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

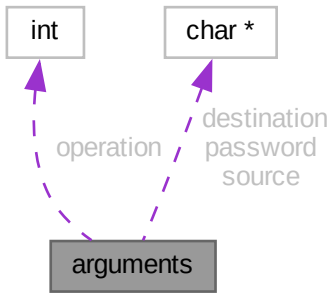
Here is a list of all documented files with brief descriptions:

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
```

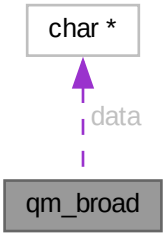
Definition at line 667 of file tcfs.c.

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
```

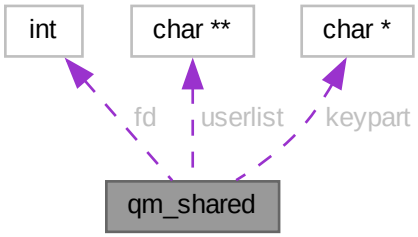
Definition at line 41 of file common.h.

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
```

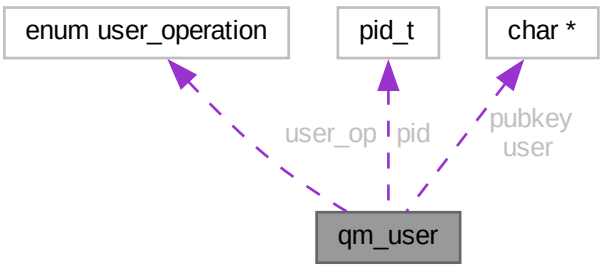
Definition at line 36 of file common.h.

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

```
00001 #include <sys/stat.h>
00002 #include <sys/types.h>
00003 #include <fcntl.h>
00004 #include <mqueue.h>
00005 #include <unistd.h>
00006 #include <string.h>
00007 #include <errno.h>
00008
00009 #define MAX_QM_SIZE 512 //Max size of a message
00010 #define MAX_QM_N 100 //Max number of messages that can be enqueued
00011
00012 #ifndef QUEUE_STRUCTS
00013 #define QUEUE_STRUCTS
00014
00015 typedef enum qm_type{
00016     USER = 0,
00017     SHARED = 1,
00018     BROADCAST = 2,
00019     QM_TYPE_UNDEFINED = -1,
00020 } qm_type;
00021
00022 typedef enum user_operation{
00023     REGISTER = 0,
00024     UNREGISTER = 1,
00025 } user_operation;
00026
00027 typedef struct qm_user {
00028     user_operation user_op;
00029     pid_t pid;
00030     char *user;
00031     char *pubkey;
00032 } qm_user;
00033
00034 typedef struct qm_shared {
00035     int fd;
00036     char **userlist;
00037     char *keypart;
00038 } qm_shared;
00039
00040 typedef struct qm_broad {
00041     char *data;
00042 } qm_broad;
00043
00044 #endif
```

6.2 redis.c

```
00001
00008 #include "redis.h"
00009 #include "../print/print_utils.h"
00010 #include "../json/json_tools.h"
00011 #include <hiredis/hiredis.h>
00012
00019 const char HOST[] = "127.0.0.1";
```

```
00026 #define PORT 6380
00027
00033 redisContext *context;
00034
00040 void print_all_keys() {
00041     redisReply *keys_reply = (redisReply *)redisCommand(context, "KEYS *");
00042     if (keys_reply) {
00043         if (keys_reply->type == REDIS_REPLY_ARRAY) {
00044             for (size_t i = 0; i < keys_reply->elements; ++i) {
00045                 print_msg("\tKey: %s", keys_reply->element[i]->str);
00046             }
00047         } else {
00048             print_msg("Error retrieving keys: %s", keys_reply->str);
00049         }
00050         freeReplyObject(keys_reply);
00051     } else {
00052         print_msg("Error executing KEYS command");
00053     }
00054 }
00060 int init_context()
00061 {
00062     //Do not reinit the context
00063     if (context != NULL)
00064         return 1;
00065
00066     context = redisConnect(HOST, PORT);
00067     if (context->err) {
00068         print_err("Connection error: %s", context->errstr);
00069         return 0;
00070     }
00071     return 1;
00072 }
00078 void free_context()
00079 {
00080     redisFree(context);
00081 }
00088 qm_user *json_to_qm_user(char *json)
00089 {
00090     print_msg("DEBUG: Converting %s", json);
00091     qm_type type;
00092     //Redis return the value as json:{actual json} so we need to eliminate the json: from the string
00093     char *res = strchr(json, ':');
00094     res++; //Skip the : char
00095     qm_user *user = (qm_user *)string_to_struct(res, &type);
00096     return user;
00097 }
00104 qm_user *get_user_by_pid(pid_t pid) {
00105     qm_user *user = NULL;
00106     // Retrieve the JSON data from Redis hash
00107     print_msg("EXECUTING \"GET pid:%d\"", pid);
00108     redisReply *luaReply = (redisReply *)redisCommand(context, "GET pid:%d", pid);
00109     if (luaReply) {
00110         if (luaReply->type == REDIS_REPLY_STRING) {
00111             user = json_to_qm_user(luaReply->str);
00112             if (user) {
00113                 print_msg("Successful retrieval! PID: %d, User: %s", user->pid, user->user);
00114             } else {
00115                 print_err("Error converting JSON to struct");
00116             }
00117         } else {
00118             print_err("Reply type error %d -> executing HGET\n\tErrString: %s",
00119                       luaReply->type, luaReply->str,context->errstr);
00120         }
00121         freeReplyObject(luaReply);
00122     } else {
00123         print_err("Reply error executing HGET\n\tErrString: %s", context->errstr);
00124     }
00125     return user;
00126 }
00133 qm_user *get_user_by_name(const char *name) {
00134     qm_user *user = NULL;
00135     // Retrieve the JSON data from Redis hash
00136     print_msg("EXECUTING \"GET name:%d\"", name);
00137     redisReply *luaReply = (redisReply *)redisCommand(context, "GET name:%d", name);
00138     if (luaReply) {
00139         if (luaReply->type == REDIS_REPLY_STRING) {
00140             user = json_to_qm_user(luaReply->str);
00141             if (user) {
00142                 print_msg("Successful retrieval! PID: %d, User: %s", user->pid, user->user);
00143             } else {
00144                 print_err("Error converting JSON to struct");
00145             }
00146         } else {
00147             print_err("Reply type error %d -> executing HGET\n\tErrString: %s",
00148                       luaReply->type, luaReply->str,context->errstr);
00149         }
00150         freeReplyObject(luaReply);
```

```
00151     } else {
00152         print_err("Reply error executing HGET\n\tErrString: %s", context->errstr);
00153     }
00154     return user;
00155 }
00164 int insert(qm_user *user)
00165 {
00166     // Convert the structure to JSON
00167     const char *json = struct_to_json(USER, user);
00168     if (!json)
00169     {
00170         print_err("Error converting qm_user to JSON");
00171         return 0;
00172     }
00173     // Save to Redis with key "pid_str"
00174     print_msg("\tDB: \"SET pid:%d json:%s\"", user->pid, json);
00175     redisReply *reply_pid =(redisReply *) redisCommand(context, "SET pid:%d json:%s", user->pid,
      json);
00176     if (!reply_pid)
00177     {
00178         print_err("Error saving to Redis (pid)");
00179         free((void *)json);
00180         return 0;
00181     }
00182     freeReplyObject(reply_pid);
00183 
00184     // Save to Redis with key "user"
00185     redisReply *reply_user =(redisReply *) redisCommand(context, "SET user:%s json:%s", user->user,
      json);
00186     if (!reply_user)
00187     {
00188         print_err("Error saving to Redis (user)");
00189         free((void *)json);
00190         return 0;
00191     }
00192     freeReplyObject(reply_user);
00193     // Free the allocated JSON memory
00194     free((void *)json); //Discard qualifier
00195     return 1;
00196 }
00205 int remove_by_pid(pid_t pid)
00206 {
00207     qm_user *user_tmp = get_user_by_pid(pid);
00208     // Remove the structure by PID
00209     print_msg("\tDB: \"DEL pid:%d\"", pid);
00210     redisReply *reply_pid =(redisReply *) redisCommand(context, "DEL pid:%d", pid);
00211     if (!reply_pid) {
00212         print_err("Error removing structure by PID");
00213         return 0;
00214     }
00215     freeReplyObject(reply_pid);
00216     // Also remove the corresponding key by name
00217     print_msg("\tDB: \"DEL user:%s\"", user_tmp->user);
00218     redisReply *reply_name =(redisReply *) redisCommand(context, "DEL user:%s", user_tmp->user);
00219     if (!reply_name) {
00220         print_err("Error removing key by name");
00221         return 0;
00222     }
00223     free(user_tmp);
00224     freeReplyObject(reply_name);
00225     return 1;
00226 }
00235 int remove_by_user(char *name)
00236 {
00237     qm_user *user_tmp = get_user_by_name(name);
00238     // Remove the structure by name
00239     char key_name[64]; // Adjust the size as needed
00240     snprintf(key_name, sizeof(key_name), "user:%s", name);
00241     redisReply *reply_name =(redisReply *) redisCommand(context, "DEL %s", key_name);
00242     if (!reply_name) {
00243         print_err("Error removing structure by name");
00244         return 0;
00245     }
00246     freeReplyObject(reply_name);
00247     // Also remove the corresponding key by PID
00248     redisReply *reply_pid =(redisReply *) redisCommand(context, "DEL %d", user_tmp->pid);
00249     if (!reply_pid) {
00250         print_err("Error removing key by PID");
00251         return 0;
00252     }
00253     freeReplyObject(reply_pid);
00254     return 1;
00255 }
```

6.3 redis.h

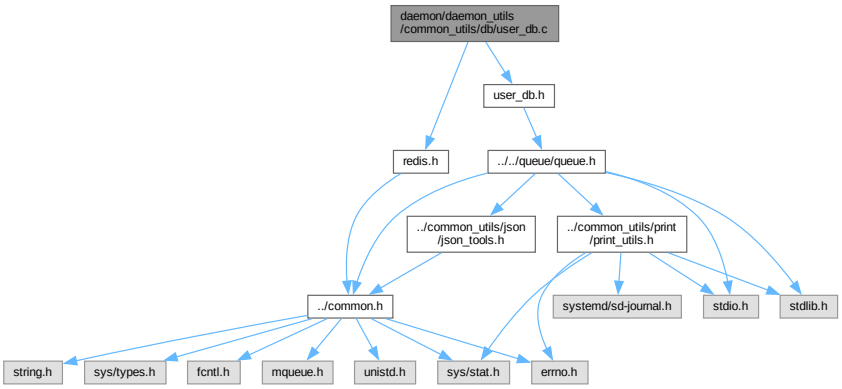
```
00001 #include "../../common.h"
00002
00003 void print_all_keys();
00004
00005 int init_context();
00006
00007 qm_user *json_to_qm_user(char *json);
00008
00009 qm_user *get_user_by_pid(pid_t pid);
00010
00011 qm_user *get_user_by_name(const char *name);
00012
00013 int insert(qm_user *user);
00014
00015 int remove_by_pid(pid_t pid);
00016
00017 int remove_by_user(char *name);
00018
00019 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:

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.

```
00001 #include "user_db.h"
00002 #include "redis.h"
00003
00014 int register_user(qm_user *user_msg)
00015 {
00016     print_msg("Registering new user");
00017     if (init_context() == 0)
00018         return 0;
00019     print_all_keys();
00020     if (insert(user_msg) == 0)
00021         return 0;
00022     return 1;
00023 }
00029 int unregister_user(pid_t pid)
00030 {
00031     print_all_keys();
00032     print_msg("Removing user");
00033     return remove_by_pid(pid);
00034 }
00041 void disconnect_db(void)
00042 {
00043     print_msg("Freeing context...");
00044     free_context();
00045 }
```

6.6 user_db.h

```
00001 #include "../../queue/queue.h"
00002
00003 int register_user(qm_user *user_msg);
00004 int unregister_user(pid_t pid);
00005 void disconnect_db(void);
```

6.7 json_tools.cpp

```
00001 #include "../../common.h"
00002 #include <iostream>
00003 #include <string.h>
00004 #include <vector>
00005 #include <cstring> // For strcpy
00006 #include <cstdlib> // For malloc and free
00007 #include "/usr/include/nlohmann/json.hpp" // Assuming you're using nlohmann's JSON library
00008 #include "../print/print_utils.h"
00009
00023 char* struct_to_json(qm_type qmt, void* q_mess) {
00024     nlohmann::json json_obj;
00025
00026     switch (qmt) {
00027         case USER: {
00028             qm_user* user = static_cast<qm_user*>(q_mess);
00029             if (user->user_op == REGISTER)
00030                 print_msg("Register");
00031             if (user->user_op == UNREGISTER)
00032                 print_msg("Unregister");
00033             json_obj["user_op"] = user->user_op;
00034             json_obj["pid"] = user->pid;
00035             json_obj["user"] = user->user;
00036             json_obj["pubkey"] = user->pubkey;
00037             break;
00038         }
00039         case SHARED: {
00040             qm_shared* shared = static_cast<qm_shared*>(q_mess);
00041             json_obj["fd"] = shared->fd;
00042
00043             // Converti la matrice di stringhe in un array di stringhe JSON
00044             nlohmann::json userlist_array = nlohmann::json::array();
00045             for (size_t i = 0; shared->userlist[i] != nullptr; ++i) {
00046                 userlist_array.push_back(shared->userlist[i]);
00047             }
00048             json_obj["userlist"] = userlist_array;
00049
00050             json_obj["keypart"] = shared->keypart;
00051             break;
00052         }
```

```
00053         case BROADCAST: {
00054             qm_broad* broad = static_cast<qm_broad*>(q_mess);
00055             json_obj["data"] = broad->data;
00056             break;
00057         }
00058     }
00059     // Cast Json obj to string
00060     std::string json_str = json_obj.dump();
00061     // Allocate memory for result
00062     char* result = (char*)malloc(json_str.size() + 1);
00063     if (result) {
00064         strcpy(result, json_str.c_str());
00065     }
00066     print_msg("JSONIFIED: %s", result);
00067     return result;
00068 }
00069
00078 void* string_to_struct(const char* json_string, qm_type* type) {
00079     try {
00080         nlohmann::json json_obj = nlohmann::json::parse(json_string);
00081
00082         if (json_obj.contains("user_op")) {
00083             *type = USER;
00084             qm_user* user = static_cast<qm_user*>(std::malloc(sizeof(qm_user)));
00085             user->user_op = json_obj["user_op"];
00086             user->pid = json_obj["pid"];
00087             user->user = strdup(json_obj["user"].get<std::string>().c_str());
00088             user->pubkey = strdup(json_obj["pubkey"].get<std::string>().c_str());
00089             return user;
00090         } else if (json_obj.contains("fd")) {
00091             *type = SHARED;
00092             qm_shared* shared = static_cast<qm_shared*>(std::malloc(sizeof(qm_shared)));
00093             shared->fd = json_obj["fd"];
00094
00095             // Populate userlist array
00096             std::vector<std::string> userlist = json_obj["userlist"];
00097             shared->userlist = static_cast<char**>(std::malloc((userlist.size() + 1) *
    sizeof(char*)));
00098             for (size_t i = 0; i < userlist.size(); ++i) {
00099                 shared->userlist[i] = strdup(userlist[i].c_str());
00100             }
00101             shared->userlist[userlist.size()] = nullptr;
00102
00103             shared->keypart = strdup(json_obj["keypart"].get<std::string>().c_str());
00104             return shared;
00105         } else if (json_obj.contains("data")) {
00106             *type = BROADCAST;
00107             qm_broad* broad = static_cast<qm_broad*>(std::malloc(sizeof(qm_broad)));
00108             broad->data = strdup(json_obj["data"].get<std::string>().c_str());
00109             return broad;
00110         } else {
00111             *type = QM_TYPE_UNDEFINED;
00112             return nullptr;
00113         }
00114     } catch (const std::exception& e) {
00115         std::cerr « "Error parsing JSON: " « e.what() « std::endl;
00116         return nullptr;
00117     }
00118 }
```

6.8 json_tools.h

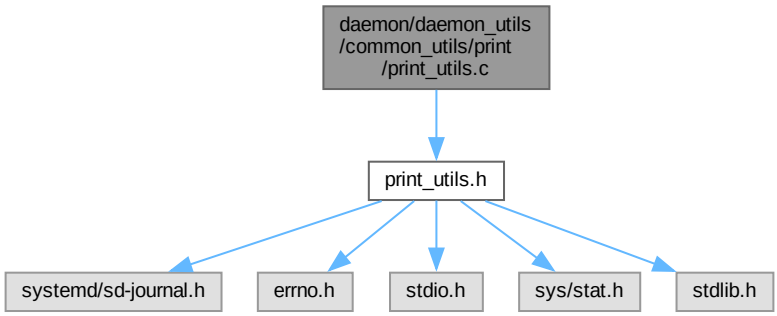
```
00001 #include "../../common.h"
00002
00003 extern const char *struct_to_json(qm_type qmt, void *q_mess);
00004 extern void* string_to_struct(const char* json_string, qm_type* type);
```

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

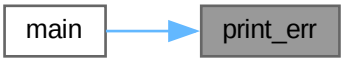
"Err_Numebr:d" will be appended to the formatted string describing the error number

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

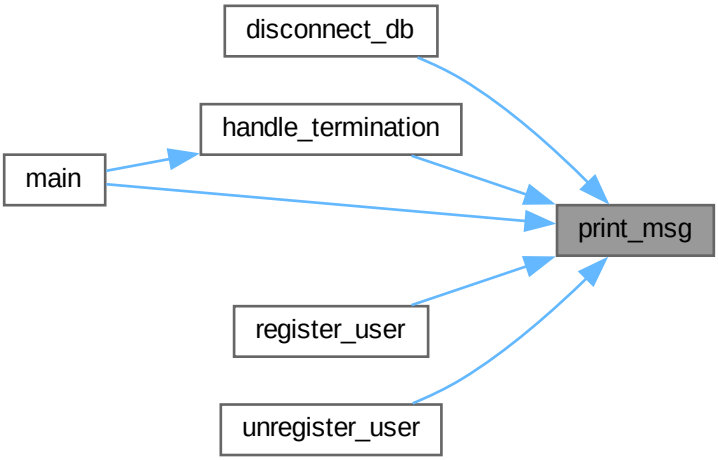
Will also log using systemD

"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.

```
00001 #include "print_utils.h"
00002
00013 int cleared = 0;
00014
00022 void log_message(const char *log){
00023     printf("%s\n", log);
00024     // Path of the log folder and log file
00025     const char *logFolder = "/var/log/tcfs";
00030     const char *logFile = "/var/log/tcfs/log.txt";
00031
00032     // Check if the folder exists, otherwise create it
00033     struct stat st;
00034     if (stat(logFolder, &st) == -1) {
00035         mkdir(logFolder, 0700);
00036     }
00037
00038     FILE *file;
00039     if (cleared == 0)
00040     {
00041         cleared = 1;
00042         file = fopen(logFile, "w");
00043     } else {
00044         file = fopen(logFile, "a");
00045     }
00046
00047     // Open the log file in append mode
00048     if (file == NULL) {
00049         perror("Error opening the log file");
00050     }
00051
00052     // Write the message to the log file
00053     fprintf(file, "%s\n", log);
00054
00055     // Close the file
00056     fclose(file);
00057 }
00058
00069 void print_err(const char *format, ...)
00070 {
00071     va_list args;
00072     va_start(args, format);
00073     char buffer[1024];
00074     vsnprintf(buffer, sizeof(buffer), format, args);
00075     va_end(args);
00076
00077     log_message(buffer);
00078
00079     sd_journal_print(LOG_ERR, "ERROR=%s Err_Number:%d -> %s", buffer, errno, strerror(errno));
00080 }
00089 void print_msg(const char *format, ...)
00090 {
00091     va_list args;
00092     va_start(args, format);
00093     char buffer[1024];
00094     vsnprintf(buffer, sizeof(buffer), format, args);
00095     va_end(args);
00096
00097     log_message(buffer);
00098
00099     sd_journal_send("MESSAGE=%s", buffer, NULL);
00100 }
00101
00110 void print_warn(const char *format, ...)
00111 {
00112     va_list args;
00113     va_start(args, format);
00114     char buffer[1024];
00115     vsnprintf(buffer, sizeof(buffer), format, args);
00116     va_end(args);
00117
00118     log_message(buffer);
00119
00120     sd_journal_print(LOG_WARNING, "WARNING=%s", buffer, NULL);
00121 }
00122
00131 void print_debug(const char *format, ...)
00132 {
00133     va_list args;
00134     va_start(args, format);
00135     char buffer[1024];
00136     vsnprintf(buffer, sizeof(buffer), format, args);
00137     va_end(args);
```

```
00138
00139     log_message(buffer);
00140
00141     sd_journal_print(LOG_DEBUG, "DEBUG=%s", buffer, NULL);
00142 }
```

6.11 print_utils.h

```
00001 #include <systemd/sd-journal.h>
00002 #include <errno.h>
00003 #include <stdio.h>
00004 #include <sys/stat.h>
00005 #include <stdlib.h>
00006
00007 void print_err(const char *format, ...);
00008 void print_msg(const char *format, ...);
00009 void print_warn(const char *format, ...);
00010 void print_debug(const char *format, ...);
```

6.12 tcfs_daemon_tools.c

```
00001 #include "tcfs_daemon_tools.h"
00002 #include "../message_handler/message_handler.h"
00003
00016 void *handle_incoming_messages(void *queue_id)
00017 {
00018     qm_type qmt;
00019     qm_user *user_msg;
00020     qm_shared *shared_msg;
00021     qm_broad *broadcast_msg;
00022
00023
00024     print_msg("Starting handler for incoming messages");
00025     void *tmp_struct;
00026     while (1) {
00027         tmp_struct = dequeue(*(mqd_t *) queue_id, &qmt);
00028         switch (qmt) {
00029             case USER:
00030                 print_msg("Handling user message");
00031                 user_msg = (qm_user *) tmp_struct;
00032                 handle_user_message(user_msg);
00033                 break;
00034             case SHARED:
00035                 print_msg("Handling shared message");
00036                 shared_msg = (qm_shared *) tmp_struct;
00037                 //handle_shared_message()
00038                 break;
00039             case BROADCAST:
00040                 print_msg("Handling broadcast message");
00041                 broadcast_msg = (qm_broad *) tmp_struct;
00042                 //handle_broadcast_message()
00043                 break;
00044             case QM_TYPE_UNDEFINED:
00045                 print_err("Received un unknown message type, skipping...");
00046                 break;
00047         }
00048         free(tmp_struct);
00049     }
00050     return NULL;
00051 }
00052
00060 void *handle_outgoing_messages(void *queue_id)
00061 {
00062     print_msg("Handling outgoing messages");
00063     //sleep(1);
00064
00065     char s1[] = "TEST";
00066     char s2[] = "pubkey";
00067
00068     struct qm_user test_msg;
00069     test_msg.user_op = REGISTER;
00070     test_msg.pid = 104;
00071     test_msg.user = s1;
00072     test_msg.pubkey = s2;
00073
00074     print_msg("Enqueueing test registration...");
00075     int res = enqueue(*(mqd_t *)queue_id, USER, (void *)&test_msg);
00076     print_msg("TEST message send with result %d", res);
00077
```

```
00078     if (res != 1){
00079         print_err("enqueue err ");
00080     }
00081
00082     struct qm_user test_msg2;
00083     test_msg2.user_op = UNREGISTER;
00084     test_msg2.pid = 104;
00085     test_msg2.user = "";
00086     test_msg2.pubkey = "";
00087
00088     sleep(3);
00089
00090     print_msg("Enqueueing test remove...");
00091     res = enqueue(*(mqd_t *)queue_id, USER, (void *)&test_msg2);
00092     print_msg("TEST message send with result %d", res);
00093
00094     if (res != 1){
00095         print_err("enqueue err ");
00096     }
00097
00098     return NULL;
00099 }
00100
00101 /*
00102  *
00103 void* monitor_termination(void* queue_id) {
00104     while (1) {
00105         pthread_mutex_lock(&terminate_mutex);
00106         if (terminate) {
00107             pthread_mutex_unlock(&terminate_mutex);
00108             break;
00109         }
00110         pthread_mutex_unlock(&terminate_mutex);
00111         sleep(1);
00112     }
00113     print_err("Terminating threads");
00114     remove_empty_queue(*(int *)queue_id);
00115     return NULL;
00116 }*/
```

6.13 tcfs_daemon_tools.h

```
00001 #include <stdlib.h>
00002 #include <unistd.h>
00003 #include <fcntl.h>
00004 #include <stdbool.h>
00005 #include <sys/socket.h>
00006 #include <sys/un.h>
00007 #include <sys/stat.h>
00008 #include <pthread.h>
00009 #include <signal.h>
00010 #include "../queue/queue.h"
00011 #include "../message_handler/message_handler.h"
00012
00013 // Condition variable & mutex
00014 extern volatile int terminate;
00015 extern pthread_mutex_t terminate_mutex;
00016
00017 void *handle_incoming_messages(void *queue_id);
00018 void *handle_outgoing_messages(void *queue_id);
00019 void *monitor_termination(void *queue_id);
00020 void cleanup_threads(pthread_t thread1, pthread_t thread2);
```

6.14 message_handler.c

```
00001 #include "message_handler.h"
00002 #include "../common_utils/json/json_tools.h"
00003 #include "../common_utils/db/user_db.h"
00004 #include "../common_utils/print/print_utils.h"
00005
00006 int handle_user_message(qm_user *user_msg)
00007 {
00008     if (user_msg->user_op == REGISTER){
00009         register_user(user_msg);
00010     } else if (user_msg->user_op == UNREGISTER)
00011     {
00012         unregister_user(user_msg->pid);
00013         //TODO: next line is a test, remove it
00014         free_context();
```

```
00015     } else
00016     {
00017         print_err("Unknown user operation %d", user_msg->user_op);
00018         return 0;
00019     }
00020
00021     return 1;
00022 }
```

6.15 message_handler.h

```
00001 #include "../common.h"
00002 #include "../common_utils/print/print_utils.h"
00003
00004 int handle_user_message(qm_user *user_msg);
```

6.16 queue.c

```
00001 #include "queue.h"
00002
00003 mqd_t init_queue(char *queue)
00004 {
00005     struct mq_attr attr;
00006     mqd_t mq;
00007
00008     // Initialize queue attributes
00009     attr.mq_flags = 0;
00010     attr.mq_maxmsg = MAX_QM_N; // Maximum number of messages in the queue
00011     attr.mq_msgsize = MAX_QM_SIZE; // Maximum size of a single message
00012     attr.mq_curmsgs = 0;
00013
00014     // Create the message queue
00015     mq = mq_open(queue, O_CREAT | O_RDWR /*| O_RDONLY | O_NONBLOCK*/, 0777, &attr); //TODO: Better
    define permissions
00016     printf("mqopen %d\n", mq);
00017     if (mq == (mqd_t)-1) {
00018         print_err("mq_open cannot create que in %s %d %s", queue, errno, strerror(errno));
00019         print_msg("mq_open cannot create que in %s %d %s", queue, errno, strerror(errno));
00020         return 0;
00021     }
00022     printf("Message queue created successfully at %s!\n", queue);
00023     return mq;
00024 }
00025 int enqueue(mqd_t queue_d, qm_type qmt, void *q_mess)
00026 {
00027     const char *qm_json = struct_to_json(qmt, q_mess);
00028
00029     if (mq_send(queue_d, qm_json, strlen(qm_json)+1, 0) == -1)
00030     {
00031         print_err("mq_send %s", qm_json);
00032         free((void *)qm_json);
00033         return 0;
00034     }
00035     print_msg("Message sent successfully!\n");
00036     free((void *)qm_json);
00037     return 1;
00038 }
00039 void *dequeue(mqd_t queue_d, qm_type *qmt)
00040 {
00041     char *qm_json = (char *)malloc(sizeof(char ) * MAX_QM_SIZE);
00042
00043     if (mq_receive(queue_d, qm_json, MAX_QM_SIZE, 0) == -1)
00044     {
00045         free((void *)qm_json);
00046         print_err("mq_rec %d %s", errno, strerror(errno));
00047         return NULL;
00048     }
00049
00050     print_msg("Dequeued %s", qm_json);
00051     void *tmp_struct = string_to_struct(qm_json, qmt);
00052
00053     free((void *)qm_json);
00054     return tmp_struct;
00055 }
```

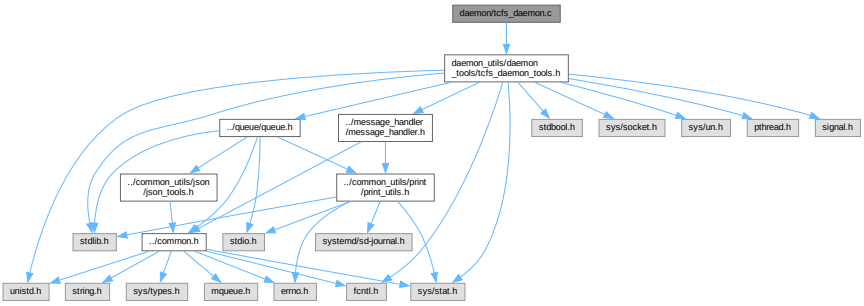
6.17 queue.h

```
00001 #include "../common.h"
00002 #include "../common_utils/print/print_utils.h"
00003 #include "../common_utils/json/json_tools.h"
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006
00007 #define MESSAGE_BUFFER_SIZE 256
00008 #define MQUEUE_N 3;
00009
00010
00011
00012 mqd_t init_queue(char *queue);
00013 int enqueue(mqd_t queue_d, qm_type qmt, void *q_mess);
00014 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

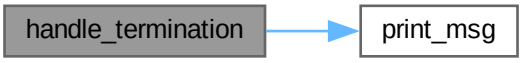
Todo : Implement remove_queue() to clear and delete the queue

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

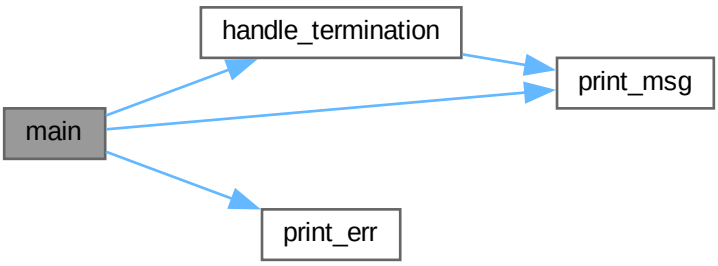
Todo : The brief description is basically false advertisement. It only spawn a thread and hangs infinitely

: 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
```

```
00073     // Fork off for the second time
00074     pid = fork();
00075
00076     // An error occurred
00077     if (pid < 0)
00078         exit(EXIT_FAILURE);
00079
00080     // Success: Let the parent terminate
00081     if (pid > 0)
00082         exit(EXIT_SUCCESS);
00083
00084     // Set new file permissions
00085     umask(0);
00086
00087     // Change the working directory to the root directory
00088     // or another appropriated directory
00089     chdir("/");
00090
00091     // Close all open file descriptors
00092     int x;
00093     for (x = sysconf(_SC_OPEN_MAX); x>=0; x--)
00094     {
00095         close (x);
00096     }*/
00097
00098     pthread_t thread1, thread2;
00099
00100     mqd_t queue_id = init_queue((char *)MQUEUE);
00101     printf("TEST %d", (int)queue_id);
00102     if (queue_id == 0)
00103     {
00104         print_err("Cannot open message queue in %s", (char *)MQUEUE);
00105         unlink(MQUEUE);
00106         return -errno;
00107     }
00108
00109     if (pthread_create(&thread1, NULL, handle_incoming_messages, &queue_id) != 0) {
00110         print_err("Failed to create thread1");
00111         mq_close(queue_id);
00112         unlink(MQUEUE);
00113         return -errno;
00114     }
00115
00116     if (pthread_create(&thread2, NULL, handle_outgoing_messages, &queue_id) != 0) {
00117         print_err("Failed to create thread1");
00118         mq_close(queue_id);
00119         unlink(MQUEUE);
00120         return -errno;
00121     }
00122
00123     while (!terminate) {}
00124
00125     pthread_join(thread1, NULL);
00126     pthread_join(thread2, NULL);
00127
00128     mq_close(queue_id);
00129     unlink(MQUEUE);
00130
00131
00132     print_err("TCFS daemon threads returned, this should have never happened");
00133
00134     return -1;
00135 }
```

6.20 tcfs_kmodule.c

```
00001 /*
00002 #include <linux/kernel.h>
00003 #include <linux/module.h>
00004 #include <linux/syscalls.h>
00005 #include <linux/slab.h>
00006
00007 MODULE_LICENSE("GPL");
00008
00009 static char *key = NULL;
00010 static size_t key_size = 0;
00011
00012 SYSCALL_DEFINE2(putkey, char __user *, user_key, size_t, size)
00013 {
00014 char *new_key = kmalloc(size, GFP_KERNEL);
00015 if (!new_key)
00016 return -ENOMEM;
00017
```

```
00018 if (copy_from_user(new_key, user_key, size)) {
00019 kfree(new_key);
00020 return -EFAULT;
00021 }
00022
00023 kfree(key);
00024 key = new_key;
00025 key_size = size;
00026
00027 return 0;
00028 }
00029
00030 SYSCALL_DEFINE2(getkey, char __user *, user_key, size_t, size)
00031 {
00032 if (size < key_size)
00033 return -EINVAL;
00034
00035 if (copy_to_user(user_key, key, key_size))
00036 return -EFAULT;
00037
00038 return key_size;
00039 }
00040 */
```

6.21 tcfs_helper_tools.c

```
00001 #include "tcfs_helper_tools.h"
00002
00003 #define PASS_SIZE 33
00004
00005 int handle_local_mount();
00006 int handle_remote_mount();
00007 int handle_folder_mount();
00008
00009 int do_mount()
00010 {
00011     int choice = -1;
00012     do
00013     {
00014         printf("Chose between:\n"
00015                "\t1. Network FS\n"
00016                "\t2. Local FS\n"
00017                "\t3. Local folder");
00018         scanf("%d", &choice);
00019         if (choice != 1 && choice != 2 && choice != 3)
00020             printf("Err: Select 1 or 2\n");
00021     } while (choice != 1 && choice != 2 && choice != 3);
00022     printf("You chose %d\n", choice);
00023
00024     if (choice == 1)
00025     {
00026         return handle_remote_mount();
00027     } else if (choice == 2)
00028     {
00029         return handle_local_mount();
00030     } else if (choice == 3)
00031     {
00032         return handle_folder_mount();
00033     }
00034     printf("Unrecoverable error\n");
00035     return 0;
00036 }
00037
00038 int generate_random_string(char *str)
00039 {
00040     if (str == NULL)
00041         return 0;
00042     for (int i = 0; i < 10; i++)
00043         str[i] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"[rand() % 62];
00044     str[10] = '\0';
00045     return 1;
00046 }
00047
00048 int directory_exists(const char *path) {
00049     struct stat sb;
00050     return stat(path, &sb) == 0 && S_ISDIR(sb.st_mode);
00051 }
00052
00053 char *setup_env()
00054 {
00055     printf("SETUP ENV\n");
00056     char *home = getenv("HOME");
00057     printf("$HOME=%s\n",home);
```

```
00058
00059     char *tcfs_path = malloc((strlen(home) + strlen("/.tcfs\0")) * sizeof(char));
00060     char rand_path_name[11];
00061     char *new_path = NULL;
00062
00063     if (home == NULL)
00064     {
00065         perror("Could not get $HOME\n");
00066         return 0;
00067     }
00068
00069     if (tcfs_path == NULL)
00070     {
00071         perror("Could not allocate string tcfs_path");
00072         return 0;
00073     }
00074     sprintf(tcfs_path, "%s/%s", home, ".tcfs");
00075
00076     //$HOME/.tcfs does not exist if this is true
00077     if (directory_exists(tcfs_path) == 0)
00078     {
00079         if (mkdir(tcfs_path, 0770) == -1)
00080         {
00081             perror("Cannot create .tcfs directory");
00082             return 0;
00083         }
00084     }
00085     //Create a folder to mount the source to
00086     //Generate a random path name
00087     if (generate_random_string(rand_path_name) == 0)
00088     {
00089         fprintf(stderr, "Err: Name generation for temp folder failed\n");
00090         return 0;
00091     }
00092     //Build the path from / to the generated path
00093     new_path = malloc((strlen(rand_path_name) + strlen(tcfs_path) + 1) * sizeof(char));
00094     if (new_path == NULL)
00095     {
00096         perror("Cannot allocate new memory for path name");
00097         return 0;
00098     }
00099     sprintf(new_path, "%s/%s", tcfs_path, rand_path_name);
00100     if (mkdir(new_path, 0770) == -1)
00101     {
00102         perror("Cannot create the tmp folder inside .tcfs");
00103         return 0;
00104     }
00105
00106     printf("New path %s\n", new_path);
00107     free(tcfs_path);
00108     return new_path;
00109 }
00110
00111 void get_pass (char *pw) {
00112     struct termios old, new;
00113     int i = 0;
00114     int ch = 0;
00115
00116     // Disable character echo
00117     tcgetattr(STDIN_FILENO, &old);
00118     new = old;
00119     new.c_lflag &= ~ECHO;
00120     tcsetattr(STDIN_FILENO, TCSANOW, &new);
00121
00122     printf("Please enter a password exactly %d characters long:\n", PASS_SIZE);
00123
00124     while (strlen(pw)*sizeof(char) < (PASS_SIZE-1)*sizeof(char))
00125     {
00126         while (1)
00127         {
00128             ch = getchar();
00129             if (ch == '\r' || ch == '\n' || ch == EOF) {
00130                 break;
00131             }
00132             if (i < PASS_SIZE - 1)
00133             {
00134                 pw[i] = ch;
00135                 pw[i + 1] = '\0';
00136             }
00137             i++;
00138         }
00139     }
00140
00141     // Restore terminal settings
00142     tcsetattr(STDIN_FILENO, TCSANOW, &old);
00143     printf("\nPassword successfully entered!\n");
00144 }
```

```
00145
00146 void get_source_dest(char *source, char *dest)
00147 {
00148     printf("Please type the path to the source\n");
00149     scanf("%s", source);
00150
00151     printf("Please type where it should be mounted\n");
00152     scanf("%s", dest);
00153 }
00154
00155 char *create_tcfs_mount_folder()
00156 {
00157     char *tmp_path = NULL;
00158
00159     //Create a folder to mount it to
00160     srand(time(NULL));
00161     char random_string[11];
00162     if (generate_random_string(random_string) == 0)
00163     {
00164         fprintf(stderr, "Err: cannot generate a folder to mount to\n");
00165         return 0;
00166     }
00167     tmp_path = setup_env();
00168     if (tmp_path == NULL)
00169     {
00170         fprintf(stderr, "Err: could not get temp path\n");
00171         return 0;
00172     }
00173     printf("Creating dir: %s\n", tmp_path);
00174     return tmp_path;
00175 }
00176
00177 int mount_tcfs_folder(char *tmp_path, char *destination)
00178 {
00179     char pass[PASS_SIZE] = "\0";
00180     struct termios old, new;
00181
00182     // Disable character echo
00183     tcgetattr(STDIN_FILENO, &old);
00184     new = old;
00185     new.c_lflag &= ~ECHO;
00186     tcsetattr(STDIN_FILENO, TCSANOW, &new);
00187
00188     get_pass(pass);
00189     if (pass[0] == '\0')
00190     {
00191         tcsetattr(STDIN_FILENO, TCSANOW, &old);
00192         fprintf(stderr, "Could not get password\n");
00193         return 0;
00194     }
00195
00196     //Mount tmpfolder to the destination
00197     char *tcfs_command = malloc((strlen("tcfs -s ")+strlen(tmp_path)+ strlen(" -d ")+
00198                                  strlen(destination)+ strlen(" -p ")+strlen(pass)));
00199     sprintf(tcfs_command, "tcfs -s %s -d %s -p %s", tmp_path, destination, pass);
00200
00201     int status_tcfs_mount = system(tcfs_command);
00202     if (!(WIFEXITED(status_tcfs_mount) && WEXITSTATUS(status_tcfs_mount) == 0))
00203     {
00204         tcsetattr(STDIN_FILENO, TCSANOW, &old);
00205         perror("Could not execute the command");
00206         return 0;
00207     }
00208     free(tcfs_command);
00209     tcsetattr(STDIN_FILENO, TCSANOW, &old);
00210     return 1;
00211 }
00212
00213 int handle_local_mount()
00214 {
00215     char source[PATH_MAX];
00216     char destination[PATH_MAX];
00217     char *tmp_path = NULL;
00218
00219     get_source_dest(source, destination);
00220
00221     tmp_path = create_tcfs_mount_folder();
00222     if (tmp_path == NULL)
00223     {
00224         printf("Err: could not get tmp folder path\n");
00225         return 0;
00226     }
00227
00228     //Mount block device to temp folder
00229     char *command = malloc((strlen("mount ") + strlen(source) + strlen(" ") + strlen(tmp_path)) *
    sizeof(char));
00230     if (command == NULL)
```

```
00231     {
00232         perror("cannot allocate memoty for the command");
00233         return 0;
00234     }
00235     sprintf(command, "sudo mount -o umask=0755,gid=1000,uid=1000 %s %s", source, tmp_path);
00236     printf("executing: %s\n", command);
00237     int status_tmp_mount = system(command);
00238     if (!(WIFEXITED(status_tmp_mount) && WEXITSTATUS(status_tmp_mount) == 0)) {
00239         perror("Could not execute the command");
00240         return 0;
00241     }
00242 
00243     int res = mount_tcfs_folder(tmp_path, destination);
00244     if (res == 0) return 0;
00245 
00246     free(tmp_path);
00247     free(command);
00248     return 1;
00249 }
00250 
00251 int handle_folder_mount()
00252 {
00253     char source[PATH_MAX];
00254     char destination[PATH_MAX];
00255 
00256     get_source_dest(source, destination);
00257     if (source[0] == '\0' || destination[0] == '\0')
00258     {
00259         printf("Err: Could not get source or destination\n");
00260         return 0;
00261     }
00262     printf("Source:%s\tdestination:%s\n", source, destination);
00263 
00264     int res = mount_tcfs_folder(source, destination);
00265     if (res == 0) return 0;
00266 
00267     return 1;
00268 }
00269 
00270 void clearKeyboardBuffer() {
00271     int ch;
00272     while ((ch = getchar()) != EOF && ch != '\n');
00273 }
00274 
00275 int handle_remote_mount()
00276 {
00277     char source[PATH_MAX] = "\0";
00278     char destination[PATH_MAX] = "\0";
00279     char command[100] = "\0";
00280 
00281     printf("WARN: This function is not complete, I don't know how many remote FileSystems support
      extended "
00282            "attributes, please mount it manually. "
00283            "\nEX:sudo mount -t nfs -o umask=0755,gid=1000,uid=1000 10.10.10.10:/NFS /mnt\n");
00284 
00285 
00286     clearKeyboardBuffer();
00287     printf("Enter the command: ");
00288     int ch;
00289     int loop = 0;
00290     while (loop < 99 && (ch = getc(stdin)) != EOF && ch != '\n') {
00291         command[loop] = ch;
00292         ++loop;
00293     }
00294     command[loop] = '\0'; // Null-terminate the string
00295 
00296     printf("Command: %s\n", command);
00297     int status= system(command);
00298     if (!(WIFEXITED(status) && WEXITSTATUS(status) == 0)) {
00299         perror("Could not execute the command");
00300         return 0;
00301     }
00302 
00303     printf("Where has it been mounted? ");
00304     loop = 0;
00305     while (loop < PATH_MAX - 1 && (ch = getc(stdin)) != EOF && ch != '\n') {
00306         source[loop] = ch;
00307         ++loop;
00308     }
00309     source[loop] = '\0'; // Null-terminate the string
00310 
00311     printf("Source: %s\n", source);
00312 
00313     printf("Where should TCFS mount it? ");
00314     loop = 0;
00315     while (loop < PATH_MAX - 1 && (ch = getc(stdin)) != EOF && ch != '\n') {
00316         destination[loop] = ch;
```

```
00317         ++loop;
00318     }
00319     destination[loop] = '\0'; // Null-terminate the string
00320 
00321     printf("Destination: %s\n", destination);
00322 
00323 
00324     int res = mount_tcfs_folder(source, destination);
00325     return res;
00326 }
```

6.22 tcfs_helper_tools.h

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <limits.h>
00004 #include <sys/stat.h>
00005 #include <sys/types.h>
00006 #include <unistd.h>
00007 #include <time.h>
00008 #include <string.h>
00009 #include <termios.h>
00010 
00011 int do_mount();
```

6.23 user_tcfs.c

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <argp.h>
00004 #include "tcfs_helper_tools.h"
00005 
00006 // Define the program documentation
00007 const char *argp_program_version = "TCFS user helper program";
00008 const char *argp_program_bug_address = "carloalbertogiordano@duck.com";
00009 static char doc[] = "TCFS user accepts one of three arguments: mount, create-shared, or umount.";
00010 
00011 // Define the accepted options
00012 static struct argp_option options[] = {
00013         {"mount", 'm', 0, 0, "Perform mount operation", -1},
00014         {"create-shared", 'c', 0, 0, "Perform create-shared operation", -1},
00015         {"umount", 'u', 0, 0, "Perform umount operation", -1},
00016         {NULL}
00017 };
00018 
00019 // Structure to hold the parsed arguments
00020 struct arguments {
00021     int operation;
00022 };
00023 
00024 // Parse the arguments
00025 static error_t parse_opt(int key, char *arg, struct argp_state *state) {
00026     (void )arg;
00027 
00028     struct arguments *arguments = state->input;
00029     switch (key) {
00030         case 'm':
00031             arguments->operation = 1; // Mount
00032             break;
00033         case 'c':
00034             arguments->operation = 2; // Create-shared
00035             break;
00036         case 'u':
00037             arguments->operation = 3; // Umount
00038             break;
00039         default:
00040             return ARGP_ERR_UNKNOWN;
00041     }
00042     return 0;
00043 }
00044 
00045 // Define the argp object
00046 static struct argp argp = {
00047         .options = options,
00048         .parser = parse_opt,
00049         .doc = doc,
00050         .args_doc = NULL,
00051         .children = NULL,
00052         .help_filter = NULL
```

```
00053 };
00054
00055 int main(int argc, char *argv[]) {
00056     struct arguments arguments;
00057     arguments.operation = 0; // Default value
00058
00059     // Parse the arguments
00060     argp_parse(&argp, argc, argv, 0, 0, &arguments);
00061
00062     arguments.operation = 1; //TODO: option 1 is the only one implemented
00063     switch (arguments.operation) {
00064         case 1:
00065             printf("Mounting your FS, Please specify the location\n");
00066             int result = do_mount();
00067             if (result == 0)
00068             {
00069                 fprintf(stderr, "An error occurred\n");
00070                 exit(-1);
00071             }
00072             break;
00073         case 2:
00074             printf("You chose the 'create-shared' operation.\n");
00075             // Add specific logic for 'create-shared' here.
00076             break;
00077         case 3:
00078             printf("You chose the 'umount' operation.\n");
00079             // Add specific logic for 'umount' here.
00080             break;
00081         default:
00082             printf("Invalid argument. Choose from 'mount', 'create-shared', or 'umount'.\n");
00083             return 1;
00084     }
00085
00086     return 0;
00087 }
```

6.24 tcfs.c

```
00001 #define FUSE_USE_VERSION 30
00002 #define HAVE_SETXATTR
00003
00004 #ifdef HAVE_CONFIG_H
00005 #include <config.h>
00006 #endif
00007
00008 /* For pread()/pwrite() */
00009 #if __STDC_VERSION__ >= 199901L
00010 # define _XOPEN_SOURCE 600
00011 #else
00012 # define _XOPEN_SOURCE 500
00013 #endif /* __STDC_VERSION__ */
00014
00015 #include <assert.h>
00016 #include <fuse.h>
00017 #include <stdio.h>
00018 #include <string.h>
00019 #include <unistd.h>
00020 #include <linux/limits.h>
00021 #include <dirent.h>
00022 #include <errno.h>
00023 #include <sys/time.h>
00024 #include <sys/xattr.h>
00025 #include <fcntl.h>            /* Definition of AT_* constants */
00026 #include <sys/stat.h>
00027 #include <time.h>
00028 #include <limits.h>
00029 #include <argp.h>
00030 #include <pwd.h>
00031 #include "utils/tcfs_utils/tcfs_utils.h"
00032 #include "utils/crypt-utils/crypt-utils.h"
00033
00034 char *root_path;
00035 char *password;
00036
00037 static int tcfs_getxattr(const char *fuse_path, const char *name, char *value, size_t size);
00038
00039 static int tcfs_opendir(const char *fuse_path, struct fuse_file_info *fi)
00040 {
00041     (void) fuse_path;
00042     (void) fi;
00043     printf("Called opendir UNIMPLEMENTED\n");
00044     /*int res = 0;
00045     DIR *dp;
```

```
00046     char path[PATH_MAX];
00047
00048     *path = prefix_path(fuse_path);
00049
00050     dp = opendir(path);
00051     if (dp == NULL)
00052         res = -errno;
00053
00054     fi->fh = (intptr_t) dp;
00055
00056     return res;*/
00057     return 0;
00058 }
00059
00060 static int tcfs_getattr(const char *fuse_path, struct stat *stbuf)
00061 {
00062     printf("Called getattr\n");
00063     char *path = prefix_path(fuse_path, root_path);
00064
00065     int res;
00066
00067     res = stat(path, stbuf);
00068     if (res == -1)
00069         return -errno;
00070
00071     return 0;
00072 }
00073
00074 static int tcfs_access(const char *fuse_path, int mask)
00075 {
00076     printf("Callen access\n");
00077     char *path = prefix_path(fuse_path, root_path);
00078
00079     int res;
00080
00081     res = access(path, mask);
00082     if (res == -1)
00083         return -errno;
00084
00085     return 0;
00086 }
00087
00088 static int tcfs_readlink(const char *fuse_path, char *buf, size_t size)
00089 {
00090     char *path = prefix_path(fuse_path, root_path);
00091
00092     int res;
00093
00094     res = readlink(path, buf, size - 1);
00095     if (res == -1)
00096         return -errno;
00097
00098     buf[res] = '\0';
00099     return 0;
00100 }
00101
00102 static int tcfs_readdir(const char *fuse_path, void *buf, fuse_fill_dir_t filler,
00103                         off_t offset, struct fuse_file_info *fi)
00104 {
00105     (void) offset;
00106     (void) fi;
00107
00108     printf("Called readdir %s\n", fuse_path);
00109     char *path = prefix_path(fuse_path, root_path);
00110
00111     DIR *dp;
00112     struct dirent *de;
00113
00114     dp = opendir(path);
00115     if (dp == NULL)
00116     {
00117         perror("Could not open the directory");
00118         return -errno;
00119     }
00120
00121     while ((de = readdir(dp)) != NULL) {
00122         struct stat st;
00123         memset(&st, 0, sizeof(st));
00124         st.st_ino = de->d_ino;
00125         st.st_mode = de->d_type « 12;
00126         if (filler(buf, de->d_name, &st, 0))
00127             break;
00128     }
00129
00130     closedir(dp);
00131     return 0;
00132 }
```

```
00133
00134 static int tcfs_mknod(const char *fuse_path, mode_t mode, dev_t rdev)
00135 {
00136     printf("Called mknod\n");
00137     char *path = prefix_path(fuse_path, root_path);
00138
00139     int res;
00140
00141     /* On Linux this could just be 'mknod(path, mode, rdev)' but this
00142        is more portable */
00143     if (S_ISREG(mode)) {
00144         res = open(path, O_CREAT | O_EXCL | O_WRONLY, mode);
00145         if (res >= 0)
00146             res = close(res);
00147     } else if (S_ISFIFO(mode))
00148         res = mkfifo(path, mode);
00149     else
00150         res = mknod(path, mode, rdev);
00151     if (res == -1)
00152         return -errno;
00153
00154     return 0;
00155 }
00156
00157 static int tcfs_mkdir(const char *fuse_path, mode_t mode)
00158 {
00159     printf("Called mkdir\n");
00160     char *path = prefix_path(fuse_path, root_path);
00161
00162     int res;
00163
00164     res = mkdir(path, mode);
00165     if (res == -1)
00166         return -errno;
00167
00168     return 0;
00169 }
00170
00171 static int tcfs_unlink(const char *fuse_path)
00172 {
00173     printf("Called unlink\n");
00174     char *path = prefix_path(fuse_path, root_path);
00175
00176     int res;
00177
00178     res = unlink(path);
00179     if (res == -1)
00180         return -errno;
00181
00182     return 0;
00183 }
00184
00185 static int tcfs_rmdir(const char *fuse_path)
00186 {
00187     printf("Called rmdir\n");
00188     char *path = prefix_path(fuse_path, root_path);
00189
00190     int res;
00191
00192     res = rmdir(path);
00193     if (res == -1)
00194         return -errno;
00195
00196     return 0;
00197 }
00198
00199 static int tcfs_symlink(const char *from, const char *to)
00200 {
00201     printf("Called symlink\n");
00202     int res;
00203
00204     res = symlink(from, to);
00205     if (res == -1)
00206         return -errno;
00207
00208     return 0;
00209 }
00210
00211 static int tcfs_rename(const char *from, const char *to)
00212 {
00213     printf("Called rename\n");
00214     int res;
00215
00216     res = rename(from, to);
00217     if (res == -1)
00218         return -errno;
00219
```

```
00220     return 0;
00221 }
00222
00223 static int tcfs_link(const char *from, const char *to)
00224 {
00225     printf("Called link\n");
00226     int res;
00227
00228     res = link(from, to);
00229     if (res == -1)
00230         return -errno;
00231
00232     return 0;
00233 }
00234
00235 static int tcfs_chmod(const char *fuse_path, mode_t mode)
00236 {
00237     printf("Called chmod\n");
00238     char *path = prefix_path(fuse_path, root_path);
00239
00240     int res;
00241
00242     res = chmod(path, mode);
00243     if (res == -1)
00244         return -errno;
00245
00246     return 0;
00247 }
00248
00249 static int tcfs_chown(const char *fuse_path, uid_t uid, gid_t gid)
00250 {
00251     printf("Called chown\n");
00252     char *path = prefix_path(fuse_path, root_path);
00253
00254     int res;
00255
00256     res = lchown(path, uid, gid);
00257     if (res == -1)
00258         return -errno;
00259
00260     return 0;
00261 }
00262
00263 static int tcfs_truncate(const char *fuse_path, off_t size)
00264 {
00265     printf("Called truncate\n");
00266     char *path = prefix_path(fuse_path, root_path);
00267
00268     int res;
00269
00270     res = truncate(path, size);
00271     if (res == -1)
00272         return -errno;
00273
00274     return 0;
00275 }
00276
00277 //#ifdef HAVE_UTIMENSAT
00278 static int tcfs_utimens(const char *fuse_path, const struct timespec ts[2])
00279 {
00280     printf("Called utimens\n");
00281     char *path = prefix_path(fuse_path, root_path);
00282
00283     int res;
00284     struct timeval tv[2];
00285
00286     tv[0].tv_sec = ts[0].tv_sec;
00287     tv[0].tv_usec = ts[0].tv_nsec / 1000;
00288     tv[1].tv_sec = ts[1].tv_sec;
00289     tv[1].tv_usec = ts[1].tv_nsec / 1000;
00290
00291     res = utimes(path, tv);
00292     if (res == -1)
00293         return -errno;
00294
00295     return 0;
00296 }
00297 //#endif
00298
00299 static int tcfs_open(const char *fuse_path, struct fuse_file_info *fi)
00300 {
00301     printf("Called open\n");
00302     char *path = prefix_path(fuse_path, root_path);
00303     int res;
00304
00305     res = open(path, fi->flags);
00306     if (res == -1)
```

```
00307         return -errno;
00308
00309     close(res);
00310     return 0;
00311 }
00312
00313 static inline int file_size(FILE *file) {
00314     struct stat st;
00315
00316     if (fstat(fileno(file), &st) == 0)
00317         return st.st_size;
00318
00319     return -1;
00320 }
00321
00322 static int tcfs_read(const char *fuse_path, char *buf, size_t size, off_t offset, struct
      fuse_file_info *fi)
00323 {
00324     (void) size;
00325     (void) fi;
00326
00327     printf("Calling read\n");
00328     FILE *path_ptr, *tmpf;
00329     char *path;
00330     int res;
00331
00332     //Retrieve the username
00333     char username_buf[1024];
00334     size_t username_buf_size = 1024;
00335     get_user_name(username_buf, username_buf_size);
00336
00337     path = prefix_path(fuse_path, root_path);
00338
00339     path_ptr = fopen(path, "r");
00340     tmpf = tmpfile();
00341
00342     //Get key size
00343     char* size_key_char = malloc(sizeof(char) * 20);
00344     if (tcfs_getxattr(fuse_path, "user.key_len", size_key_char, 20) == -1)
00345     {
00346         perror("Could not get file key size");
00347         return -errno;
00348     }
00349     ssize_t size_key = strtol(size_key_char, NULL, 10);
00350
00351     //Retrive the file key
00352     unsigned char *encrypted_key = malloc((size_key+1) * sizeof(char));
00353     encrypted_key[size_key] = '\0';
00354     if (tcfs_getxattr(fuse_path, "user.key", (char *)encrypted_key, size_key) == -1){
00355         perror("Could not get encrypted key for file in tcfs_read");
00356         return -errno;
00357     }
00358
00359     //Decrypt the file key
00360     unsigned char *decrypted_key;
00361     decrypted_key = decrypt_string(encrypted_key, password);
00362
00363     /* Decrypt*/
00364     if (do_crypt(path_ptr, tmpf, DECRYPT, decrypted_key) != 1)
00365     {
00366         perror("Err: do_crypt cannot decrypt file");
00367         return -errno;
00368     }
00369
00370     /* Something went terribly wrong if this is the case. */
00371     if (path_ptr == NULL || tmpf == NULL)
00372         return -errno;
00373
00374     if (fflush(tmpf) != 0)
00375     {
00376         perror("Err: Cannot flush file in read process");
00377         return -errno;
00378     }
00379     if (fseek(tmpf, offset, SEEK_SET) != 0)
00380     {
00381         perror("Err: cannot fseek while reading file");
00382         return -errno;
00383     }
00384
00385     /* Read our tmpfile into the buffer. */
00386     res = fread(buf, 1, file_size(tmpf), tmpf);
00387     if (res == -1) {
00388         perror("Err: cannot fread whine in read");
00389         res = -errno;
00390     }
00391
00392     fclose(tmpf);
```

```
00393     fclose(path_ptr);
00394     free(encrypted_key);
00395     free(decrypted_key);
00396     return res;
00397 }
00398
00399 static int tcfs_write(const char *fuse_path, const char *buf, size_t size, off_t offset, struct
      fuse_file_info *fi)
00400 {
00401     (void) fi;
00402     printf("Called write\n");
00403
00404     FILE *path_ptr, *tmpf;
00405     char *path;
00406     int res;
00407     int tmpf_descriptor;
00408
00409     path = prefix_path(fuse_path, root_path);
00410     path_ptr = fopen(path, "r+");
00411     tmpf = tmpfile();
00412     tmpf_descriptor = fileno(tmpf);
00413
00414     //Get the key size
00415     char* size_key_char = malloc(sizeof(char) * 20);
00416     if (tcfs_getxattr(fuse_path, "user.key_len", size_key_char, 20) == -1)
00417     {
00418         perror("Could not get file key size");
00419         return -errno;
00420     }
00421     ssize_t size_key = strtol(size_key_char, NULL, 10);
00422
00423     //Retrieve the file key
00424     unsigned char *encrypted_key = malloc(sizeof(unsigned char) * (size_key+1));
00425     encrypted_key[size_key] = '\0';
00426     if (tcfs_getxattr(fuse_path, "user.key", (char *)encrypted_key, size_key) == -1){
00427         perror("Could not get file encrypted key in tcfs write");
00428         return -errno;
00429     }
00430
00431     //Decrypt the file key
00432     unsigned char *decrypted_key = malloc(sizeof(unsigned char) * 33);
00433     decrypted_key[32] = '\0';
00434     decrypted_key = decrypt_string(encrypted_key, password);
00435
00436     /* Something went terribly wrong if this is the case. */
00437     if (path_ptr == NULL || tmpf == NULL) {
00438         fprintf(stderr, "Something went terribly wrong, cannot create new files\n");
00439         return -errno;
00440     }
00441
00442     /* if the file to write to exists, read it into the tempfile */
00443     if (tcfs_access(fuse_path, R_OK) == 0 && file_size(path_ptr) > 0) {
00444         if (do_crypt(path_ptr, tmpf, DECRYPT, decrypted_key) == 0) {
00445             perror("do_crypt: Cannot cypher file\n");
00446             return --errno;
00447         }
00448         rewind(path_ptr);
00449         rewind(tmpf);
00450     }
00451
00452     /* Read our tmpfile into the buffer. */
00453     res = pwrite(tmpf_descriptor, buf, size, offset);
00454     if (res == -1){
00455         printf("%d\n", res);
00456         perror("pwrite: cannot read tmpfile into the buffer\n");
00457         res = -errno;
00458     }
00459
00460     /* Encrypt*/
00461     if (do_crypt(tmpf, path_ptr, ENCRYPT, decrypted_key) == 0) {
00462         perror("do_crypt 2: cannot cypher file\n");
00463         return -errno;
00464     }
00465
00466     fclose(tmpf);
00467     fclose(path_ptr);
00468     free(encrypted_key);
00469     free(decrypted_key);
00470
00471     return res;
00472 }
00473
00474 static int tcfs_statfs(const char *fuse_path, struct statvfs *stbuf)
00475 {
00476     printf("Called statfs\n");
00477     char *path = prefix_path(fuse_path, root_path);
00478
```

```
00479     int res;
00480
00481     res = statvfs(path, stbuf);
00482     if (res == -1)
00483         return -errno;
00484
00485     return 0;
00486 }
00487
00488 static int tcfs_setxattr(const char *fuse_path, const char *name, const char *value, size_t size, int
      flags)
00489 {
00490     char *path = prefix_path(fuse_path, root_path);
00491     int res = 1;
00492     if ((res = lsetxattr(path, name, value, size, flags)) == -1)
00493         perror("tcfs_lsetxattr");
00494     if (res == -1)
00495         return -errno;
00496     return 0;
00497 }
00498
00499 static int tcfs_create(const char* fuse_path, mode_t mode, struct fuse_file_info* fi)
00500 {
00501     (void) fi;
00502     (void) mode;
00503     printf("Called create\n");
00504
00505     FILE *res;
00506     res = fopen(prefix_path(fuse_path, root_path), "w");
00507     if(res == NULL)
00508         return -errno;
00509
00510     //Flag file as encrypted
00511     if (tcfs_setxattr(fuse_path, "user.encrypted", "true", 4, 0) != 0) //(fsetxattr(fileno(res),
      "user.encrypted", "true", 4, 0) != 0)
00512     {
00513         fclose(res);
00514         return -errno;
00515     }
00516
00517     //Generate and set a new encrypted key for the file
00518     unsigned char *key = malloc(sizeof(unsigned char) * 33);
00519     key[32] = '\0';
00520     generate_key(key);
00521
00522     if (key == NULL)
00523     {
00524         perror("cannot generate file key");
00525         return -errno;
00526     }
00527     if (is_valid_key(key) == 0)
00528     {
00529         fprintf(stderr, "Generated key size invalid\n");
00530         return -1;
00531     }
00532
00533     //Encrypt the generated key
00534     int encrypted_key_len;
00535     unsigned char *encrypted_key = encrypt_string(key, password, &encrypted_key_len);
00536
00537     //Set the file key
00538     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)
00539     {
00540         perror("Err setting key xattr");
00541         return -errno;
00542     }
00543     //Set key size
00544     char encrypted_key_len_char[20];
00545     snprintf(encrypted_key_len_char, sizeof(encrypted_key_len_char), "%d", encrypted_key_len);
00546     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)
00547     {
00548         perror("Err setting key_len xattr");
00549         return -errno;
00550     }
00551
00552     free(encrypted_key);
00553     free(key);
00554     fclose(res);
00555     return 0;
00556 }
00557
00558
00559 static int tcfs_release(const char *fuse_path, struct fuse_file_info *fi)
00560 {
```

```
00561     /* Just a stub.  This method is optional and can safely be left
00562        unimplemented */
00563     char *path = prefix_path(fuse_path, root_path);
00564 
00565     (void) path;
00566     (void) fi;
00567     return 0;
00568 }
00569 
00570 static int tcfs_fsync(const char *fuse_path, int isdatasync,
00571                       struct fuse_file_info *fi)
00572 {
00573     /* Just a stub.  This method is optional and can safely be left
00574        unimplemented */
00575     char *path = prefix_path(fuse_path, root_path);
00576 
00577     (void) path;
00578     (void) isdatasync;
00579     (void) fi;
00580     return 0;
00581 }
00582 
00583 static int tcfs_getxattr(const char *fuse_path, const char *name, char *value, size_t size)
00584 {
00585     char *path = prefix_path(fuse_path, root_path);
00586     printf("Called getxattr on %s name:%s size:%zu\n", path, name, size);
00587 
00588     if (strcmp(name, "security.capability") == 0) //TODO: I don't know why this is called every time,
      understand why and handle this
00589         return 0;
00590 
00591     int res = (int)lgetxattr(path, name, value, size);
00592     if (res == -1)
00593     {
00594         perror("Could not get xattr for file");
00595         return -errno;
00596     }
00597     return res;
00598 }
00599 
00600 static int tcfs_listxattr(const char *fuse_path, char *list, size_t size)
00601 {
00602     printf("Called listxattr\n");
00603     char *path = prefix_path(fuse_path, root_path);
00604 
00605     int res = llistxattr(path, list, size);
00606     if (res == -1)
00607         return -errno;
00608     return res;
00609 }
00610 
00611 static int tcfs_removexattr(const char *fuse_path, const char *name)
00612 {
00613     printf("Called removexattr\n");
00614     char *path = prefix_path(fuse_path, root_path);
00615 
00616     int res = lremovexattr(path, name);
00617     if (res == -1)
00618         return -errno;
00619     return 0;
00620 }
00621 
00622 static struct fuse_operations tcfs_oper = {
00623         .opendir        = tcfs_opendir,
00624         .getattr    = tcfs_getattr,
00625         .access     = tcfs_access,
00626         .readlink   = tcfs_readlink,
00627         .readdir    = tcfs_readdir,
00628         .mknod      = tcfs_mknod,
00629         .mkdir      = tcfs_mkdir,
00630         .symlink    = tcfs_symlink,
00631         .unlink     = tcfs_unlink,
00632         .rmdir      = tcfs_rmdir,
00633         .rename     = tcfs_rename,
00634         .link       = tcfs_link,
00635         .chmod      = tcfs_chmod,
00636         .chown      = tcfs_chown,
00637         .truncate   = tcfs_truncate,
00638         .utimens    = tcfs_utimens,
00639         .open       = tcfs_open,
00640         .read       = tcfs_read,
00641         .write      = tcfs_write,
00642         .statfs     = tcfs_statfs,
00643         .create     = tcfs_create,
00644         .release    = tcfs_release,
00645         .fsync      = tcfs_fsync,
00646         .setxattr   = tcfs_setxattr,
```

```
00647         .getxattr   = tcfs_getxattr,
00648         .listxattr  = tcfs_listxattr,
00649         .removexattr    = tcfs_removexattr,
00650 };
00651
00652 const char *argp_program_version = "TCFS Alpha";
00653 const char *argp_program_bug_address = "carloalbertogiordano@duck.com";
00654
00655 static char doc[] = "This is an implementation on TCFS\ntcfs -s <source_path> -d <dest_path> -p
      <password> [fuse arguments]";
00656
00657 static char args_doc[] = "";
00658
00659 static struct argp_option options[] = {
00660         {"source", 's', "SOURCE", 0, "Source file path", -1},
00661         {"destination", 'd', "DESTINATION", 0, "Destination file path", -1},
00662         {"password", 'p', "PASSWORD", 0, "Password", -1},
00663         {NULL}
00664 };
00665
00666 struct arguments {
00667     char *source;
00668     char *destination;
00669     char *password;
00670 };
00671
00672 static error_t parse_opt(int key, char *arg, struct argp_state *state) {
00673     struct arguments *arguments = state->input;
00674
00675     switch (key) {
00676         case 's':
00677             arguments->source = arg;
00678             break;
00679         case 'd':
00680             arguments->destination = arg;
00681             break;
00682         case 'p':
00683             arguments->password = arg;
00684             break;
00685         case ARGP_KEY_ARG:
00686             return ARGP_ERR_UNKNOWN;
00687         default:
00688             return ARGP_ERR_UNKNOWN;
00689     }
00690
00691     return 0;
00692 }
00693
00694 static struct argp argp = {options, parse_opt, args_doc, doc, 0, NULL, NULL};
00695
00696 int main(int argc, char *argv[])
00697 {
00698     umask(0);
00699
00700     struct arguments arguments;
00701
00702     arguments.source = NULL;
00703     arguments.destination = NULL;
00704     arguments.password = NULL;
00705
00706     argp_parse(&argp, argc, argv, 0, 0, &arguments);
00707
00708     if (arguments.source == NULL || arguments.destination == NULL || arguments.password == NULL) {
00709         printf("Err: You need to specify at least 3 arguments\n");
00710         return -1;
00711     }
00712
00713     printf("Source: %s\n", arguments.source);
00714     printf("Destination: %s\n", arguments.destination);
00715     root_path = arguments.source;
00716
00717     if (is_valid_key((unsigned char *)arguments.password) == 0){
00718         fprintf(stderr, "Inserted key not valid\n");
00719         return 1;
00720     }
00721
00722     struct fuse_args args_fuse = FUSE_ARGS_INIT(0, NULL);
00723     fuse_opt_add_arg(&args_fuse, "./tcfs");
00724     fuse_opt_add_arg(&args_fuse, arguments.destination);
00725     fuse_opt_add_arg(&args_fuse, "-f"); //TODO: this is forced for now, but will be passed via options
      in the future
00726     fuse_opt_add_arg(&args_fuse, "-s"); //TODO: this is forced for now, but will be passed via options
      in the future
00727
00728     //Print what we are passing to fuse TODO: This will be removed
00729     for (int i=0; i < args_fuse.argc; i++) {
00730         printf("%s ", args_fuse.argv[i]);
```

```
00731     }
00732     printf("\n");
00733 
00734     //Get username
00735     /*
00736     char buf[1024];
00737     size_t buf_size = 1024;
00738     get_user_name(buf, buf_size);
00739     */
00740 
00741     password = arguments.password;
00742 
00743     return fuse_main(args_fuse.argc, args_fuse.argv, &tcfs_oper, NULL);
00744 }
```

6.25 crypt-utils.c

```
00001 /*
00002  * High level function interface for performing AES encryption on FILE pointers
00003  * Uses OpenSSL libcrypto EVP API
00004  *
00005  * By Andy Sayler (www.andysayler.com)
00006  * Created  04/17/12
00007  * Modified 18/10/23 by [Carlo Alberto Giordano]
00008  *
00009  * Derived from OpenSSL.org EVP_Encrypt_* Manpage Examples
00010  * http://www.openssl.org/docs/crypto/EVP_EncryptInit.html#EXAMPLES
00011  *
00012  * With additional information from Saju Pillai's OpenSSL AES Example
00013  * http://saju.net.in/blog/?p=36
00014  * http://saju.net.in/code/misc/openssl_aes.c.txt
00015  *
00016  */
00017 #include "crypt-utils.h"
00018 
00019 #define BLOCKSIZE 1024
00020 #define IV_SIZE 32
00021 #define KEY_SIZE 32
00022 
00023 extern int do_crypt(FILE* in, FILE* out, int action, unsigned char *key_str){
00024     /* Local Vars */
00025 
00026     /* Buffers */
00027     unsigned char inbuf[BLOCKSIZE];
00028     int inlen;
00029     /* Allow enough space in output buffer for additional cipher block */
00030     unsigned char outbuf[BLOCKSIZE + EVP_MAX_BLOCK_LENGTH];
00031     int outlen;
00032     int writelen;
00033 
00034     /* OpenSSL libcrypto vars */
00035     EVP_CIPHER_CTX *ctx;
00036     ctx = EVP_CIPHER_CTX_new();
00037 
00038     unsigned char key[KEY_SIZE];
00039     unsigned char iv[IV_SIZE];
00040     int nrounds = 5;
00041 
00042     /* tmp vars */
00043     int i;
00044     /* Setup Encryption Key and Cipher Engine if in cipher mode */
00045     if(action >= 0){
00046         if(!key_str){
00047             /* Error */
00048             fprintf(stderr, "Key_str must not be NULL\n");
00049             return 0;
00050         }
00051         /* Build Key from String */
00052         i = EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(), NULL,
00053                            key_str, (int)strlen((const char *)key_str), nrounds, key, iv);
00054         if (i != 32) {
00055             /* Error */
00056             fprintf(stderr, "Key size is %d bits - should be 256 bits\n", i*8);
00057             return 0;
00058         }
00059         /* Init Engine */
00060         EVP_CIPHER_CTX_init(ctx);
00061         EVP_CipherInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv, action);
00062     }
00063 
00064     /* Loop through Input File*/
00065     for(;;){
00066         /* Read Block */
```

```
00067         inlen = fread(inbuf, sizeof(*inbuf), BLOCKSIZE, in);
00068         if(inlen <= 0){
00069             /* EOF -> Break Loop */
00070             break;
00071         }
00072 
00073         /* If in cipher mode, perform cipher transform on block */
00074         if(action >= 0){
00075             if(!EVP_CipherUpdate(ctx, outbuf, &outlen, inbuf, inlen))
00076             {
00077                 /* Error */
00078                 EVP_CIPHER_CTX_cleanup(ctx);
00079                 return 0;
00080             }
00081         }
00082             /* If in pass-through mode. copy block as is */
00083         else{
00084             memcpy(outbuf, inbuf, inlen);
00085             outlen = inlen;
00086         }
00087 
00088         /* Write Block */
00089         writelen = fwrite(outbuf, sizeof(*outbuf), outlen, out);
00090         if(writelen != outlen){
00091             /* Error */
00092             perror("fwrite error");
00093             EVP_CIPHER_CTX_cleanup(ctx);
00094             return 0;
00095         }
00096     }
00097 
00098     /* If in cipher mode, handle necessary padding */
00099     if(action >= 0){
00100         /* Handle remaining cipher block + padding */
00101         if(!EVP_CipherFinal_ex(ctx, outbuf, &outlen))
00102         {
00103             /* Error */
00104             EVP_CIPHER_CTX_cleanup(ctx);
00105             return 0;
00106         }
00107         /* Write remainign cipher block + padding*/
00108         fwrite(outbuf, sizeof(*inbuf), outlen, out);
00109         EVP_CIPHER_CTX_cleanup(ctx);
00110     }
00111 
00112     /* Success */
00113     return 1;
00114 }
00115 
00116 // Verify the entropy
00117 int check_entropy(void) {
00118     FILE *entropy_file = fopen("/proc/sys/kernel/random/entropy_avail", "r");
00119     if (entropy_file == NULL) {
00120         perror("Err: Cannot open entropy file");
00121         return -1;
00122     }
00123 
00124     int entropy_value;
00125     if (fscanf(entropy_file, "%d", &entropy_value) != 1) {
00126         perror("Err: Cannot estimate entropy");
00127         fclose(entropy_file);
00128         return -1;
00129     }
00130 
00131     fclose(entropy_file);
00132     return entropy_value;
00133 }
00134 
00135 //Add new entropy
00136 void add_entropy(void) {
00137     FILE *urandom = fopen("/dev/urandom", "rb");
00138     if (urandom == NULL) {
00139         perror("Err: Cannot open /dev/urandom");
00140         exit(EXIT_FAILURE);
00141     }
00142 
00143     unsigned char random_data[32];
00144     size_t bytes_read = fread(random_data, 1, sizeof(random_data), urandom);
00145     fclose(urandom);
00146 
00147     if (bytes_read != sizeof(random_data)) {
00148         fprintf(stderr, "Err: Cannot read data\n");
00149         exit(EXIT_FAILURE);
00150     }
00151 
00152     // Usa i dati casuali per aggiungere entropia
00153     RAND_add(random_data, sizeof(random_data), 0.5); // 0.5 è un peso arbitrario
```

```
00154
00155     fprintf(stdout, "Entropy added successfully!\n");
00156 }
00157
00158
00159 void generate_key(unsigned char *destination) {
00160     fprintf(stdout, "Generating a new key...\n");
00161
00162     //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
00163     for (int i = 0; i < 10; i++) {
00164         int entropy = check_entropy();
00165         if (entropy < 128) {
00166             fprintf(stderr, "WARN: not enough entropy, creating some...\n");
00167             add_entropy();
00168         }
00169
00170         if (RAND_bytes(destination, 32) != 1) {
00171             fprintf(stderr, "Err: Cannot generate key\n");
00172             destination = NULL;
00173         }
00174
00175         if (strlen((const char *)destination) == 32)
00176             break;
00177     }
00178
00179     if (is_valid_key(destination) == 0) {
00180         fprintf(stderr, "Err: Generated key is inval1d\n");
00181         print_aes_key(destination);
00182         destination = NULL;
00183     }
00184 }
00185
00186 unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int *encrypted_key_len) {
00187     EVP_CIPHER_CTX* ctx;
00188     const EVP_CIPHER* cipher = EVP_aes_256_cbc();
00189     unsigned char iv[AES_BLOCK_SIZE];
00190     memset(iv, 0, AES_BLOCK_SIZE);
00191
00192     ctx = EVP_CIPHER_CTX_new();
00193     if (!ctx) {
00194         return NULL;
00195     }
00196
00197     EVP_EncryptInit_ex(ctx, cipher, NULL, (const unsigned char*)key, iv);
00198
00199     size_t plaintext_len = strlen((const char*)plaintext);
00200     unsigned char ciphertext[plaintext_len + AES_BLOCK_SIZE];
00201     memset(ciphertext, 0, sizeof(ciphertext));
00202
00203     int len;
00204     EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
00205     EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
00206     EVP_CIPHER_CTX_free(ctx);
00207
00208     unsigned char* encoded_string = malloc(len * 2 + 1);
00209     if (!encoded_string) {
00210         return NULL;
00211     }
00212
00213     for (int i = 0; i < len; i++) {
00214         sprintf((char*)&encoded_string[i * 2], "%02x", ciphertext[i]);
00215     }
00216     encoded_string[len * 2] = '\0';
00217
00218     *encrypted_key_len = len * 2;
00219     return encoded_string;
00220 }
00221
00222
00223 unsigned char* decrypt_string(unsigned char* ciphertext, const char* key) {
00224     EVP_CIPHER_CTX *ctx;
00225     const EVP_CIPHER *cipher = EVP_aes_256_cbc(); // Choose the correct algorithm
00226     unsigned char iv[AES_BLOCK_SIZE];
00227     memset(iv, 0, AES_BLOCK_SIZE);
00228
00229     ctx = EVP_CIPHER_CTX_new();
00230     EVP_DecryptInit_ex(ctx, cipher, NULL, (const unsigned char*)key, iv);
00231
00232     size_t decoded_len = strlen((const char *)ciphertext);
00233
00234     unsigned char plaintext[decoded_len];
00235     memset(plaintext, 0, sizeof(plaintext));
00236
00237     int len;
00238     EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, (int )decoded_len);
00239     EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
```

```
00240     EVP_CIPHER_CTX_free(ctx);
00241
00242     unsigned char* decrypted_string = (unsigned char*)malloc(decoded_len + 1);
00243     memcpy(decrypted_string, plaintext, decoded_len);
00244     decrypted_string[decoded_len] = '\0';
00245
00246
00247     return decrypted_string;
00248 }
00249
00250 int is_valid_key(const unsigned char* key)
00251 {
00252     char str[33];
00253     memcpy(str, key, 32);
00254     str[32] = '\0';
00255     size_t key_length = strlen(str);
00256     return key_length != 32 ? 0:1;
00257 }
00258
00259
00260 /*
00261 int rebuild_key(char *key, char *cert, char *dest){
00262     return -1;
00263 }*/
```

6.26 crypt-utils.h

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <unistd.h>
00005 #include <sys/mman.h>
00006
00007 #include <openssl/evp.h>
00008 #include <openssl/aes.h>
00009 #include <openssl/rand.h>
00010 #include <openssl/bio.h>
00011 #include <openssl/buffer.h>
00012
00013 #include "../tcfs_utils/tcfs_utils.h" //TODO: Remove, for debugging only
00014
00015 #define BLOCKSIZE 1024
00016 #define ENCRYPT 1
00017 #define DECRYPT 0
00018
00019 /* int do_crypt(FILE* in, FILE* out, int action, char* key_str)
00020  * Purpose: Perform cipher on in File* and place result in out File*
00021  * Args: FILE* in      : Input File Pointer
00022  *       FILE* out     : Output File Pointer
00023  *       int action    : Cipher action (1=encrypt, 0=decrypt, -1=pass-through (copy))
00024  *   unsigned char *key_str : C-string containing passphrase from which key is derived
00025  * Return: 0 on error, 1 on success
00026  */
00027 extern int do_crypt(FILE* in, FILE* out, int action, unsigned char *key_str);
00028
00029 /* void generate_key(unsigned char *destination)
00030  * Purpose: Generate an AES 256 key of size 32 bytes
00031  * Args: unsigned char *destination    : The destination for the generated key. it must be 33 bytes
      long to account for a \0
00032  * Return: void, if the generation failed an error will be thrown
00033  */
00034 void generate_key(unsigned char *destination);
00035
00036 /*unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int *encrypted_len)
00037  * Purpose: Encrypt a string with AES-256
00038  * Args: unsigned char* plaintext   : The plaintext to be encrypted
00039  *       const char* key            : The key for the encryption
00040  *       int *encrypted_len         : This will be filled with the encrypted text length
00041  * Return: The encrypted string + \0. On error null is returned
00042  * */
00043 unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int *encrypted_len);
00044
00045 /*unsigned char* decrypt_string(unsigned char* base64_ciphertext, const char* key);
00046  * Purpose: Decrypt a string with AES-256
00047  * Args: unsigned char* base64_ciphertext   : The cyphertext to be decrypted
00048  *       const char* key            : The key for the decryption
00049  * Return: The decrypted string + \0. On error null is returned
00050  * */
00051 unsigned char* decrypt_string(unsigned char* base64_ciphertext, const char* key);
00052
00053 /*int is_valid_key(const unsigned char* key);
00054  * Purpose: Check if a AES-256 key is valid
00055  * Args: unsigned char* key   : The key to be checked
```

```
00056  * Return: 1 if the key is valid, 0 if it is invalid
00057  * */
00058 int is_valid_key(const unsigned char* key);
00059
00060 /*
00061 int rebuild_key(char *key, char *cert, char *dest);
00062 */
```

6.27 password_manager.c

```
00001 //TODO: This util will handle requesting keys to kernel
00002
00003 #include "password_manager.h"
00004 #include "../crypt-utils/crypt-utils.h"
00005 /*
00006 char *true_key;
00007
00008 int insert_key(char* key, char* cert, int is_sys_call)
00009 {
00010     if (is_sys_call == WITH_SYS_CALL)
00011     {
00012         fprintf(stderr, "The kernal module has not been implemented yet, saving key in userspace\n \
00013                          This will change in the future");
00014         insert_key(key, cert, WITHOUT_SYS_CALL);
00015     }
00016     return rebuild_key(key, cert, true_key);
00017 }
00018
00019 char *request_key(int is_sys_call){
00020     return NULL;
00021 }
00022 int delete_key(int is_sys_call){
00023     return -1;
00024 }*/
```

6.28 password_manager.h

```
00001 #include <stddef.h>
00002 #include <stdio.h>
00003
00004 #define WITH_SYS_CALL 1
00005 #define WITHOUT_SYS_CALL 0
00006 /*
00007 int insert_key(char* key, char* cert, int is_sys_call);
00008 char *request_key(int is_sys_call);
00009 int delete_key(int is_sys_call);*/
```

6.29 tcfs_utils.c

```
00001 #include "tcfs_utils.h"
00002 #include "../crypt-utils/crypt-utils.h"
00003
00004 void get_user_name(char *buf, size_t size)
00005 {
00006     uid_t uid = geteuid();
00007     struct passwd *pw = getpwuid(uid);
00008     if (pw)
00009         snprintf(buf, size, "%s", pw->pw_name);
00010     else
00011         perror("Error: Could not retrieve username.\n");
00012 }
00013
00014 /* is_encrypted: returns 1 if file is encrypted, 0 otherwise*/
00015 int is_encrypted(const char *path)
00016 {
00017     int ret;
00018     char xattr_val[5];
00019     getxattr(path, "user.encrypted", xattr_val, sizeof(char)*5);
00020     xattr_val[4] == '\n';
00021
00022     return strcmp(xattr_val, "true") == 0 ? 1:0;
00023 }
00024
00025 char *prefix_path(const char *path, const char *realpath)
00026 {
```

```
00027     if (path == NULL || realpath == NULL)
00028     {
00029         perror("Err: path or realpath is NULL");
00030         return NULL;
00031     }
00032
00033     size_t len = strlen(path) + strlen(realpath) + 1;
00034     char *root_dir = malloc(len * sizeof(char));
00035
00036     if (root_dir == NULL)
00037     {
00038         perror("Err: Could not allocate memory while in prefix_path");
00039         return NULL;
00040     }
00041
00042     if (strcpy(root_dir, realpath) == NULL)
00043     {
00044         perror("strcpy: Cannot copy path");
00045         return NULL;
00046     }
00047     if (strcat(root_dir, path) == NULL)
00048     {
00049         perror("strcat: in prefix_path cannot concatenate the paths");
00050         return NULL;
00051     }
00052     return root_dir;
00053 }
00054
00055 /* read_file: for debugging tempfiles */
00056 int read_file(FILE *file)
00057 {
00058     int c;
00059     int file_contains_something = 0;
00060     FILE *read = file; /* don't move original file pointer */
00061     if (read) {
00062         while ((c = getc(read)) != EOF) {
00063             file_contains_something = 1;
00064             putc(c, stderr);
00065         }
00066     }
00067     if (!file_contains_something)
00068         fprintf(stderr, "file was empty\n");
00069     rewind(file);
00070     /* fseek(tmpf, offset, SEEK_END); */
00071     return 0;
00072 }
00073 /* Get the xattr value describing the key of a file
00074  * return 1 on success else 0
00075  * */
00076 int get_encrypted_key(char *filepath, unsigned char *encrypted_key)
00077 {
00078     printf("\tGet Encrypted key for file %s\n", filepath);
00079     if (is_encrypted(filepath) == 1) {
00080         printf("\t\tencrypted file\n");
00081
00082         FILE *src_file = fopen(filepath, "r");
00083         if (src_file == NULL)
00084         {
00085             fclose(src_file);
00086             perror("Could not open the file to get the key");
00087             return -errno;
00088         }
00089         int src_fd;
00090         src_fd = fileno(src_file);
00091         if (src_fd == -1)
00092         {
00093             fclose(src_file);
00094             perror("Could not get fd for the file");
00095             return -errno;
00096         }
00097
00098         if (fgetxattr(src_fd, "user.key", encrypted_key,  33) != -1) {
00099             fclose(src_file);
00100             return 1;
00101         }
00102     }
00103     return 0;
00104 }
00105 /*For debugging only*/
00106 void print_aes_key(unsigned char *key) {
00107     printf("AES HEX:%s -> ", key);
00108     for (int i = 0; i < 32; i++) {
00109         printf("%02x", key[i]);
00110     }
00111     printf("\n");
00112 }
```

6.30 tcfs_utils.h

```
00001 #include <string.h>
00002 #include <stdio.h>
00003 #include <pwd.h>
00004 #include <unistd.h>
00005 #include <sys/xattr.h>
00006 #include <stdlib.h>
00007 #include <errno.h>
00008
00009 /* void get_user_name(char *buf, size_t size)
00010  * Purpose: Fetch the username of the current user
00011  * Args: char *buf      : The username will be written to this buffer
00012  *       size_t size    : The size of the buffer;
00013  * Return: Nothing
00014  */
00015 void get_user_name(char *buf, size_t size);
00016
00017 /* is_encrypted: returns 1 if encryption succeeded, 0 otherwise. There is currently no use for this
      function */
00018 int is_encrypted(const char *path);
00019
00020 /* char *prefix_path(const char *path))
00021  * Purpose: Prefix the realpath to the fuse path
00022  * Args: char *path      : The fuse path
00023  *       char *realpath  : The realpath
00024  * Return: NULL on error, char* on success
00025  */
00026 char *prefix_path(const char *path, const char *realpath);
00027
00028 /* read_file: for debugging tempfiles */
00029 int read_file(FILE *file);
00030
00031 /* int get_encrypted_key(char *filepath, void *encrypted_key)
00032  * Purpose: Get the encrypted file key from its xattrs
00033  * Args: char *filepath      : The full-path of the file
00034  *       char *encrypted_key  : The buffer to save the encrypted key to
00035  * Return: 0 on error, 1 on success
00036  */
00037 int get_encrypted_key(char *filepath, unsigned char *encrypted_key);
00038
00039 /*For debugging only*/
00040 void print_aes_key(unsigned char *key);
```

Index