_key = strtol(size_key_char, NULL, 10);

    //Retrive the file key
    unsigned char *encrypted_key = malloc((size_key+1) * sizeof(char));
    encrypted_key[size_key] = '\0';
    if (tcfs_getxattr(fuse_path, "user.key", (char *)encrypted_key, size_key) == -1){
        perror("Could not get encrypted key for file in tcfs_read");
        return -errno;
    }

    //Decrypt the file key
    unsigned char *decrypted_key;
    decrypted_key = decrypt_string(encrypted_key, password);

    /* Decrypt*/
    if (do_crypt(path_ptr, tmpf, DECRYPT, decrypted_key) != 1)
    {
        perror("Err: do_crypt cannot decrypt file");
        return -errno;
    }

    /* Something went terribly wrong if this is the case. */
    if (path_ptr == NULL || tmpf == NULL)
        return -errno;

    if (fflush(tmpf) != 0)
    {
        perror("Err: Cannot flush file in read process");
        return -errno;
    }
    if (fseek(tmpf, offset, SEEK_SET) != 0)
    {
        perror("Err: cannot fseek while reading file");
        return -errno;
    }

    /* Read our tmpfile into the buffer. */
    res = fread(buf, 1, file_size(tmpf), tmpf);
    if (res == -1) {
        perror("Err: cannot fread whine in read");
        res = -errno;
    }

    fclose(tmpf);
```

```c
    fclose(path_ptr);
    free(encrypted_key);
    free(decrypted_key);
    return res;
}

static int tcfs_write(const char *fuse_path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
{
    (void) fi;
    printf("Called write\n");

    FILE *path_ptr, *tmpf;
    char *path;
    int res;
    int tmpf_descriptor;

    path = prefix_path(fuse_path, root_path);
    path_ptr = fopen(path, "r+");
    tmpf = tmpfile();
    tmpf_descriptor = fileno(tmpf);

    //Get the key size
    char* size_key_char = malloc(sizeof(char) * 20);
    if (tcfs_getxattr(fuse_path, "user.key_len", size_key_char, 20) == -1)
    {
        perror("Could not get file key size");
        return -errno;
    }
    ssize_t size_key = strtol(size_key_char, NULL, 10);

    //Retrieve the file key
    unsigned char *encrypted_key = malloc(sizeof(unsigned char) * (size_key+1));
    encrypted_key[size_key] = '\0';
    if (tcfs_getxattr(fuse_path, "user.key", (char *)encrypted_key, size_key) == -1){
        perror("Could not get file encrypted key in tcfs write");
        return -errno;
    }

    //Decrypt the file key
    unsigned char *decrypted_key = malloc(sizeof(unsigned char) * 33);
    decrypted_key[32] = '\0';
    decrypted_key = decrypt_string(encrypted_key, password);

    /* Something went terribly wrong if this is the case. */
    if (path_ptr == NULL || tmpf == NULL) {
        fprintf(stderr, "Something went terribly wrong, cannot create new files\n");
        return -errno;
    }

    /* if the file to write to exists, read it into the tempfile */
    if (tcfs_access(fuse_path, R_OK) == 0 && file_size(path_ptr) > 0) {
        if (do_crypt(path_ptr, tmpf, DECRYPT, decrypted_key) == 0) {
            perror("do_crypt: Cannot cypher file\n");
            return --errno;
        }
        rewind(path_ptr);
        rewind(tmpf);
    }

    /* Read our tmpfile into the buffer. */
    res = pwrite(tmpf_descriptor, buf, size, offset);
    if (res == -1){
        printf("%d\n", res);
        perror("pwrite: cannot read tmpfile into the buffer\n");
        res = -errno;
    }

    /* Encrypt*/
    if (do_crypt(tmpf, path_ptr, ENCRYPT, decrypted_key) == 0) {
        perror("do_crypt 2: cannot cypher file\n");
        return -errno;
    }

    fclose(tmpf);
    fclose(path_ptr);
    free(encrypted_key);
    free(decrypted_key);

    return res;
}

static int tcfs_statfs(const char *fuse_path, struct statvfs *stbuf)
{
    printf("Called statfs\n");
    char *path = prefix_path(fuse_path, root_path);

    int res;

    res = statvfs(path, stbuf);
    if (res == -1)
        return -errno;

    return 0;
}

static int tcfs_setxattr(const char *fuse_path, const char *name, const char *value, size_t size, int flags)
{
    char *path = prefix_path(fuse_path, root_path);
    int res = 1;
    if ((res = lsetxattr(path, name, value, size, flags)) == -1)
        perror("tcfs_lsetxattr");
    if (res == -1)
        return -errno;
    return 0;
}

static int tcfs_create(const char* fuse_path, mode_t mode, struct fuse_file_info* fi)
{
    (void) fi;
    (void) mode;
    printf("Called create\n");

    FILE *res;
    res = fopen(prefix_path(fuse_path, root_path), "w");
    if(res == NULL)
        return -errno;

    //Flag file as encrypted
    if (tcfs_setxattr(fuse_path, "user.encrypted", "true", 4, 0) != 0) //(fsetxattr(fileno(res), "user.encrypted", "true", 4, 0) != 0)
    {
        fclose(res);
        return -errno;
    }

    //Generate and set a new encrypted key for the file
    unsigned char *key = malloc(sizeof(unsigned char) * 33);
    key[32] = '\0';
    generate_key(key);

    if (key == NULL)
    {
        perror("cannot generate file key");
        return -errno;
    }
    if (is_valid_key(key) == 0)
    {
        fprintf(stderr, "Generated key size invalid\n");
        return -1;
    }

    //Encrypt the generated key
    int encrypted_key_len;
    unsigned char *encrypted_key = encrypt_string(key, password, &encrypted_key_len);

    //Set the file key
    if (tcfs_setxattr(fuse_path, "user.key", (const char *)encrypted_key, encrypted_key_len, 0) != 0) //(fsetxattr(fileno(res), "user.key", encrypted_key, 32, 0) != 0)
    {
        perror("Err setting key xattr");
        return -errno;
    }
    //Set key size
    char encrypted_key_len_char[20];
    snprintf(encrypted_key_len_char, sizeof(encrypted_key_len_char), "%d", encrypted_key_len);
    if (tcfs_setxattr(fuse_path, "user.key_len", encrypted_key_len_char , sizeof(encrypted_key_len_char), 0) != 0) //(fsetxattr(fileno(res), "user.key", encrypted_key, 32, 0) != 0)
    {
        perror("Err setting key_len xattr");
        return -errno;
    }

    free(encrypted_key);
    free(key);
    fclose(res);
    return 0;
}


static int tcfs_release(const char *fuse_path, struct fuse_file_info *fi)
{
    /* Just a stub.  This method is optional and can safely be left
       unimplemented */
    char *path = prefix_path(fuse_path, root_path);

    (void) path;
    (void) fi;
    return 0;
}

static int tcfs_fsync(const char *fuse_path, int isdatasync,
                     struct fuse_file_info *fi)
{
    /* Just a stub.  This method is optional and can safely be left
       unimplemented */
    char *path = prefix_path(fuse_path, root_path);

    (void) path;
    (void) isdatasync;
    (void) fi;
    return 0;
}

static int tcfs_getxattr(const char *fuse_path, const char *name, char *value, size_t size)
{
    char *path = prefix_path(fuse_path, root_path);
    printf("Called getxattr on %s name:%s size:%zu\n", path, name, size);

    if (strcmp(name, "security.capability") == 0) //TODO: I don't know why this is called every time, understand why and handle this
        return 0;

    int res = (int)lgetxattr(path, name, value, size);
    if (res == -1)
    {
        perror("Could not get xattr for file");
        return -errno;
    }
    return res;
}

static int tcfs_listxattr(const char *fuse_path, char *list, size_t size)
{
    printf("Called listxattr\n");
    char *path = prefix_path(fuse_path, root_path);

    int res = llistxattr(path, list, size);
    if (res == -1)
        return -errno;
    return res;
}

static int tcfs_removexattr(const char *fuse_path, const char *name)
{
    printf("Called removexattr\n");
    char *path = prefix_path(fuse_path, root_path);

    int res = lremovexattr(path, name);
    if (res == -1)
        return -errno;
    return 0;
}

static struct fuse_operations tcfs_oper = {
        .opendir        = tcfs_opendir,
        .getattr    = tcfs_getattr,
        .access     = tcfs_access,
        .readlink   = tcfs_readlink,
        .readdir    = tcfs_readdir,
        .mknod      = tcfs_mknod,
        .mkdir      = tcfs_mkdir,
        .symlink    = tcfs_symlink,
        .unlink     = tcfs_unlink,
        .rmdir      = tcfs_rmdir,
        .rename     = tcfs_rename,
        .link       = tcfs_link,
        .chmod      = tcfs_chmod,
        .chown      = tcfs_chown,
        .truncate   = tcfs_truncate,
        .utimens    = tcfs_utimens,
        .open       = tcfs_open,
        .read       = tcfs_read,
        .write      = tcfs_write,
        .statfs     = tcfs_statfs,
        .create     = tcfs_create,
        .release    = tcfs_release,
        .fsync      = tcfs_fsync,
        .setxattr   = tcfs_setxattr,
        .getxattr   = tcfs_getxattr,
        .listxattr  = tcfs_listxattr,
        .removexattr    = tcfs_removexattr,
};

const char *argp_program_version = "TCFS Alpha";
const char *argp_program_bug_address = "carloalbertogiordano@duck.com";

static char doc[] = "This is an implementation on TCFS\ntcfs -s <source_path> -d <dest_path> -p <password> [fuse arguments]";

static char args_doc[] = "";

static struct argp_option options[] = {
        {"source", 's', "SOURCE", 0, "Source file path", -1},
        {"destination", 'd', "DESTINATION", 0, "Destination file path", -1},
        {"password", 'p', "PASSWORD", 0, "Password", -1},
        {NULL}
};

struct arguments {
    char *source;
    char *destination;
    char *password;
};

static error_t parse_opt(int key, char *arg, struct argp_state *state) {
    struct arguments *arguments = state->input;

    switch (key) {
        case 's':
            arguments->source = arg;
            break;
        case 'd':
            arguments->destination = arg;
            break;
        case 'p':
            arguments->password = arg;
            break;
        case ARGP_KEY_ARG:
            return ARGP_ERR_UNKNOWN;
        default:
            return ARGP_ERR_UNKNOWN;
    }

    return 0;
}

static struct argp argp = {options, parse_opt, args_doc, doc, 0, NULL, NULL};

int main(int argc, char *argv[])
{
    umask(0);

    struct arguments arguments;

    arguments.source = NULL;
    arguments.destination = NULL;
    arguments.password = NULL;

    argp_parse(&argp, argc, argv, 0, 0, &arguments);

    if (arguments.source == NULL || arguments.destination == NULL || arguments.password == NULL) {
        printf("Err: You need to specify at least 3 arguments\n");
        return -1;
    }

    printf("Source: %s\n", arguments.source);
    printf("Destination: %s\n", arguments.destination);
    root_path = arguments.source;

    if (is_valid_key((unsigned char *)arguments.password) == 0){
        fprintf(stderr, "Inserted key not valid\n");
        return 1;
    }

    struct fuse_args args_fuse = FUSE_ARGS_INIT(0, NULL);
    fuse_opt_add_arg(&args_fuse, "./tcfs");
    fuse_opt_add_arg(&args_fuse, arguments.destination);
    fuse_opt_add_arg(&args_fuse, "-f"); //TODO: this is forced for now, but will be passed via options in the future
    fuse_opt_add_arg(&args_fuse, "-s"); //TODO: this is forced for now, but will be passed via options in the future

    //Print what we are passing to fuse TODO: This will be removed
    for (int i=0; i < args_fuse.argc; i++) {
        printf("%s ", args_fuse.argv[i]);
    }
    printf("\n");

    //Get username
    /*
    char buf[1024];
    size_t buf_size = 1024;
    get_user_name(buf, buf_size);
    */

    password = arguments.password;

    return fuse_main(args_fuse.argc, args_fuse.argv, &tcfs_oper, NULL);
}
```

6.25 crypt-utils.c

```
/*
 * High level function interface for performing AES encryption on FILE pointers
 * Uses OpenSSL libcrypto EVP API
 *
 * By Andy Sayler (www.andysayler.com)
 * Created  04/17/12
 * Modified 18/10/23 by [Carlo Alberto Giordano]
 *
 * Derived from OpenSSL.org EVP_Encrypt_* Manpage Examples
 * http://www.openssl.org/docs/crypto/EVP_EncryptInit.html#EXAMPLES
 *
 * With additional information from Saju Pillai's OpenSSL AES Example
 * http://saju.net.in/blog/?p=36
 * http://saju.net.in/code/misc/openssl_aes.c.txt
 *
 */
#include "crypt-utils.h"

#define BLOCKSIZE 1024
#define IV_SIZE 32
#define KEY_SIZE 32

extern int do_crypt(FILE* in, FILE* out, int action, unsigned char *key_str){
    /* Local Vars */

    /* Buffers */
    unsigned char inbuf[BLOCKSIZE];
    int inlen;
    /* Allow enough space in output buffer for additional cipher block */
    unsigned char outbuf[BLOCKSIZE + EVP_MAX_BLOCK_LENGTH];
    int outlen;
    int writelen;

    /* OpenSSL libcrypto vars */
    EVP_CIPHER_CTX *ctx;
    ctx = EVP_CIPHER_CTX_new();

    unsigned char key[KEY_SIZE];
    unsigned char iv[IV_SIZE];
    int nrounds = 5;

    /* tmp vars */
    int i;
    /* Setup Encryption Key and Cipher Engine if in cipher mode */
    if(action >= 0){
        if(!key_str){
            /* Error */
            fprintf(stderr, "Key_str must not be NULL\n");
            return 0;
        }
        /* Build Key from String */
        i = EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(), NULL,
                           key_str, (int)strlen((const char *)key_str), nrounds, key, iv);
        if (i != 32) {
            /* Error */
            fprintf(stderr, "Key size is %d bits - should be 256 bits\n", i*8);
            return 0;
        }
        /* Init Engine */
        EVP_CIPHER_CTX_init(ctx);
        EVP_CipherInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv, action);
    }

    /* Loop through Input File*/
    for(;;){
        /* Read Block */
        inlen = fread(inbuf, sizeof(*inbuf), BLOCKSIZE, in);
        if(inlen <= 0){
            /* EOF -> Break Loop */
            break;
        }

        /* If in cipher mode, perform cipher transform on block */
        if(action >= 0){
            if(!EVP_CipherUpdate(ctx, outbuf, &outlen, inbuf, inlen))
            {
                /* Error */
                EVP_CIPHER_CTX_cleanup(ctx);
                return 0;
            }
        }
            /* If in pass-through mode. copy block as is */
        else{
            memcpy(outbuf, inbuf, inlen);
            outlen = inlen;
        }

        /* Write Block */
        writelen = fwrite(outbuf, sizeof(*outbuf), outlen, out);
        if(writelen != outlen){
            /* Error */
            perror("fwrite error");
            EVP_CIPHER_CTX_cleanup(ctx);
            return 0;
        }
    }

    /* If in cipher mode, handle necessary padding */
    if(action >= 0){
        /* Handle remaining cipher block + padding */
        if(!EVP_CipherFinal_ex(ctx, outbuf, &outlen))
        {
            /* Error */
            EVP_CIPHER_CTX_cleanup(ctx);
            return 0;
        }
        /* Write remainign cipher block + padding*/
        fwrite(outbuf, sizeof(*inbuf), outlen, out);
        EVP_CIPHER_CTX_cleanup(ctx);
    }

    /* Success */
    return 1;
}

// Verify the entropy
int check_entropy(void) {
    FILE *entropy_file = fopen("/proc/sys/kernel/random/entropy_avail", "r");
    if (entropy_file == NULL) {
        perror("Err: Cannot open entropy file");
        return -1;
    }

    int entropy_value;
    if (fscanf(entropy_file, "%d", &entropy_value) != 1) {
        perror("Err: Cannot estimate entropy");
        fclose(entropy_file);
        return -1;
    }

    fclose(entropy_file);
    return entropy_value;
}

//Add new entropy
void add_entropy(void) {
    FILE *urandom = fopen("/dev/urandom", "rb");
    if (urandom == NULL) {
        perror("Err: Cannot open /dev/urandom");
        exit(EXIT_FAILURE);
    }

    unsigned char random_data[32];
    size_t bytes_read = fread(random_data, 1, sizeof(random_data), urandom);
    fclose(urandom);

    if (bytes_read != sizeof(random_data)) {
        fprintf(stderr, "Err: Cannot read data\n");
        exit(EXIT_FAILURE);
    }

    // Usa i dati casuali per aggiungere entropia
    RAND_add(random_data, sizeof(random_data), 0.5); // 0.5 è un peso arbitrario

    fprintf(stdout, "Entropy added successfully!\n");
}


void generate_key(unsigned char *destination) {
    fprintf(stdout, "Generating a new key...\n");

    //Why? Because if we try to create a large number of files there might not be enough random bytes in the system to generate a key
    for (int i = 0; i < 10; i++) {
        int entropy = check_entropy();
        if (entropy < 128) {
            fprintf(stderr, "WARN: not enough entropy, creating some...\n");
            add_entropy();
        }

        if (RAND_bytes(destination, 32) != 1) {
            fprintf(stderr, "Err: Cannot generate key\n");
            destination = NULL;
        }

        if (strlen((const char *)destination) == 32)
            break;
    }

    if (is_valid_key(destination) == 0) {
        fprintf(stderr, "Err: Generated key is inval1d\n");
        print_aes_key(destination);
        destination = NULL;
    }
}

unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int *encrypted_key_len) {
    EVP_CIPHER_CTX* ctx;
    const EVP_CIPHER* cipher = EVP_aes_256_cbc();
    unsigned char iv[AES_BLOCK_SIZE];
    memset(iv, 0, AES_BLOCK_SIZE);

    ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        return NULL;
    }

    EVP_EncryptInit_ex(ctx, cipher, NULL, (const unsigned char*)key, iv);

    size_t plaintext_len = strlen((const char*)plaintext);
    unsigned char ciphertext[plaintext_len + AES_BLOCK_SIZE];
    memset(ciphertext, 0, sizeof(ciphertext));

    int len;
    EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
    EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
    EVP_CIPHER_CTX_free(ctx);

    unsigned char* encoded_string = malloc(len * 2 + 1);
    if (!encoded_string) {
        return NULL;
    }

    for (int i = 0; i < len; i++) {
        sprintf((char*)&encoded_string[i * 2], "%02x", ciphertext[i]);
    }
    encoded_string[len * 2] = ’\0’;

    *encrypted_key_len = len * 2;
    return encoded_string;
}


unsigned char* decrypt_string(unsigned char* ciphertext, const char* key) {
    EVP_CIPHER_CTX *ctx;
    const EVP_CIPHER *cipher = EVP_aes_256_cbc(); // Choose the correct algorithm
    unsigned char iv[AES_BLOCK_SIZE];
    memset(iv, 0, AES_BLOCK_SIZE);

    ctx = EVP_CIPHER_CTX_new();
    EVP_DecryptInit_ex(ctx, cipher, NULL, (const unsigned char*)key, iv);

    size_t decoded_len = strlen((const char *)ciphertext);

    unsigned char plaintext[decoded_len];
    memset(plaintext, 0, sizeof(plaintext));

    int len;
    EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, (int )decoded_len);
    EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
    EVP_CIPHER_CTX_free(ctx);

    unsigned char* decrypted_string = (unsigned char*)malloc(decoded_len + 1);
    memcpy(decrypted_string, plaintext, decoded_len);
    decrypted_string[decoded_len] = '\0';


    return decrypted_string;
}

int is_valid_key(const unsigned char* key)
{
    char str[33];
    memcpy(str, key, 32);
    str[32] = '\0';
    size_t key_length = strlen(str);
    return key_length != 32 ? 0:1;
}


/*
int rebuild_key(char *key, char *cert, char *dest){
    return -1;
}*/
```

6.26 crypt-utils.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

#include <openssl/evp.h>
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <openssl/bio.h>
#include <openssl/buffer.h>

#include "../tcfs_utils/tcfs_utils.h" //TODO: Remove, for debugging only

#define BLOCKSIZE 1024
#define ENCRYPT 1
#define DECRYPT 0

/* int do_crypt(FILE* in, FILE* out, int action, char* key_str)
 * Purpose: Perform cipher on in File* and place result in out File*
 * Args: FILE* in      : Input File Pointer
 *       FILE* out     : Output File Pointer
 *       int action    : Cipher action (1=encrypt, 0=decrypt, -1=pass-through (copy))
 *   unsigned char *key_str : C-string containing passphrase from which key is derived
 * Return: 0 on error, 1 on success
 */
extern int do_crypt(FILE* in, FILE* out, int action, unsigned char *key_str);

/* void generate_key(unsigned char *destination)
 * Purpose: Generate an AES 256 key of size 32 bytes
 * Args: unsigned char *destination    : The destination for the generated key. it must be 33 bytes long to account for a \0
 * Return: void, if the generation failed an error will be thrown
 */
void generate_key(unsigned char *destination);

/*unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int *encrypted_len)
 * Purpose: Encrypt a string with AES-256
 * Args: unsigned char* plaintext   : The plaintext to be encrypted
 *       const char* key            : The key for the encryption
 *       int *encrypted_len         : This will be filled with the encrypted text length
 * Return: The encrypted string + \0. On error null is returned
 * */
unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int *encrypted_len);

/*unsigned char* decrypt_string(unsigned char* base64_ciphertext, const char* key);
 * Purpose: Decrypt a string with AES-256
 * Args: unsigned char* base64_ciphertext   : The cyphertext to be decrypted
 *       const char* key            : The key for the decryption
 * Return: The decrypted string + \0. On error null is returned
 * */
unsigned char* decrypt_string(unsigned char* base64_ciphertext, const char* key);

/*int is_valid_key(const unsigned char* key);
 * Purpose: Check if a AES-256 key is valid
 * Args: unsigned char* key   : The key to be checked
 * Return: 1 if the key is valid, 0 if it is invalid
 * */
int is_valid_key(const unsigned char* key);

/*
int rebuild_key(char *key, char *cert, char *dest);
*/
```

6.27 password_manager.c

```
//TODO: This util will handle requesting keys to kernel

#include "password_manager.h"
#include "../crypt-utils/crypt-utils.h"
/*
char *true_key;

int insert_key(char* key, char* cert, int is_sys_call)
{
    if (is_sys_call == WITH_SYS_CALL)
    {
        fprintf(stderr, "The kernal module has not been implemented yet, saving key in userspace\n \
                        This will change in the future");
        insert_key(key, cert, WITHOUT_SYS_CALL);
    }
    return rebuild_key(key, cert, true_key);
}

char *request_key(int is_sys_call){
    return NULL;
}
int delete_key(int is_sys_call){
    return -1;
}*/
```

6.28 password_manager.h

```
#include <stddef.h>
#include <stdio.h>

#define WITH_SYS_CALL 1
#define WITHOUT_SYS_CALL 0
/*
int insert_key(char* key, char* cert, int is_sys_call);
char *request_key(int is_sys_call);
int delete_key(int is_sys_call);*/
```

6.29 tcfs_utils.c

```
#include "tcfs_utils.h"
#include "../crypt-utils/crypt-utils.h"

void get_user_name(char *buf, size_t size)
{
    uid_t uid = geteuid();
    struct passwd *pw = getpwuid(uid);
    if (pw)
        snprintf(buf, size, "%s", pw->pw_name);
    else
        perror("Error: Could not retrieve username.\n");
}

/* is_encrypted: returns 1 if file is encrypted, 0 otherwise*/
int is_encrypted(const char *path)
{
    int ret;
    char xattr_val[5];
    getxattr(path, "user.encrypted", xattr_val, sizeof(char)*5);
    xattr_val[4] == '\n';

    return strcmp(xattr_val, "true") == 0 ? 1:0;
}

char *prefix_path(const char *path, const char *realpath)
{
```

```c
    if (path == NULL || realpath == NULL)
    {
        perror("Err: path or realpath is NULL");
        return NULL;
    }

    size_t len = strlen(path) + strlen(realpath) + 1;
    char *root_dir = malloc(len * sizeof(char));

    if (root_dir == NULL)
    {
        perror("Err: Could not allocate memory while in prefix_path");
        return NULL;
    }

    if (strcpy(root_dir, realpath) == NULL)
    {
        perror("strcpy: Cannot copy path");
        return NULL;
    }
    if (strcat(root_dir, path) == NULL)
    {
        perror("strcat: in prefix_path cannot concatenate the paths");
        return NULL;
    }
    return root_dir;
}

/* read_file: for debugging tempfiles */
int read_file(FILE *file)
{
    int c;
    int file_contains_something = 0;
    FILE *read = file; /* don’t move original file pointer */
    if (read) {
        while ((c = getc(read)) != EOF) {
            file_contains_something = 1;
            putc(c, stderr);
        }
    }
    if (!file_contains_something)
        fprintf(stderr, "file was empty\n");
    rewind(file);
    /* fseek(tmpf, offset, SEEK_END); */
    return 0;
}
/* Get the xattr value describing the key of a file
 * return 1 on success else 0
 * */
int get_encrypted_key(char *filepath, unsigned char *encrypted_key)
{
    printf("\tGet Encrypted key for file %s\n", filepath);
    if (is_encrypted(filepath) == 1) {
        printf("\t\tencrypted file\n");

        FILE *src_file = fopen(filepath, "r");
        if (src_file == NULL)
        {
            fclose(src_file);
            perror("Could not open the file to get the key");
            return -errno;
        }
        int src_fd;
        src_fd = fileno(src_file);
        if (src_fd == -1)
        {
            fclose(src_file);
            perror("Could not get fd for the file");
            return -errno;
        }

        if (fgetxattr(src_fd, "user.key", encrypted_key,  33) != -1) {
            fclose(src_file);
            return 1;
        }
    }
    return 0;
}
/*For debugging only*/
void print_aes_key(unsigned char *key) {
    printf("AES HEX:%s -> ", key);
    for (int i = 0; i < 32; i++) {
        printf("%02x", key[i]);
    }
    printf("\n");
}
```

6.30 tcfs_utils.h

```
#include <string.h>
#include <stdio.h>
#include <pwd.h>
#include <unistd.h>
#include <sys/xattr.h>
#include <stdlib.h>
#include <errno.h>

/* void get_user_name(char *buf, size_t size)
 * Purpose: Fetch the username of the current user
 * Args: char *buf      : The username will be written to this buffer
 *       size_t size    : The size of the buffer;
 * Return: Nothing
 */
void get_user_name(char *buf, size_t size);

/* is_encrypted: returns 1 if encryption succeeded, 0 otherwise. There is currently no use for this function */
int is_encrypted(const char *path);

/* char *prefix_path(const char *path))
 * Purpose: Prefix the realpath to the fuse path
 * Args: char *path      : The fuse path
 *       char *realpath  : The realpath
 * Return: NULL on error, char* on success
 */
char *prefix_path(const char *path, const char *realpath);

/* read_file: for debugging tempfiles */
int read_file(FILE *file);

/* int get_encrypted_key(char *filepath, void *encrypted_key)
 * Purpose: Get the encrypted file key from its xattrs
 * Args: char *filepath      : The full-path of the file
 *       char *encrypted_key  : The buffer to save the encrypted key to
 * Return: 0 on error, 1 on success
 */
int get_encrypted_key(char *filepath, unsigned char *encrypted_key);

/*For debugging only*/
void print_aes_key(unsigned char *key);
```

Index