TCFS

0.2

Generated on Tue Nov 28 2023 11:35:48 for TCFS by Doxygen 1.9.8

Tue Nov 28 2023 11:35:48

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 Documentation

Documentation is lacking but it can be found here

1.3.2 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.

1.3.3 Build

- Clone the TCFS repository to your local machine:

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

```
##
```

1.4 Build and run the userpace module

- Compile: Run the Makefile in the userspace-module directory

```
make all
```

- Run: Run the compiled file. NOTE: Password must be 256 bit or 32 bytes

```
build/fuse-module/tcfs -s "source_dir" -d "dest_dir" -p "password"
```

#

1.4.1 Build and run the daemon

- Build and install: To install the daemon run this commands in the tcfs_daemon directory

```
make; make install
```

#

1.4.2 Build and run the helper program

- Compile: Run the Makefile in the user directory

```
make
```

- Run: Run the compiled file

```
build/tcfs_helper/tcfs_helper
```

#

1.4.3 Kernel module

- This part of the project is not being developed at the moment.

1.5 Usage of the fuse module

1.5.1 This is not raccomended, consider using the tcfs_helper program

1.5.2 Mount an NFS share using TCFS:

First, mount the NFS share to a directory, 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.

1.5.3 Unmount the NFS share when you're done:

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

then unmount the NFS share.

1.6 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.

1.7 License

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

1.8 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.

1.9 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.
  - Switch to public/private key
- Implement threshold sharing files.
- Daemon:
  - ~~Implement user registration and deregistration~~
  - Implement accessing and creation of shared files
  - Update the userspace module to handle the features that the daemon provides

Chapter 2

Todo List

Member handle_termination (int signum)

: Implement remove_queue() to clear and delete the queue

Member init_queue (char *queue)

Define permissions for mq_open

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

Member PORT

This should be passed as a parameter to the daemon

Struct qm_shared

Handle creation of shared files and not only accessing them. This mey imply a new field

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

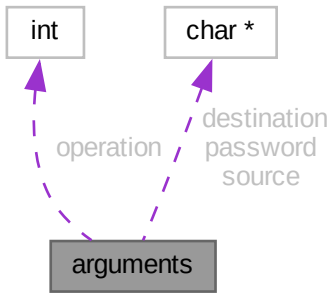
Here is a list of all documented files with brief descriptions:

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 725 of file tcfs.c.

5.1.2.2 operation

```
int arguments::operation
```

Definition at line 22 of file user_tcfs.c.

5.1.2.3 password

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

Definition at line 726 of file tcfs.c.

5.1.2.4 source

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

Definition at line 724 of file tcfs.c.

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

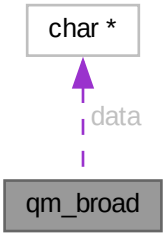
- user/user_tcfs.c
- userspace-module/tcfs.c

5.2 qm_broad Struct Reference

Represents a broadcast message. Contains the data that is broadcasted to all users.

```
#include <common.h>
```

Collaboration diagram for qm_broad:



Public Attributes

- char * data

5.2.1 Detailed Description

Represents a broadcast message. Contains the data that is broadcasted to all users.

Definition at line 81 of file common.h.

5.2.2 Member Data Documentation

5.2.2.1 data

```
char* qm_broad::data
```

The data that is broadcasted.

Definition at line 82 of file common.h.

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

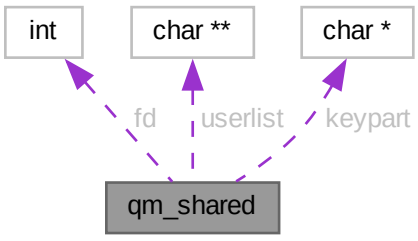
- daemon/daemon_utils/common.h

5.3 qm_shared Struct Reference

Represents a shared message.
Contains information about the file descriptor ti which the TCFS module wants to access, the user list to ask for keyparts and the key part of the caller.
.

```
#include <common.h>
```

Collaboration diagram for qm_shared:



Public Attributes

- int fd
- char ** userlist
- char * keypart

5.3.1 Detailed Description

Represents a shared message.
Contains information about the file descriptor ti which the TCFS module wants to access,
the user list to ask for keyparts and the key part of the caller.
.

Todo Handle creation of shared files and not only accessing them. This mey imply a new field

Definition at line 70 of file common.h.

5.3.2 Member Data Documentation

5.3.2.1 fd

```
int qm_shared::fd
```

The file descriptor of the shared file.

Definition at line 71 of file common.h.

5.3.2.2 keypart

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

The part of the key given by the caller that is needed to decrypt the shared file.

Definition at line 73 of file common.h.

5.3.2.3 userlist

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

The list of users who created the shared file.

Note

This is really a matrix of chars

Definition at line 72 of file common.h.

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

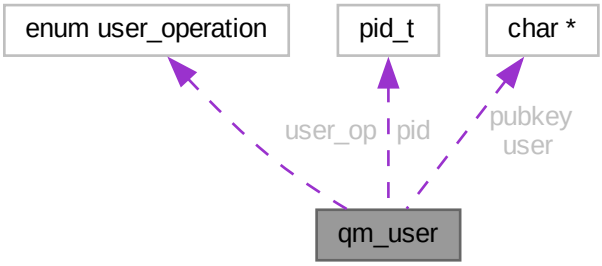
- daemon/daemon_utils/common.h

5.4 qm_user Struct Reference

Represents a user message.
Contains information about the user's operation, process ID, username and public key.
.

`#include <common.h>`

Collaboration diagram for qm_user:



Public Attributes

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

5.4.1 Detailed Description

Represents a user message.
Contains information about the user's operation, process ID, username and public key.
.

Definition at line 56 of file common.h.

5.4.2 Member Data Documentation

5.4.2.1 pid

`pid_t qm_user::pid`

The process ID of the user.

Definition at line 58 of file common.h.

Referenced by get_user_by_name(), get_user_by_pid(), insert(), and remove_by_user().

5.4.2.2 pubkey

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

The public key of the user.

Definition at line 60 of file common.h.

5.4.2.3 user

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

The username of the user.

Definition at line 59 of file common.h.

Referenced by get_user_by_name(), get_user_by_pid(), insert(), and remove_by_pid().

5.4.2.4 user_op

```
user_operation qm_user::user_op
```

The operation that the user wants to perform.

Definition at line 57 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
00014 #define MAX_QM_SIZE 512
00020 #define MAX_QM_N 100
00021
00022 #ifndef QUEUE_STRUCTS
00023 #define QUEUE_STRUCTS
00024
00033 typedef enum qm_type{
00034     USER = 0,
00035     SHARED = 1,
00036     BROADCAST = 2,
00037     QM_TYPE_UNDEFINED = -1,
00038 } qm_type;
00039
00046 typedef enum user_operation{
00047     REGISTER = 0,
00048     UNREGISTER = 1,
00049 } user_operation;
00050
00056 typedef struct qm_user {
00057     user_operation user_op;
00058     pid_t pid;
00059     char *user;
00060     char *pubkey;
00061 } qm_user;
00062
00070 typedef struct qm_shared {
00071     int fd;
00072     char **userlist;
00073     char *keypart;
00074 } qm_shared;
00075
00081 typedef struct qm_broad {
00082     char *data;
00083 } qm_broad;
00084
00085 #endif
```

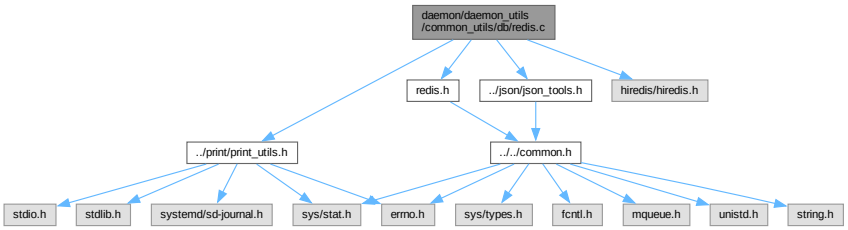
6.2 daemon/daemon_utils/common_utils/db/redis.c File Reference

All the function in this file should not be used directly, instead use the function defined by user_db.

```
#include "redis.h"
#include "../json/json_tools.h"
```

```
#include "../print/print_utils.h"
#include <hiredis/hiredis.h>
```

Include dependency graph for redis.c:



Macros

- #define PORT 6380

  The port of the redis DB.

Functions

- void print_all_keys ()

  For debugging only. Prints all the keys in the database.
- int init_context ()

  initialize the context for the Redis DB
- void free_context ()

  Free the hiredis context variable.
- qm_user ∗ json_to_qm_user (char ∗json)

  Internal function to simplify the casting of a json to a qm_user struct.
- qm_user ∗ get_user_by_pid (pid_t pid)

  Fetch the user on the DB with key pid.
- qm_user ∗ get_user_by_name (const char ∗name)

  Fetch the user on the DB with key name.
- int insert (qm_user ∗user)

  Insert a new user in the DB.
- int remove_by_pid (pid_t pid)

  Remove a user from the DB using the PID as key.
- int remove_by_user (char ∗name)

  Remove a user from the DB using the name as key.

Variables

- const char HOST [ ] = "127.0.0.1"
- redisContext ∗ context

  Pointer to the context of Redis DB.

6.2.1 Detailed Description

All the function in this file should not be used directly, instead use the function defined by user_db.

This file is marked as internal and the corresponding header should not be used by the user. Please refer to the see section

See also

\ref user_db.c

Definition in file redis.c.

6.2.2 Macro Definition Documentation

6.2.2.1 PORT

```
#define PORT 6380
```

The port of the redis DB.

This definition is marked as internal and be used directly

Todo This should be passed as a parameter to the daemon

Definition at line 27 of file redis.c.

6.2.3 Function Documentation

6.2.3.1 free_context()

```
void free_context ( )
```

Free the hiredis context variable.

This function is marked as internal and should not be used by the user

Returns

void

Definition at line 92 of file redis.c.

References context.

Referenced by disconnect_db().

Here is the caller graph for this function:

6.2.3.2 get_user_by_name()

```
qm_user * get_user_by_name (
            const char * name )
```

Fetch the user on the DB with key name.

This function is marked as internal and should not be used by the user

Parameters

const	char *name The key of the row

Returns

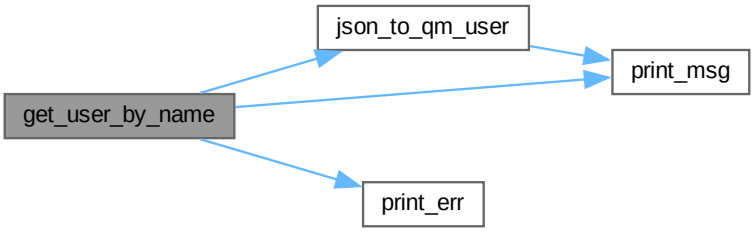
qm_user* A pointer to the allocated qm_user* struct

Definition at line 165 of file redis.c.

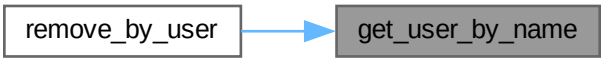
References context, json_to_qm_user(), qm_user::pid, print_err(), print_msg(), and qm_user::user.

Referenced by remove_by_user().

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.3.3 get_user_by_pid()

```
qm_user * get_user_by_pid (
            pid_t pid )
```

Fetch the user on the DB with key pid.

This function is marked as internal and should not be used by the user

Parameters

pid_t	pid The key of the row

Returns

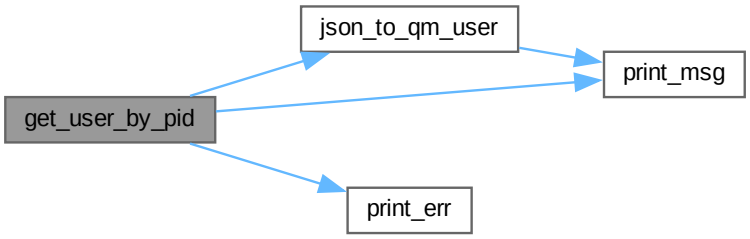
qm_user* A pointer to the allocated qm_user* struct

Definition at line 122 of file redis.c.

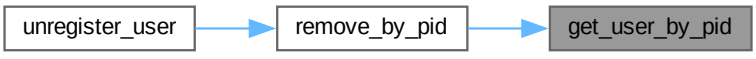
References context, json_to_qm_user(), qm_user::pid, print_err(), print_msg(), and qm_user::user.

Referenced by remove_by_pid().

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.3.4 init_context()

```
int init_context ( )
```

initialize the context for the Redis DB

This function is marked as internal and should not be used by the user

Returns

1 if initialization was successful or the database was already initialized, 0 on failure

Definition at line 72 of file redis.c.

References context, PORT, and print_err().

Referenced by register_user().

Here is the call graph for this function:

Here is the caller graph for this function:

6.2.3.5 insert()

```
int insert (
            qm_user * user )
```

Insert a new user in the DB.

This function is marked as internal and should not be used by the user

Parameters

*qm_user**	A pointer to the allocated qm_user* struct

Returns

1 if successful, 0 otherwise. An error might be printen by print_err() function,

See also

print_err

Note

The user will be set 2 times, once with key user->pid and once with key user->name

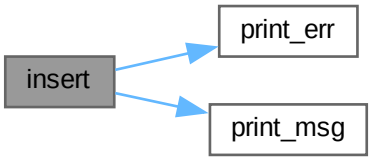
If an error is thrown it will be printed by print_err() function

Definition at line 211 of file redis.c.

References context, qm_user::pid, print_err(), print_msg(), and qm_user::user.

Referenced by register_user().

Here is the call graph for this function:



Here is the caller graph for this function:

6.2.3.6 json_to_qm_user()

```
qm_user * json_to_qm_user (
            char * json )
```

Internal function to simplify the casting of a json to a qm_user struct.

This function is marked as internal and should not be used by the user

Parameters

char	∗json the json string representing the qm_user struct

Returns

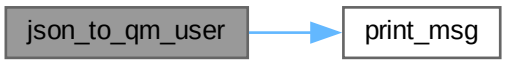
qm_user∗ A pointer to the allocated qm_user∗ struct

Definition at line 104 of file redis.c.

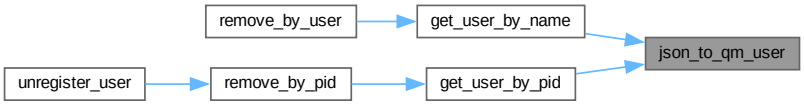
References print_msg().

Referenced by get_user_by_name(), and get_user_by_pid().

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.3.7 print_all_keys()

```
void print_all_keys ( )
```

For debugging only. Prints all the keys in the database.

This function is marked as internal and should not be used by the user

Returns

void

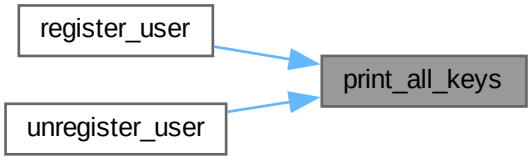
Definition at line 42 of file redis.c.

References context, and print_msg().

Referenced by register_user(), and unregister_user().

Here is the call graph for this function:

Here is the caller graph for this function:



6.2.3.8 remove_by_pid()

```
int remove_by_pid (
            pid_t pid )
```

Remove a user from the DB using the PID as key.

This function is marked as internal and should not be used by the user

Parameters

pid_t	pid The key

Returns

1 if successful, 0 otherwise. An error might be printen by print_err() function,

See also

print_err

Note

Will also remove the corresponding entry by name.

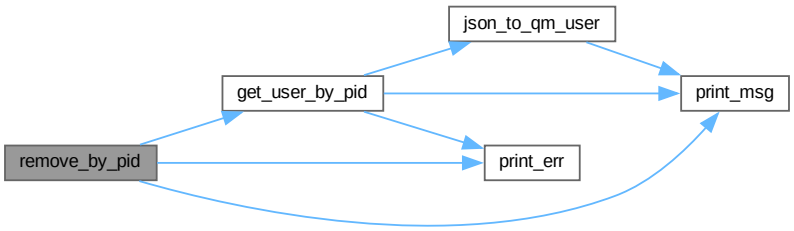
If an error is thrown it will be printed using the print_err() function

Definition at line 256 of file redis.c.

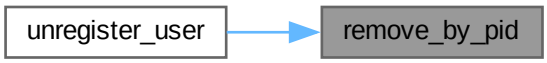
References context, get_user_by_pid(), print_err(), print_msg(), and qm_user::user.

Referenced by unregister_user().

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.3.9 remove_by_user()

```
int remove_by_user (
            char * name )
```

Remove a user from the DB using the name as key.

This function is marked as internal and should not be used by the user

Parameters

char	∗name The key

Returns

1 if successful, 0 otherwise. An error might be printen by print_err() function,

See also

print_err

Note

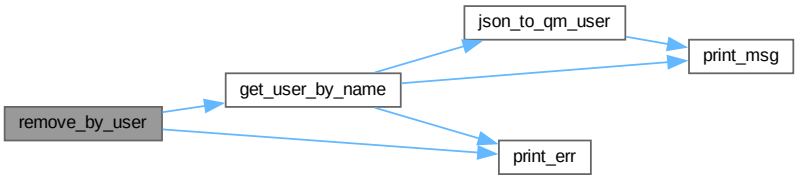
Will also remove the corresponding entry by PID

If an error is thrown it will be printed using the print_err() function

Definition at line 292 of file redis.c.

References context, get_user_by_name(), qm_user::pid, and print_err().

Here is the call graph for this function:



6.2.4 Variable Documentation

6.2.4.1 context

```
redisContext * context
```

Pointer to the context of Redis DB.

This variable is marked as internal and should not be used by the user

Definition at line 34 of file redis.c.

Referenced by free_context(), get_user_by_name(), get_user_by_pid(), init_context(), insert(), print_all_keys(), remove_by_pid(), and remove_by_user().

6.2.4.2 HOST

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

Definition at line 20 of file redis.c.

6.3 redis.c

Go to the documentation of this file.

```
00001
00009 #include "redis.h"
00010 #include "../json/json_tools.h"
00011 #include "../print/print_utils.h"
00012 #include <hiredis/hiredis.h>
00013
00020 const char HOST[] = "127.0.0.1";
00027 #define PORT 6380
00028
00034 redisContext *context;
00035
00041 void
00042 print_all_keys ()
00043 {
00044   redisReply *keys_reply = (redisReply *)redisCommand (context, "KEYS *");
00045   if (keys_reply)
00046     {
00047       if (keys_reply->type == REDIS_REPLY_ARRAY)
00048         {
00049           for (size_t i = 0; i < keys_reply->elements; ++i)
00050             {
00051               print_msg ("\tKey: %s", keys_reply->element[i]->str);
00052             }
00053         }
00054       else
00055         {
00056           print_msg ("Error retrieving keys: %s", keys_reply->str);
00057         }
00058       freeReplyObject (keys_reply);
00059     }
00060   else
00061     {
00062       print_msg ("Error executing KEYS command");
00063     }
00064 }
00071 int
00072 init_context ()
00073 {
00074   // Do not reinit the context
00075   if (context != NULL)
00076     return 1;
00077
00078   context = redisConnect (HOST, PORT);
00079   if (context->err)
00080     {
00081       print_err ("Connection error: %s", context->errstr);
00082       return 0;
00083     }
00084   return 1;
00085 }
00091 void
00092 free_context ()
00093 {
00094   redisFree (context);
00095 }
00103 qm_user *
00104 json_to_qm_user (char *json)
00105 {
00106   print_msg ("DEBUG: Converting %s", json);
00107   qm_type type;
00108   // Redis return the value as json:{actual json} so we need to eliminate the
00109   // json: from the string
00110   char *res = strchr (json, ':');
00111   res++; // Skip the : char
00112   qm_user *user = (qm_user *)string_to_struct (res, &type);
00113   return user;
00114 }
00121 qm_user *
00122 get_user_by_pid (pid_t pid)
```

```
00123 {
00124   qm_user *user = NULL;
00125   // Retrieve the JSON data from Redis hash
00126   print_msg ("EXECUTING \"GET pid:%d\"", pid);
00127   redisReply *luaReply
00128       = (redisReply *)redisCommand (context, "GET pid:%d", pid);
00129   if (luaReply)
00130     {
00131       if (luaReply->type == REDIS_REPLY_STRING)
00132         {
00133           user = json_to_qm_user (luaReply->str);
00134           if (user)
00135             {
00136               print_msg ("Successful retrieval! PID: %d, User: %s", user->pid,
00137                          user->user);
00138             }
00139           else
00140             {
00141               print_err ("Error converting JSON to struct");
00142             }
00143         }
00144       else
00145         {
00146           print_err ("Reply type error %d -> executing HGET\n\tErrString: %s",
00147                      luaReply->type, luaReply->str, context->errstr);
00148         }
00149       freeReplyObject (luaReply);
00150     }
00151   else
00152     {
00153       print_err ("Reply error executing HGET\n\tErrString: %s",
00154                  context->errstr);
00155     }
00156   return user;
00157 }
00164 qm_user *
00165 get_user_by_name (const char *name)
00166 {
00167   qm_user *user = NULL;
00168   // Retrieve the JSON data from Redis hash
00169   print_msg ("EXECUTING \"GET name:%d\"", name);
00170   redisReply *luaReply
00171       = (redisReply *)redisCommand (context, "GET name:%d", name);
00172   if (luaReply)
00173     {
00174       if (luaReply->type == REDIS_REPLY_STRING)
00175         {
00176           user = json_to_qm_user (luaReply->str);
00177           if (user)
00178             {
00179               print_msg ("Successful retrieval! PID: %d, User: %s", user->pid,
00180                          user->user);
00181             }
00182           else
00183             {
00184               print_err ("Error converting JSON to struct");
00185             }
00186         }
00187       else
00188         {
00189           print_err ("Reply type error %d -> executing HGET\n\tErrString: %s",
00190                      luaReply->type, luaReply->str, context->errstr);
00191         }
00192       freeReplyObject (luaReply);
00193     }
00194   else
00195     {
00196       print_err ("Reply error executing HGET\n\tErrString: %s",
00197                  context->errstr);
00198     }
00199   return user;
00200 }
00210 int
00211 insert (qm_user *user)
00212 {
00213   // Convert the structure to JSON
00214   const char *json = struct_to_json (USER, user);
00215   if (!json)
00216     {
00217       print_err ("Error converting qm_user to JSON");
00218       return 0;
00219     }
00220   // Save to Redis with key "pid_str"
00221   print_msg ("\tDB: \"SET pid:%d json:%s\"", user->pid, json);
00222   redisReply *reply_pid = (redisReply *)redisCommand (
00223       context, "SET pid:%d json:%s", user->pid, json);
00224   if (!reply_pid)
```

```
00225     {
00226       print_err ("Error saving to Redis (pid)");
00227       free ((void *)json);
00228       return 0;
00229     }
00230   freeReplyObject (reply_pid);
00231
00232   // Save to Redis with key "user"
00233   redisReply *reply_user = (redisReply *)redisCommand (
00234       context, "SET user:%s json:%s", user->user, json);
00235   if (!reply_user)
00236     {
00237       print_err ("Error saving to Redis (user)");
00238       free ((void *)json);
00239       return 0;
00240     }
00241   freeReplyObject (reply_user);
00242   // Free the allocated JSON memory
00243   free ((void *)json); // Discard qualifier
00244   return 1;
00245 }
00255 int
00256 remove_by_pid (pid_t pid)
00257 {
00258   qm_user *user_tmp = get_user_by_pid (pid);
00259   // Remove the structure by PID
00260   print_msg ("\tDB: \"DEL pid:%d\"", pid);
00261   redisReply *reply_pid
00262       = (redisReply *)redisCommand (context, "DEL pid:%d", pid);
00263   if (!reply_pid)
00264     {
00265       print_err ("Error removing structure by PID");
00266       return 0;
00267     }
00268   freeReplyObject (reply_pid);
00269   // Also remove the corresponding key by name
00270   print_msg ("\tDB: \"DEL user:%s\"", user_tmp->user);
00271   redisReply *reply_name
00272       = (redisReply *)redisCommand (context, "DEL user:%s", user_tmp->user);
00273   if (!reply_name)
00274     {
00275       print_err ("Error removing key by name");
00276       return 0;
00277     }
00278   free (user_tmp);
00279   freeReplyObject (reply_name);
00280   return 1;
00281 }
00291 int
00292 remove_by_user (char *name)
00293 {
00294   qm_user *user_tmp = get_user_by_name (name);
00295   // Remove the structure by name
00296   char key_name[64]; // Adjust the size as needed
00297   snprintf (key_name, sizeof (key_name), "user:%s", name);
00298   redisReply *reply_name
00299       = (redisReply *)redisCommand (context, "DEL %s", key_name);
00300   if (!reply_name)
00301     {
00302       print_err ("Error removing structure by name");
00303       return 0;
00304     }
00305   freeReplyObject (reply_name);
00306   // Also remove the corresponding key by PID
00307   redisReply *reply_pid
00308       = (redisReply *)redisCommand (context, "DEL %d", user_tmp->pid);
00309   if (!reply_pid)
00310     {
00311       print_err ("Error removing key by PID");
00312       return 0;
00313     }
00314   freeReplyObject (reply_pid);
00315   return 1;
00316 }
```

6.4 redis.h

```
00001 #include "../../common.h"
00002
00003 void print_all_keys ();
00004
00005 int init_context ();
00006
```

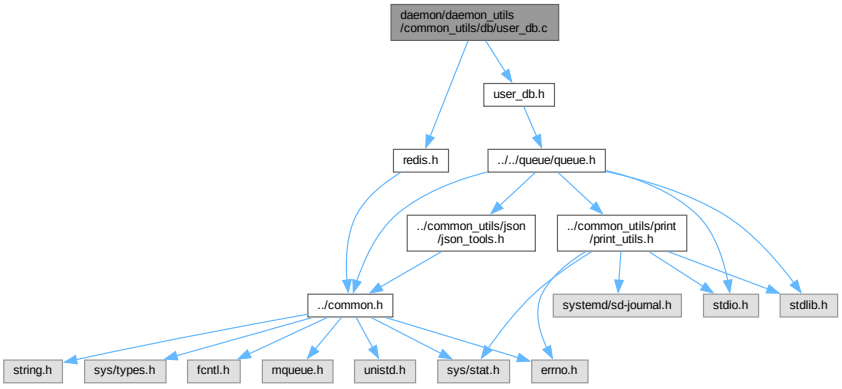
```
00007 qm_user *json_to_qm_user (char *json);
00008
00009 qm_user *get_user_by_pid (pid_t pid);
00010
00011 qm_user *get_user_by_name (const char *name);
00012
00013 int insert (qm_user *user);
00014
00015 int remove_by_pid (pid_t pid);
00016
00017 int remove_by_user (char *name);
00018
00019 void free_context ();
```

6.5 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.

6.5.1 Detailed Description

This file contains the functions to interact with the database.

Definition in file user_db.c.

6.5.2 Function Documentation

6.5.2.1 disconnect_db()

```
void disconnect_db (
            void  )
```

Free the context of the DB.

Parameters

void	

Returns

void

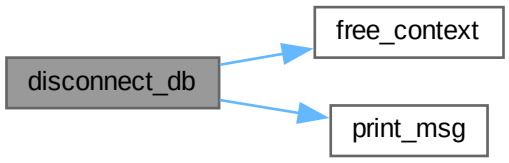
Note

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

Definition at line 45 of file user_db.c.

References free_context(), and print_msg().

Here is the call graph for this function:



6.5.2.2 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. An error might be printen by print_err() function,

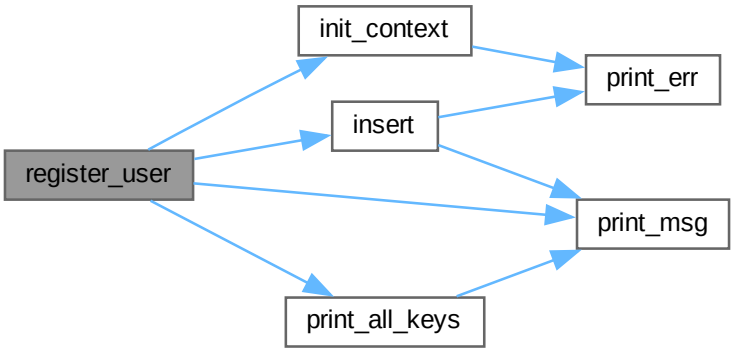
See also

print_err

Definition at line 15 of file user_db.c.

References init_context(), insert(), print_all_keys(), and print_msg().

Here is the call graph for this function:



6.5.2.3 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. An error might be printen by print_err() function,

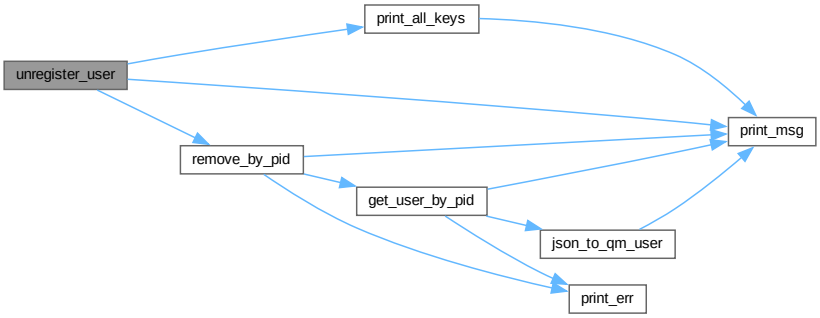
See also

print_err

Definition at line 31 of file user_db.c.

References print_all_keys(), print_msg(), and remove_by_pid().

Here is the call graph for this function:



6.6 user_db.c

Go to the documentation of this file.

```
00001 #include "user_db.h"
00002 #include "redis.h"
00003
00014 int
00015 register_user (qm_user *user_msg)
00016 {
00017   print_msg ("Registering new user");
00018   if (init_context () == 0)
00019     return 0;
00020   print_all_keys ();
00021   if (insert (user_msg) == 0)
00022     return 0;
00023   return 1;
00024 }
00030 int
00031 unregister_user (pid_t pid)
00032 {
00033   print_all_keys ();
00034   print_msg ("Removing user");
00035   return remove_by_pid (pid);
00036 }
00044 void
00045 disconnect_db (void)
00046 {
00047   print_msg ("Freeing context...");
00048   free_context ();
00049 }
```

6.7 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);
```

6.8 json_tools.cpp

```
00001 #include "../../common.h"
00002 #include "../print/print_utils.h"
00003 #include "/usr/include/nlohmann/json.hpp" // Assuming you're using nlohmann's JSON library
00004 #include <cstdlib>                       // For malloc and free
00005 #include <cstring>                       // For strcpy
00006 #include <iostream>
00007 #include <string.h>
00008 #include <vector>
00009
00026 char *
00027 struct_to_json (qm_type qmt, void *q_mess)
00028 {
00029   nlohmann::json json_obj;
00030
00031   switch (qmt)
00032     {
00033     case USER:
00034       {
00035         qm_user *user = static_cast<qm_user *> (q_mess);
00036         if (user->user_op == REGISTER)
00037           print_msg ("Register");
00038         if (user->user_op == UNREGISTER)
00039           print_msg ("Unregister");
00040         json_obj["user_op"] = user->user_op;
00041         json_obj["pid"] = user->pid;
00042         json_obj["user"] = user->user;
00043         json_obj["pubkey"] = user->pubkey;
00044         break;
00045       }
00046     case SHARED:
00047       {
00048         qm_shared *shared = static_cast<qm_shared *> (q_mess);
00049         json_obj["fd"] = shared->fd;
00050
00051         // Converti la matrice di stringhe in un array di stringhe JSON
00052         nlohmann::json userlist_array = nlohmann::json::array ();
00053         for (size_t i = 0; shared->userlist[i] != nullptr; ++i)
00054           {
00055             userlist_array.push_back (shared->userlist[i]);
00056           }
00057         json_obj["userlist"] = userlist_array;
00058
00059         json_obj["keypart"] = shared->keypart;
00060         break;
00061       }
00062     case BROADCAST:
00063       {
00064         qm_broad *broad = static_cast<qm_broad *> (q_mess);
00065         json_obj["data"] = broad->data;
00066         break;
00067       }
00068     }
00069   // Cast Json obj to string
00070   std::string json_str = json_obj.dump ();
00071   // Allocate memory for result
00072   char *result = (char *)malloc (json_str.size () + 1);
00073   if (result)
00074     {
00075       strcpy (result, json_str.c_str ());
00076     }
00077   print_msg ("JSONIFIED: %s", result);
00078   return result;
00079 }
00080
00091 void *
00092 string_to_struct (const char *json_string, qm_type *type)
00093 {
00094   try
00095     {
00096       nlohmann::json json_obj = nlohmann::json::parse (json_string);
00097
00098       if (json_obj.contains ("user_op"))
00099         {
00100           *type = USER;
00101           qm_user *user
00102               = static_cast<qm_user *> (std::malloc (sizeof (qm_user)));
00103           user->user_op = json_obj["user_op"];
00104           user->pid = json_obj["pid"];
00105           user->user = strdup (json_obj["user"].get<std::string> ().c_str ());
00106           user->pubkey
00107               = strdup (json_obj["pubkey"].get<std::string> ().c_str ());
00108           return user;
00109         }
00110       else if (json_obj.contains ("fd"))
00111         {
```

```
00112           *type = SHARED;
00113           qm_shared *shared
00114               = static_cast<qm_shared *> (std::malloc (sizeof (qm_shared)));
00115           shared->fd = json_obj["fd"];
00116
00117           // Populate userlist array
00118           std::vector<std::string> userlist = json_obj["userlist"];
00119           shared->userlist = static_cast<char **> (
00120               std::malloc ((userlist.size () + 1) * sizeof (char *)));
00121           for (size_t i = 0; i < userlist.size (); ++i)
00122             {
00123               shared->userlist[i] = strdup (userlist[i].c_str ());
00124             }
00125           shared->userlist[userlist.size ()] = nullptr;
00126
00127           shared->keypart
00128               = strdup (json_obj["keypart"].get<std::string> ().c_str ());
00129           return shared;
00130         }
00131       else if (json_obj.contains ("data"))
00132         {
00133           *type = BROADCAST;
00134           qm_broad *broad
00135               = static_cast<qm_broad *> (std::malloc (sizeof (qm_broad)));
00136           broad->data = strdup (json_obj["data"].get<std::string> ().c_str ());
00137           return broad;
00138         }
00139       else
00140         {
00141           *type = QM_TYPE_UNDEFINED;
00142           return nullptr;
00143         }
00144     }
00145   catch (const std::exception &e)
00146     {
00147       std::cerr « "Error parsing JSON: " « e.what () « std::endl;
00148       return nullptr;
00149     }
00150 }
```

6.9 json_tools.h

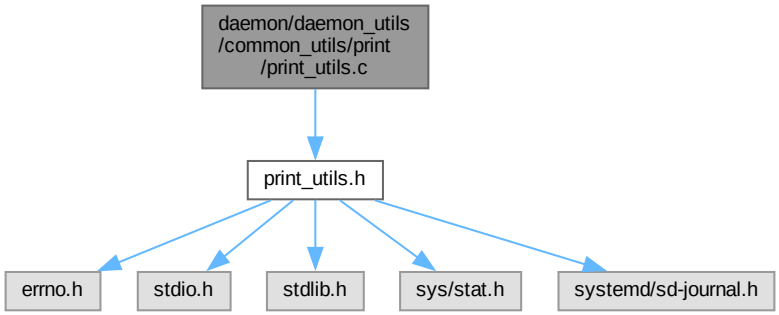
```
00001 #include "../../common.h"
00002
00003 extern const char *struct_to_json (qm_type qmt, void *q_mess);
00004 extern void *string_to_struct (const char *json_string, qm_type *type);
```

6.10 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

  If it is 0 the log file will be cleared, if is 1 the log file will we open as append.

6.10.1 Detailed Description

This file defines some QoL functions.

Definition in file print_utils.c.

6.10.2 Function Documentation

6.10.2.1 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 145 of file print_utils.c.

6.10.2.2 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

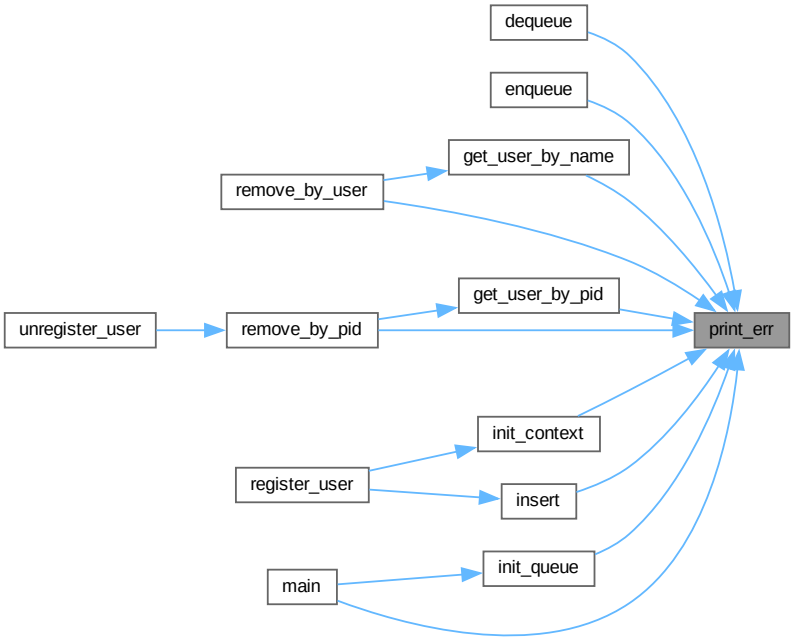
"Err_Numebr:d" will be appended to the formatted string describing the error number

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

Definition at line 79 of file print_utils.c.

Referenced by dequeue(), enqueue(), get_user_by_name(), get_user_by_pid(), init_context(), init_queue(), insert(), main(), remove_by_pid(), and remove_by_user().

Here is the caller graph for this function:



6.10.2.3 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

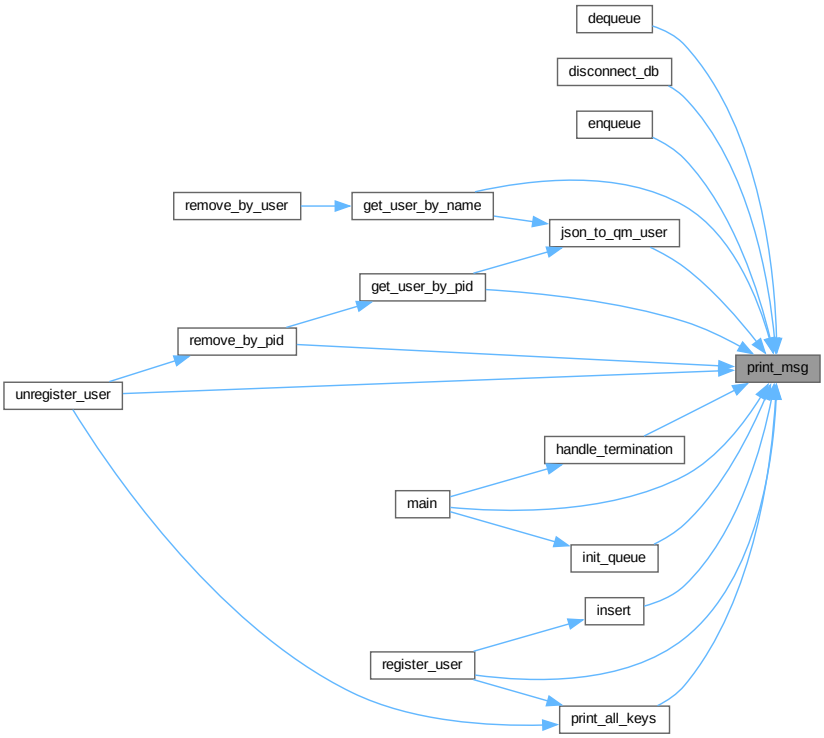
Will also log using systemD

"MESSAGE=" will be prepended to format

Definition at line 101 of file print_utils.c.

Referenced by dequeue(), disconnect_db(), enqueue(), get_user_by_name(), get_user_by_pid(), handle_termination(), init_queue(), insert(), json_to_qm_user(), main(), print_all_keys(), register_user(), remove_by_pid(), and unregister_user().

Here is the caller graph for this function:



6.10.2.4 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 123 of file print_utils.c.

6.10.3 Variable Documentation

6.10.3.1 cleared

```
int cleared = 0
```

If it is 0 the log file will be cleared, if is 1 the log file will we open as append.

Definition at line 14 of file print_utils.c.

6.11 print_utils.c

Go to the documentation of this file.

```
00001 #include "print_utils.h"
00002
00014 int cleared = 0;
00015
00024 void
00025 log_message (const char *log)
00026 {
00027   printf ("%s\n", log);
00028   // Path of the log folder and log file
00029   const char *logFolder = "/var/log/tcfs";
00034   const char *logFile = "/var/log/tcfs/log.txt";
00035
00036   // Check if the folder exists, otherwise create it
00037   struct stat st;
00038   if (stat (logFolder, &st) == -1)
00039     {
00040       mkdir (logFolder, 0700);
00041     }
00042
00043   FILE *file;
00044   if (cleared == 0)
00045     {
00046       cleared = 1;
00047       file = fopen (logFile, "w");
00048     }
00049   else
00050     {
00051       file = fopen (logFile, "a");
00052     }
00053
00054   // Open the log file in append mode
00055   if (file == NULL)
00056     {
00057       perror ("Error opening the log file");
00058     }
00059
00060   // Write the message to the log file
00061   fprintf (file, "%s\n", log);
00062
00063   // Close the file
00064   fclose (file);
00065 }
00066
00078 void
00079 print_err (const char *format, ...)
00080 {
00081   va_list args;
00082   va_start (args, format);
00083   char buffer[1024];
00084   vsnprintf (buffer, sizeof (buffer), format, args);
```

```
00085   va_end (args);
00086
00087   log_message (buffer);
00088
00089   sd_journal_print (LOG_ERR, "ERROR=%s Err_Number:%d -> %s", buffer, errno,
00090                     strerror (errno));
00091 }
00100 void
00101 print_msg (const char *format, ...)
00102 {
00103   va_list args;
00104   va_start (args, format);
00105   char buffer[1024];
00106   vsnprintf (buffer, sizeof (buffer), format, args);
00107   va_end (args);
00108
00109   log_message (buffer);
00110
00111   sd_journal_send ("MESSAGE=%s", buffer, NULL);
00112 }
00113
00122 void
00123 print_warn (const char *format, ...)
00124 {
00125   va_list args;
00126   va_start (args, format);
00127   char buffer[1024];
00128   vsnprintf (buffer, sizeof (buffer), format, args);
00129   va_end (args);
00130
00131   log_message (buffer);
00132
00133   sd_journal_print (LOG_WARNING, "WARNING=%s", buffer, NULL);
00134 }
00135
00144 void
00145 print_debug (const char *format, ...)
00146 {
00147   va_list args;
00148   va_start (args, format);
00149   char buffer[1024];
00150   vsnprintf (buffer, sizeof (buffer), format, args);
00151   va_end (args);
00152
00153   log_message (buffer);
00154
00155   sd_journal_print (LOG_DEBUG, "DEBUG=%s", buffer, NULL);
00156 }
```

6.12 print_utils.h

```
00001 #include <errno.h>
00002 #include <stdio.h>
00003 #include <stdlib.h>
00004 #include <sys/stat.h>
00005 #include <systemd/sd-journal.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, ...);
```

6.13 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 }*/
```

6.14 tcfs_daemon_tools.h

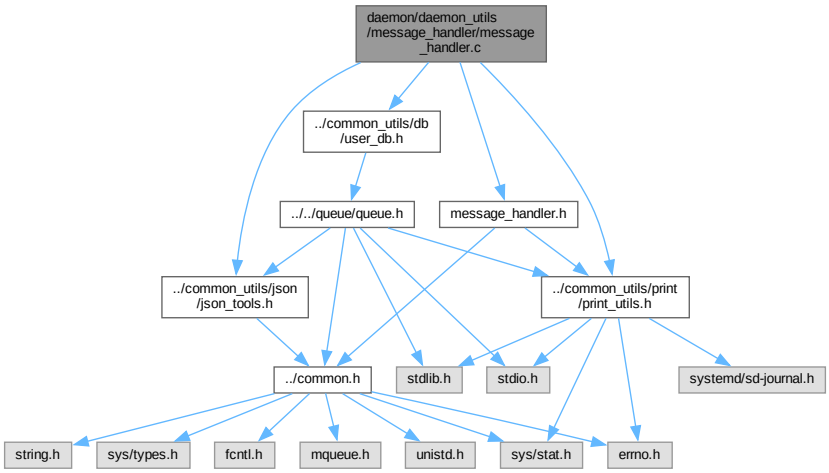
```
00001 #include "../message_handler/message_handler.h"
00002 #include "../queue/queue.h"
00003 #include <fcntl.h>
00004 #include <pthread.h>
00005 #include <signal.h>
00006 #include <stdbool.h>
00007 #include <stdlib.h>
00008 #include <sys/socket.h>
00009 #include <sys/stat.h>
00010 #include <sys/un.h>
00011 #include <unistd.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);
```

6.15 daemon/daemon_utils/message_handler/message_handler.c File Reference

This file contains the logic implementation for handling every kink of message.

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

Include dependency graph for message_handler.c:



Functions

- int handle_user_message (qm_user *user_msg)

6.15.1 Detailed Description

This file contains the logic implementation for handling every kink of message.

Definition in file message_handler.c.

6.15.2 Function Documentation

6.15.2.1 handle_user_message()

```
int handle_user_message (
            qm_user * user_msg )
```

Definition at line 12 of file message_handler.c.

6.16 message_handler.c

Go to the documentation of this file.

```
00001 #include "message_handler.h"
00002 #include "../common_utils/db/user_db.h"
00003 #include "../common_utils/json/json_tools.h"
00004 #include "../common_utils/print/print_utils.h"
00005
00011 int
00012 handle_user_message (qm_user *user_msg)
00013 {
00014   if (user_msg->user_op == REGISTER)
00015     {
00016       register_user (user_msg);
00017     }
00018   else if (user_msg->user_op == UNREGISTER)
00019     {
00020       unregister_user (user_msg->pid);
00021       // TODO: next line is a test, remove it
00022       free_context ();
00023     }
00024   else
00025     {
00026       print_err ("Unknown user operation %d", user_msg->user_op);
00027       return 0;
00028     }
00029
00030   return 1;
00031 }
```

6.17 message_handler.h

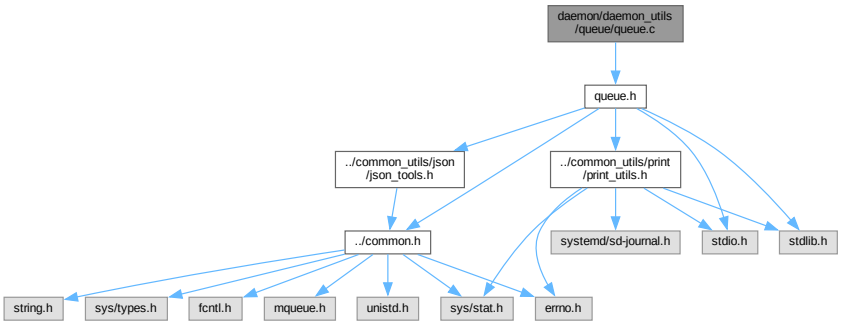
```
00001 #include "../common.h"
00002 #include "../common_utils/print/print_utils.h"
00003
00004 int handle_user_message (qm_user *user_msg);
```

6.18 daemon/daemon_utils/queue/queue.c File Reference

This file contains the implementation of a "facade pattern" for handling the queue in an easier way.

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

Include dependency graph for queue.c:



Macros

- #define MESSAGE_BUFFER_SIZE 256
- #define MQUEUE_N 256;

Functions

- mqd_t init_queue (char *queue)

  Initialize the message queue.
- int enqueue (mqd_t queue_d, qm_type qmt, void *q_mess)

  Enqueues a message void message on the queue.*
- void * dequeue (mqd_t queue_d, qm_type *qmt)

  Dequeue a message from the queue and get is as a void pointing to a structure that will be either qm_user.*

6.18.1 Detailed Description

This file contains the implementation of a "facade pattern" for handling the queue in an easier way.

Definition in file queue.c.

6.18.2 Macro Definition Documentation

6.18.2.1 MESSAGE_BUFFER_SIZE

```
#define MESSAGE_BUFFER_SIZE 256
```

Definition at line 13 of file queue.c.

6.18.2.2 MQUEUE_N

```
#define MQUEUE_N 256;
```

Definition at line 18 of file queue.c.

6.18.3 Function Documentation

6.18.3.1 dequeue()

```
void * dequeue (
            mqd_t queue_d,
            qm_type * qmt )
```

Dequeue a message from the queue and get is as a void∗ pointing to a structure that will be either qm_user.

See also

qm_user
qm_shared

qm_shared
qm_broad

qm_broad
qm_type ∗qmt will be set to the corresponding type. You can yse this value to cast the returned value back to a structure

Parameters

mqd_t	queue_d Message queue descriptor type
qm_type	∗qmt Pointer to a struct indicating the type of the returned parameter

See also

qm_type

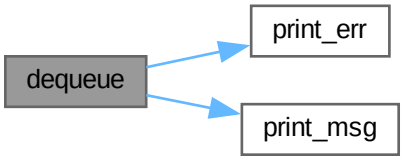
Returns

A pointer to a structure containing the structured message data. If an error occurs NULL is returned

Definition at line 95 of file queue.c.

References print_err(), and print_msg().

Here is the call graph for this function:



6.18.3.2 enqueue()

```
int enqueue (
            mqd_t queue_d,
            qm_type qmt,
            void * q_mess )
```

Enqueues a message void* message on the queue.

Parameters

mqd_t	q_mess Message queue descriptor type
qm_type	qmt enum describing the type of the message.

See also

qm_type

Parameters

void	*q_mess Actual message, this must be either qm_user

See also

qm_user
qm_shared

qm_shared
qm_broad

qm_broad

Returns

1 if successful, 0 otherwise. An error might be printen by print_err() function,

See also

print_err

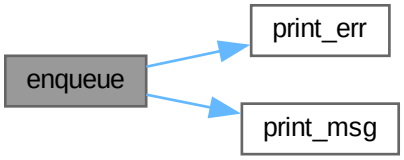
Note

The structure representing the message will be casted to a json and then it will be enqueued

Definition at line 67 of file queue.c.

References print_err(), and print_msg().

Here is the call graph for this function:



6.18.3.3 init_queue()

```
mqd_t init_queue (
            char * queue )
```

Initialize the message queue.

Parameters

char	*queue the path of the queue file

Returns

mqd_t Message queue descriptor

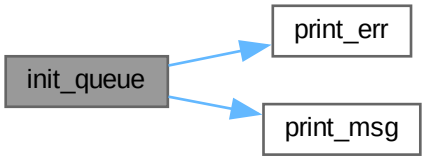
Todo Define permissions for mq_open

Definition at line 27 of file queue.c.

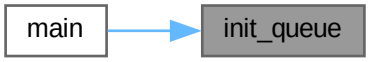
References print_err(), and print_msg().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



6.19 queue.c

Go to the documentation of this file.

```
00001 #include "queue.h"
00002
00013 #define MESSAGE_BUFFER_SIZE 256
00018 #define MQUEUE_N 256;
00019
00026 mqd_t
00027 init_queue (char *queue)
00028 {
00029   struct mq_attr attr;
00030   mqd_t mq;
00031
00032   // Initialize queue attributes
00033   attr.mq_flags = 0;
00034   attr.mq_maxmsg = MAX_QM_N;     // Maximum number of messages in the queue
00035   attr.mq_msgsize = MAX_QM_SIZE; // Maximum size of a single message
00036   attr.mq_curmsgs = 0;
00037
00038   // Create the message queue
00039   mq = mq_open (queue, O_CREAT | O_RDWR /*| O_RDONLY | O_NONBLOCK*/, 0777,
00040                 &attr); // TODO: Better define permissions
00041   printf ("mqopen %d\n", mq);
00042   if (mq == (mqd_t)-1)
00043     {
00044       print_err ("mq_open cannot create que in %s %d %s", queue, errno,
00045                  strerror (errno));
00046       print_msg ("mq_open cannot create que in %s %d %s", queue, errno,
00047                  strerror (errno));
00048       return 0;
00049     }
00050   printf ("Message queue created successfully at %s!\n", queue);
00051   return mq;
00052 }
00053
00066 int
00067 enqueue (mqd_t queue_d, qm_type qmt, void *q_mess)
00068 {
```

```
00069   const char *qm_json = struct_to_json (qmt, q_mess);
00070
00071   if (mq_send (queue_d, qm_json, strlen (qm_json) + 1, 0) == -1)
00072     {
00073       print_err ("mq_send %s", qm_json);
00074       free ((void *)qm_json);
00075       return 0;
00076     }
00077   print_msg ("Message sent successfully!\n");
00078   free ((void *)qm_json);
00079   return 1;
00080 }
00081
00094 void *
00095 dequeue (mqd_t queue_d, qm_type *qmt)
00096 {
00097   char *qm_json = (char *)malloc (sizeof (char) * MAX_QM_SIZE);
00098
00099   if (mq_receive (queue_d, qm_json, MAX_QM_SIZE, 0) == -1)
00100     {
00101       free ((void *)qm_json);
00102       print_err ("mq_rec %d %s", errno, strerror (errno));
00103       return NULL;
00104     }
00105
00106   print_msg ("Dequeued %s", qm_json);
00107   void *tmp_struct = string_to_struct (qm_json, qmt);
00108
00109   free ((void *)qm_json);
00110   return tmp_struct;
00111 }
```

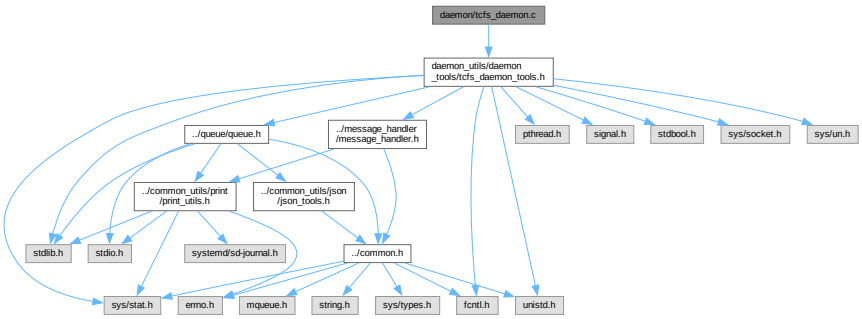
6.20 queue.h

```
00001 #include "../common.h"
00002 #include "../common_utils/json/json_tools.h"
00003 #include "../common_utils/print/print_utils.h"
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006
00007 mqd_t init_queue (char *queue);
00008 int enqueue (mqd_t queue_d, qm_type qmt, void *q_mess);
00009 void *dequeue (mqd_t queue_d, qm_type *qmt);
```

6.21 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

6.21.1 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.

6.21.2 Function Documentation

6.21.2.1 handle_termination()

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

Handle the termination if SIGTERM is received.

Parameters

int	signum Integer corresponding to SIGNUM

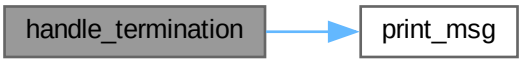
Todo : Implement remove_queue() to clear and delete the queue

Definition at line 40 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:

6.21.2.2 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

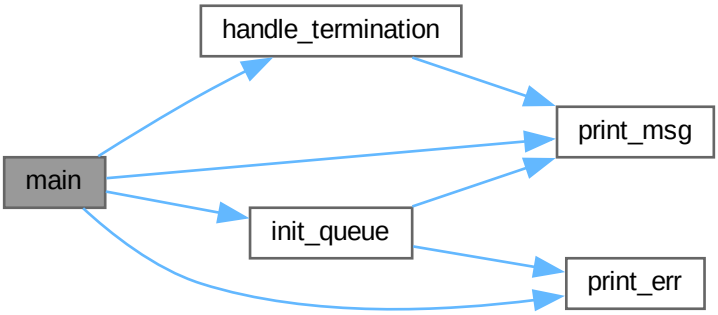
Todo : The brief description is basically false advertisement. It only spawn a thread and hangs infinitely

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

Definition at line 56 of file tcfs_daemon.c.

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

Here is the call graph for this function:



6.21.3 Variable Documentation

6.21.3.1 MQUEUE

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

the queue file location

Definition at line 32 of file tcfs_daemon.c.

Referenced by main().

6.21.3.2 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 20 of file tcfs_daemon.c.

Referenced by main().

6.21.3.3 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 26 of file tcfs_daemon.c.

6.22 tcfs_daemon.c

Go to the documentation of this file.

```
00001 #include "daemon_utils/daemon_tools/tcfs_daemon_tools.h"
00002
00020 volatile int terminate = 0;
00026 pthread_mutex_t terminate_mutex = PTHREAD_MUTEX_INITIALIZER;
00027
00032 const char MQUEUE[] = "/tcfs_queue";
00033
00039 void
00040 handle_termination (int signum)
00041 {
00042   print_msg ("TCFS TERMINATED.\n");
00043   // remove_empty_queue(queue_id);
00044   exit (0);
00045 }
00046
00055 int
00056 main ()
00057 {
00058   signal (SIGTERM, handle_termination);
00059
00060   print_msg ("TCFS daemon is starting");
00061
00062   /*pid_t pid;
00063
00064   // Fork off the parent process
00065   pid = fork();
00066
00067   // An error occurred
00068   if (pid < 0)
00069       exit(EXIT_FAILURE);
00070
00071   // Success: Let the parent terminate
00072   if (pid > 0)
00073       exit(EXIT_SUCCESS);
00074
00075   // On success: The child process becomes session leader
00076   if (setsid() < 0)
00077       exit(EXIT_FAILURE);
00078
00079   // Catch, ignore and handle signals
00080   signal(SIGCHLD, SIG_IGN);
00081   signal(SIGHUP, SIG_IGN);
00082
00083   // Fork off for the second time
00084   pid = fork();
00085
00086   // An error occurred
00087   if (pid < 0)
00088       exit(EXIT_FAILURE);
00089
00090   // Success: Let the parent terminate
00091   if (pid > 0)
00092       exit(EXIT_SUCCESS);
00093
00094   // Set new file permissions
00095   umask(0);
00096
00097   // Change the working directory to the root directory
00098   // or another appropriated directory
00099   chdir("/");
00100
00101   // Close all open file descriptors
00102   int x;
00103   for (x = sysconf(_SC_OPEN_MAX); x>=0; x--)
00104   {
00105       close (x);
00106   }*/
00107
00108   pthread_t thread1, thread2;
00109
00110   mqd_t queue_id = init_queue ((char *)MQUEUE);
00111   printf ("TEST %d", (int)queue_id);
00112   if (queue_id == 0)
00113     {
00114       print_err ("Cannot open message queue in %s", (char *)MQUEUE);
00115       unlink (MQUEUE);
00116       return -errno;
00117     }
00118
00119   if (pthread_create (&thread1, NULL, handle_incoming_messages, &queue_id)
00120       != 0)
00121     {
```

```
00122       print_err ("Failed to create thread1");
00123       mq_close (queue_id);
00124       unlink (MQUEUE);
00125       return -errno;
00126     }
00127
00128   if (pthread_create (&thread2, NULL, handle_outgoing_messages, &queue_id)
00129       != 0)
00130     {
00131       print_err ("Failed to create thread1");
00132       mq_close (queue_id);
00133       unlink (MQUEUE);
00134       return -errno;
00135     }
00136
00137   while (!terminate)
00138     {
00139     }
00140
00141   pthread_join (thread1, NULL);
00142   pthread_join (thread2, NULL);
00143
00144   mq_close (queue_id);
00145   unlink (MQUEUE);
00146
00147   print_err ("TCFS daemon threads returned, this should have never happened");
00148
00149   return -1;
00150 }
```

6.23 tcfs_kmodule.c

```
00001 /*
00002 #include <linux/kernel.h>
00003 #include <linux/module.h>
00004 #include <linux/slab.h>
00005 #include <linux/syscalls.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 */
```

6.24 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
00010 do_mount ()
00011 {
00012   int choice = -1;
00013   do
00014     {
00015       printf ("Chose between:\n"
00016               "\t1. Network FS\n"
00017               "\t2. Local FS\n"
00018               "\t3. Local folder");
00019       scanf ("%d", &choice);
00020       if (choice != 1 && choice != 2 && choice != 3)
00021         printf ("Err: Select 1 or 2\n");
00022     }
00023   while (choice != 1 && choice != 2 && choice != 3);
00024   printf ("You chose %d\n", choice);
00025
00026   if (choice == 1)
00027     {
00028       return handle_remote_mount ();
00029     }
00030   else if (choice == 2)
00031     {
00032       return handle_local_mount ();
00033     }
00034   else if (choice == 3)
00035     {
00036       return handle_folder_mount ();
00037     }
00038   printf ("Unrecoverable error\n");
00039   return 0;
00040 }
00041
00042 int
00043 generate_random_string (char *str)
00044 {
00045   if (str == NULL)
00046     return 0;
00047   for (int i = 0; i < 10; i++)
00048     str[i] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
00049         [rand () % 62];
00050   str[10] = '\0';
00051   return 1;
00052 }
00053
00054 int
00055 directory_exists (const char *path)
00056 {
00057   struct stat sb;
00058   return stat (path, &sb) == 0 && S_ISDIR (sb.st_mode);
00059 }
00060
00061 char *
00062 setup_env ()
00063 {
00064   printf ("SETUP ENV\n");
00065   char *home = getenv ("HOME");
00066   printf ("$HOME=%s\n", home);
00067
00068   char *tcfs_path
00069       = malloc ((strlen (home) + strlen ("/.tcfs\0")) * sizeof (char));
00070   char rand_path_name[11];
00071   char *new_path = NULL;
00072
00073   if (home == NULL)
00074     {
00075       perror ("Could not get $HOME\n");
00076       return 0;
00077     }
00078
00079   if (tcfs_path == NULL)
00080     {
00081       perror ("Could not allocate string tcfs_path");
00082       return 0;
00083     }
00084   sprintf (tcfs_path, "%s/%s", home, ".tcfs");
00085
00086   //$HOME/.tcfs does not exist if this is true
00087   if (directory_exists (tcfs_path) == 0)
00088     {
00089       if (mkdir (tcfs_path, 0770) == -1)
00090         {
00091           perror ("Cannot create .tcfs directory");
```

```
00092           return 0;
00093         }
00094     }
00095   // Create a folder to mount the source to
00096   // Generate a random path name
00097   if (generate_random_string (rand_path_name) == 0)
00098     {
00099       fprintf (stderr, "Err: Name generation for temp folder failed\n");
00100       return 0;
00101     }
00102   // Build the path from / to the generated path
00103   new_path = malloc ((strlen (rand_path_name) + strlen (tcfs_path) + 1)
00104                      * sizeof (char));
00105   if (new_path == NULL)
00106     {
00107       perror ("Cannot allocate new memory for path name");
00108       return 0;
00109     }
00110   sprintf (new_path, "%s/%s", tcfs_path, rand_path_name);
00111   if (mkdir (new_path, 0770) == -1)
00112     {
00113       perror ("Cannot create the tmp folder inside .tcfs");
00114       return 0;
00115     }
00116
00117   printf ("New path %s\n", new_path);
00118   free (tcfs_path);
00119   return new_path;
00120 }
00121
00122 void
00123 get_pass (char *pw)
00124 {
00125   struct termios old, new;
00126   int i = 0;
00127   int ch = 0;
00128
00129   // Disable character echo
00130   tcgetattr (STDIN_FILENO, &old);
00131   new = old;
00132   new.c_lflag &= ~ECHO;
00133   tcsetattr (STDIN_FILENO, TCSANOW, &new);
00134
00135   printf ("Please enter a password exactly %d characters long:\n", PASS_SIZE);
00136
00137   while (strlen (pw) * sizeof (char) < (PASS_SIZE - 1) * sizeof (char))
00138     {
00139       while (1)
00140         {
00141           ch = getchar ();
00142           if (ch == '\r' || ch == '\n' || ch == EOF)
00143             {
00144               break;
00145             }
00146           if (i < PASS_SIZE - 1)
00147             {
00148               pw[i] = ch;
00149               pw[i + 1] = '\0';
00150             }
00151           i++;
00152         }
00153     }
00154
00155   // Restore terminal settings
00156   tcsetattr (STDIN_FILENO, TCSANOW, &old);
00157   printf ("\nPassword successfully entered!\n");
00158 }
00159
00160 void
00161 get_source_dest (char *source, char *dest)
00162 {
00163   printf ("Please type the path to the source\n");
00164   scanf ("%s", source);
00165
00166   printf ("Please type where it should be mounted\n");
00167   scanf ("%s", dest);
00168 }
00169
00170 char *
00171 create_tcfs_mount_folder ()
00172 {
00173   char *tmp_path = NULL;
00174
00175   // Create a folder to mount it to
00176   srand (time (NULL));
00177   char random_string[11];
00178   if (generate_random_string (random_string) == 0)
```

```
00179     {
00180       fprintf (stderr, "Err: cannot generate a folder to mount to\n");
00181       return 0;
00182     }
00183   tmp_path = setup_env ();
00184   if (tmp_path == NULL)
00185     {
00186       fprintf (stderr, "Err: could not get temp path\n");
00187       return 0;
00188     }
00189   printf ("Creating dir: %s\n", tmp_path);
00190   return tmp_path;
00191 }
00192
00193 int
00194 mount_tcfs_folder (char *tmp_path, char *destination)
00195 {
00196   char pass[PASS_SIZE] = "\0";
00197   struct termios old, new;
00198
00199   // Disable character echo
00200   tcgetattr (STDIN_FILENO, &old);
00201   new = old;
00202   new.c_lflag &= ~ECHO;
00203   tcsetattr (STDIN_FILENO, TCSANOW, &new);
00204
00205   get_pass (pass);
00206   if (pass[0] == '\0')
00207     {
00208       tcsetattr (STDIN_FILENO, TCSANOW, &old);
00209       fprintf (stderr, "Could not get password\n");
00210       return 0;
00211     }
00212
00213   // Mount tmpfolder to the destination
00214   char *tcfs_command
00215       = malloc ((strlen ("tcfs -s ") + strlen (tmp_path) + strlen (" -d ")
00216                  + strlen (destination) + strlen (" -p ") + strlen (pass)));
00217   sprintf (tcfs_command, "tcfs -s %s -d %s -p %s", tmp_path, destination,
00218            pass);
00219
00220   int status_tcfs_mount = system (tcfs_command);
00221   if (!(WIFEXITED (status_tcfs_mount) && WEXITSTATUS (status_tcfs_mount) == 0))
00222     {
00223       tcsetattr (STDIN_FILENO, TCSANOW, &old);
00224       perror ("Could not execute the command");
00225       return 0;
00226     }
00227   free (tcfs_command);
00228   tcsetattr (STDIN_FILENO, TCSANOW, &old);
00229   return 1;
00230 }
00231
00232 int
00233 handle_local_mount ()
00234 {
00235   char source[PATH_MAX];
00236   char destination[PATH_MAX];
00237   char *tmp_path = NULL;
00238
00239   get_source_dest (source, destination);
00240
00241   tmp_path = create_tcfs_mount_folder ();
00242   if (tmp_path == NULL)
00243     {
00244       printf ("Err: could not get tmp folder path\n");
00245       return 0;
00246     }
00247
00248   // Mount block device to temp folder
00249   char *command = malloc (
00250       (strlen ("mount ") + strlen (source) + strlen (" ") + strlen (tmp_path))
00251       * sizeof (char));
00252   if (command == NULL)
00253     {
00254       perror ("cannot allocate memoty for the command");
00255       return 0;
00256     }
00257   sprintf (command, "sudo mount -o umask=0755,gid=1000,uid=1000 %s %s", source,
00258            tmp_path);
00259   printf ("executing: %s\n", command);
00260   int status_tmp_mount = system (command);
00261   if (!(WIFEXITED (status_tmp_mount) && WEXITSTATUS (status_tmp_mount) == 0))
00262     {
00263       perror ("Could not execute the command");
00264       return 0;
00265     }
```

```
00266
00267   int res = mount_tcfs_folder (tmp_path, destination);
00268   if (res == 0)
00269     return 0;
00270
00271   free (tmp_path);
00272   free (command);
00273   return 1;
00274 }
00275
00276 int
00277 handle_folder_mount ()
00278 {
00279   char source[PATH_MAX];
00280   char destination[PATH_MAX];
00281
00282   get_source_dest (source, destination);
00283   if (source[0] == '\0' || destination[0] == '\0')
00284     {
00285       printf ("Err: Could not get source or destination\n");
00286       return 0;
00287     }
00288   printf ("Source:%s\tdestination:%s\n", source, destination);
00289
00290   int res = mount_tcfs_folder (source, destination);
00291   if (res == 0)
00292     return 0;
00293
00294   return 1;
00295 }
00296
00297 void
00298 clearKeyboardBuffer ()
00299 {
00300   int ch;
00301   while ((ch = getchar ()) != EOF && ch != '\n')
00302     ;
00303 }
00304
00305 int
00306 handle_remote_mount ()
00307 {
00308   char source[PATH_MAX] = "\0";
00309   char destination[PATH_MAX] = "\0";
00310   char command[100] = "\0";
00311
00312   printf ("WARN: This function is not complete, I don't know how many remote "
00313           "FileSystems support extended "
00314           "attributes, please mount it manually. "
00315           "\nEX:sudo mount -t nfs -o umask=0755,gid=1000,uid=1000 "
00316           "10.10.10.10:/NFS /mnt\n");
00317
00318   clearKeyboardBuffer ();
00319   printf ("Enter the command: ");
00320   int ch;
00321   int loop = 0;
00322   while (loop < 99 && (ch = getc (stdin)) != EOF && ch != '\n')
00323     {
00324       command[loop] = ch;
00325       ++loop;
00326     }
00327   command[loop] = '\0'; // Null-terminate the string
00328
00329   printf ("Command: %s\n", command);
00330   int status = system (command);
00331   if (!(WIFEXITED (status) && WEXITSTATUS (status) == 0))
00332     {
00333       perror ("Could not execute the command");
00334       return 0;
00335     }
00336
00337   printf ("Where has it been mounted? ");
00338   loop = 0;
00339   while (loop < PATH_MAX - 1 && (ch = getc (stdin)) != EOF && ch != '\n')
00340     {
00341       source[loop] = ch;
00342       ++loop;
00343     }
00344   source[loop] = '\0'; // Null-terminate the string
00345
00346   printf ("Source: %s\n", source);
00347
00348   printf ("Where should TCFS mount it? ");
00349   loop = 0;
00350   while (loop < PATH_MAX - 1 && (ch = getc (stdin)) != EOF && ch != '\n')
00351     {
00352       destination[loop] = ch;
```

```
00353       ++loop;
00354     }
00355   destination[loop] = '\0'; // Null-terminate the string
00356
00357   printf ("Destination: %s\n", destination);
00358
00359   int res = mount_tcfs_folder (source, destination);
00360   return res;
00361 }
```

6.25 tcfs_helper_tools.h

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

6.26 user_tcfs.c

```
00001 #include "tcfs_helper_tools.h"
00002 #include <argp.h>
00003 #include <stdio.h>
00004 #include <stdlib.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, "
00010                     "create-shared, or umount.";
00011
00012 // Define the accepted options
00013 static struct argp_option options[]
00014     = { { "mount", 'm', 0, 0, "Perform mount operation", -1 },
00015         { "create-shared", 'c', 0, 0, "Perform create-shared operation", -1 },
00016         { "umount", 'u', 0, 0, "Perform umount operation", -1 },
00017         { NULL } };
00018
00019 // Structure to hold the parsed arguments
00020 struct arguments
00021 {
00022   int operation;
00023 };
00024
00025 // Parse the arguments
00026 static error_t
00027 parse_opt (int key, char *arg, struct argp_state *state)
00028 {
00029   (void)arg;
00030
00031   struct arguments *arguments = state->input;
00032   switch (key)
00033     {
00034     case 'm':
00035       arguments->operation = 1; // Mount
00036       break;
00037     case 'c':
00038       arguments->operation = 2; // Create-shared
00039       break;
00040     case 'u':
00041       arguments->operation = 3; // Umount
00042       break;
00043     default:
00044       return ARGP_ERR_UNKNOWN;
00045     }
00046   return 0;
00047 }
00048
00049 // Define the argp object
00050 static struct argp argp = { .options = options,
00051                             .parser = parse_opt,
00052                             .doc = doc,
00053                             .args_doc = NULL,
```

```
00054                             .children = NULL,
00055                             .help_filter = NULL };
00056
00057 int
00058 main (int argc, char *argv[])
00059 {
00060   struct arguments arguments;
00061   arguments.operation = 0; // Default value
00062
00063   // Parse the arguments
00064   argp_parse (&argp, argc, argv, 0, 0, &arguments);
00065
00066   arguments.operation = 1; // TODO: option 1 is the only one implemented
00067   switch (arguments.operation)
00068     {
00069     case 1:
00070       printf ("Mounting your FS, Please specify the location\n");
00071       int result = do_mount ();
00072       if (result == 0)
00073         {
00074           fprintf (stderr, "An error occurred\n");
00075           exit (-1);
00076         }
00077       break;
00078     case 2:
00079       printf ("You chose the 'create-shared' operation.\n");
00080       // Add specific logic for 'create-shared' here.
00081       break;
00082     case 3:
00083       printf ("You chose the 'umount' operation.\n");
00084       // Add specific logic for 'umount' here.
00085       break;
00086     default:
00087       printf ("Invalid argument. Choose from 'mount', 'create-shared', or "
00088               "'umount'.\n");
00089       return 1;
00090     }
00091
00092   return 0;
00093 }
```

6.27 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 "utils/crypt-utils/crypt-utils.h"
00016 #include "utils/tcfs_utils/tcfs_utils.h"
00017 #include <argp.h>
00018 #include <assert.h>
00019 #include <dirent.h>
00020 #include <errno.h>
00021 #include <fcntl.h> /* Definition of AT_* constants */
00022 #include <fuse.h>
00023 #include <limits.h>
00024 #include <linux/limits.h>
00025 #include <pwd.h>
00026 #include <stdio.h>
00027 #include <string.h>
00028 #include <sys/stat.h>
00029 #include <sys/time.h>
00030 #include <sys/xattr.h>
00031 #include <time.h>
00032 #include <unistd.h>
00033
00034 char *root_path;
00035 char *password;
00036
00037 static int tcfs_getxattr (const char *fuse_path, const char *name, char *value,
00038                           size_t size);
00039
00040 static int
```

```
00041 tcfs_opendir (const char *fuse_path, struct fuse_file_info *fi)
00042 {
00043   (void)fuse_path;
00044   (void)fi;
00045   printf ("Called opendir UNIMPLEMENTED\n");
00046   /*int res = 0;
00047   DIR *dp;
00048   char path[PATH_MAX];
00049
00050   *path = prefix_path(fuse_path);
00051
00052   dp = opendir(path);
00053   if (dp == NULL)
00054       res = -errno;
00055
00056   fi->fh = (intptr_t) dp;
00057
00058   return res;*/
00059   return 0;
00060 }
00061
00062 static int
00063 tcfs_getattr (const char *fuse_path, struct stat *stbuf)
00064 {
00065   printf ("Called getattr\n");
00066   char *path = prefix_path (fuse_path, root_path);
00067
00068   int res;
00069
00070   res = stat (path, stbuf);
00071   if (res == -1)
00072     return -errno;
00073
00074   return 0;
00075 }
00076
00077 static int
00078 tcfs_access (const char *fuse_path, int mask)
00079 {
00080   printf ("Callen access\n");
00081   char *path = prefix_path (fuse_path, root_path);
00082
00083   int res;
00084
00085   res = access (path, mask);
00086   if (res == -1)
00087     return -errno;
00088
00089   return 0;
00090 }
00091
00092 static int
00093 tcfs_readlink (const char *fuse_path, char *buf, size_t size)
00094 {
00095   char *path = prefix_path (fuse_path, root_path);
00096
00097   int res;
00098
00099   res = readlink (path, buf, size - 1);
00100   if (res == -1)
00101     return -errno;
00102
00103   buf[res] = '\0';
00104   return 0;
00105 }
00106
00107 static int
00108 tcfs_readdir (const char *fuse_path, void *buf, fuse_fill_dir_t filler,
00109               off_t offset, struct fuse_file_info *fi)
00110 {
00111   (void)offset;
00112   (void)fi;
00113
00114   printf ("Called readdir %s\n", fuse_path);
00115   char *path = prefix_path (fuse_path, root_path);
00116
00117   DIR *dp;
00118   struct dirent *de;
00119
00120   dp = opendir (path);
00121   if (dp == NULL)
00122     {
00123       perror ("Could not open the directory");
00124       return -errno;
00125     }
00126
00127   while ((de = readdir (dp)) != NULL)
```

```
00128     {
00129       struct stat st;
00130       memset (&st, 0, sizeof (st));
00131       st.st_ino = de->d_ino;
00132       st.st_mode = de->d_type « 12;
00133       if (filler (buf, de->d_name, &st, 0))
00134         break;
00135     }
00136
00137   closedir (dp);
00138   return 0;
00139 }
00140
00141 static int
00142 tcfs_mknod (const char *fuse_path, mode_t mode, dev_t rdev)
00143 {
00144   printf ("Called mknod\n");
00145   char *path = prefix_path (fuse_path, root_path);
00146
00147   int res;
00148
00149   /* On Linux this could just be 'mknod(path, mode, rdev)' but this
00150      is more portable */
00151   if (S_ISREG (mode))
00152     {
00153       res = open (path, O_CREAT | O_EXCL | O_WRONLY, mode);
00154       if (res >= 0)
00155         res = close (res);
00156     }
00157   else if (S_ISFIFO (mode))
00158     res = mkfifo (path, mode);
00159   else
00160     res = mknod (path, mode, rdev);
00161   if (res == -1)
00162     return -errno;
00163
00164   return 0;
00165 }
00166
00167 static int
00168 tcfs_mkdir (const char *fuse_path, mode_t mode)
00169 {
00170   printf ("Called mkdir\n");
00171   char *path = prefix_path (fuse_path, root_path);
00172
00173   int res;
00174
00175   res = mkdir (path, mode);
00176   if (res == -1)
00177     return -errno;
00178
00179   return 0;
00180 }
00181
00182 static int
00183 tcfs_unlink (const char *fuse_path)
00184 {
00185   printf ("Called unlink\n");
00186   char *path = prefix_path (fuse_path, root_path);
00187
00188   int res;
00189
00190   res = unlink (path);
00191   if (res == -1)
00192     return -errno;
00193
00194   return 0;
00195 }
00196
00197 static int
00198 tcfs_rmdir (const char *fuse_path)
00199 {
00200   printf ("Called rmdir\n");
00201   char *path = prefix_path (fuse_path, root_path);
00202
00203   int res;
00204
00205   res = rmdir (path);
00206   if (res == -1)
00207     return -errno;
00208
00209   return 0;
00210 }
00211
00212 static int
00213 tcfs_symlink (const char *from, const char *to)
00214 {
```

```
00215   printf ("Called symlink\n");
00216   int res;
00217
00218   res = symlink (from, to);
00219   if (res == -1)
00220     return -errno;
00221
00222   return 0;
00223 }
00224
00225 static int
00226 tcfs_rename (const char *from, const char *to)
00227 {
00228   printf ("Called rename\n");
00229   int res;
00230
00231   res = rename (from, to);
00232   if (res == -1)
00233     return -errno;
00234
00235   return 0;
00236 }
00237
00238 static int
00239 tcfs_link (const char *from, const char *to)
00240 {
00241   printf ("Called link\n");
00242   int res;
00243
00244   res = link (from, to);
00245   if (res == -1)
00246     return -errno;
00247
00248   return 0;
00249 }
00250
00251 static int
00252 tcfs_chmod (const char *fuse_path, mode_t mode)
00253 {
00254   printf ("Called chmod\n");
00255   char *path = prefix_path (fuse_path, root_path);
00256
00257   int res;
00258
00259   res = chmod (path, mode);
00260   if (res == -1)
00261     return -errno;
00262
00263   return 0;
00264 }
00265
00266 static int
00267 tcfs_chown (const char *fuse_path, uid_t uid, gid_t gid)
00268 {
00269   printf ("Called chown\n");
00270   char *path = prefix_path (fuse_path, root_path);
00271
00272   int res;
00273
00274   res = lchown (path, uid, gid);
00275   if (res == -1)
00276     return -errno;
00277
00278   return 0;
00279 }
00280
00281 static int
00282 tcfs_truncate (const char *fuse_path, off_t size)
00283 {
00284   printf ("Called truncate\n");
00285   char *path = prefix_path (fuse_path, root_path);
00286
00287   int res;
00288
00289   res = truncate (path, size);
00290   if (res == -1)
00291     return -errno;
00292
00293   return 0;
00294 }
00295
00296 // #ifdef HAVE_UTIMENSAT
00297 static int
00298 tcfs_utimens (const char *fuse_path, const struct timespec ts[2])
00299 {
00300   printf ("Called utimens\n");
00301   char *path = prefix_path (fuse_path, root_path);
```

```
00302
00303   int res;
00304   struct timeval tv[2];
00305
00306   tv[0].tv_sec = ts[0].tv_sec;
00307   tv[0].tv_usec = ts[0].tv_nsec / 1000;
00308   tv[1].tv_sec = ts[1].tv_sec;
00309   tv[1].tv_usec = ts[1].tv_nsec / 1000;
00310
00311   res = utimes (path, tv);
00312   if (res == -1)
00313     return -errno;
00314
00315   return 0;
00316 }
00317 // #endif
00318
00319 static int
00320 tcfs_open (const char *fuse_path, struct fuse_file_info *fi)
00321 {
00322   printf ("Called open\n");
00323   char *path = prefix_path (fuse_path, root_path);
00324   int res;
00325
00326   res = open (path, fi->flags);
00327   if (res == -1)
00328     return -errno;
00329
00330   close (res);
00331   return 0;
00332 }
00333
00334 static inline int
00335 file_size (FILE *file)
00336 {
00337   struct stat st;
00338
00339   if (fstat (fileno (file), &st) == 0)
00340     return st.st_size;
00341
00342   return -1;
00343 }
00344
00345 static int
00346 tcfs_read (const char *fuse_path, char *buf, size_t size, off_t offset,
00347            struct fuse_file_info *fi)
00348 {
00349   (void)size;
00350   (void)fi;
00351
00352   printf ("Calling read\n");
00353   FILE *path_ptr, *tmpf;
00354   char *path;
00355   int res;
00356
00357   // Retrieve the username
00358   char username_buf[1024];
00359   size_t username_buf_size = 1024;
00360   get_user_name (username_buf, username_buf_size);
00361
00362   path = prefix_path (fuse_path, root_path);
00363
00364   path_ptr = fopen (path, "r");
00365   tmpf = tmpfile ();
00366
00367   // Get key size
00368   char *size_key_char = malloc (sizeof (char) * 20);
00369   if (tcfs_getxattr (fuse_path, "user.key_len", size_key_char, 20) == -1)
00370     {
00371       perror ("Could not get file key size");
00372       return -errno;
00373     }
00374   ssize_t size_key = strtol (size_key_char, NULL, 10);
00375
00376   // Retrive the file key
00377   unsigned char *encrypted_key = malloc ((size_key + 1) * sizeof (char));
00378   encrypted_key[size_key] = '\0';
00379   if (tcfs_getxattr (fuse_path, "user.key", (char *)encrypted_key, size_key)
00380       == -1)
00381     {
00382       perror ("Could not get encrypted key for file in tcfs_read");
00383       return -errno;
00384     }
00385
00386   // Decrypt the file key
00387   unsigned char *decrypted_key;
00388   decrypted_key = decrypt_string (encrypted_key, password);
```

```
00389
00390   /* Decrypt*/
00391   if (do_crypt (path_ptr, tmpf, DECRYPT, decrypted_key) != 1)
00392     {
00393       perror ("Err: do_crypt cannot decrypt file");
00394       return -errno;
00395     }
00396
00397   /* Something went terribly wrong if this is the case. */
00398   if (path_ptr == NULL || tmpf == NULL)
00399     return -errno;
00400
00401   if (fflush (tmpf) != 0)
00402     {
00403       perror ("Err: Cannot flush file in read process");
00404       return -errno;
00405     }
00406   if (fseek (tmpf, offset, SEEK_SET) != 0)
00407     {
00408       perror ("Err: cannot fseek while reading file");
00409       return -errno;
00410     }
00411
00412   /* Read our tmpfile into the buffer. */
00413   res = fread (buf, 1, file_size (tmpf), tmpf);
00414   if (res == -1)
00415     {
00416       perror ("Err: cannot fread whine in read");
00417       res = -errno;
00418     }
00419
00420   fclose (tmpf);
00421   fclose (path_ptr);
00422   free (encrypted_key);
00423   free (decrypted_key);
00424   return res;
00425 }
00426
00427 static int
00428 tcfs_write (const char *fuse_path, const char *buf, size_t size, off_t offset,
00429             struct fuse_file_info *fi)
00430 {
00431   (void)fi;
00432   printf ("Called write\n");
00433
00434   FILE *path_ptr, *tmpf;
00435   char *path;
00436   int res;
00437   int tmpf_descriptor;
00438
00439   path = prefix_path (fuse_path, root_path);
00440   path_ptr = fopen (path, "r+");
00441   tmpf = tmpfile ();
00442   tmpf_descriptor = fileno (tmpf);
00443
00444   // Get the key size
00445   char *size_key_char = malloc (sizeof (char) * 20);
00446   if (tcfs_getxattr (fuse_path, "user.key_len", size_key_char, 20) == -1)
00447     {
00448       perror ("Could not get file key size");
00449       return -errno;
00450     }
00451   ssize_t size_key = strtol (size_key_char, NULL, 10);
00452
00453   // Retrieve the file key
00454   unsigned char *encrypted_key
00455       = malloc (sizeof (unsigned char) * (size_key + 1));
00456   encrypted_key[size_key] = '\0';
00457   if (tcfs_getxattr (fuse_path, "user.key", (char *)encrypted_key, size_key)
00458       == -1)
00459     {
00460       perror ("Could not get file encrypted key in tcfs write");
00461       return -errno;
00462     }
00463
00464   // Decrypt the file key
00465   unsigned char *decrypted_key = malloc (sizeof (unsigned char) * 33);
00466   decrypted_key[32] = '\0';
00467   decrypted_key = decrypt_string (encrypted_key, password);
00468
00469   /* Something went terribly wrong if this is the case. */
00470   if (path_ptr == NULL || tmpf == NULL)
00471     {
00472       fprintf (stderr,
00473                "Something went terribly wrong, cannot create new files\n");
00474       return -errno;
00475     }
```

```
00476
00477   /* if the file to write to exists, read it into the tempfile */
00478   if (tcfs_access (fuse_path, R_OK) == 0 && file_size (path_ptr) > 0)
00479     {
00480       if (do_crypt (path_ptr, tmpf, DECRYPT, decrypted_key) == 0)
00481         {
00482           perror ("do_crypt: Cannot cypher file\n");
00483           return --errno;
00484         }
00485       rewind (path_ptr);
00486       rewind (tmpf);
00487     }
00488
00489   /* Read our tmpfile into the buffer. */
00490   res = pwrite (tmpf_descriptor, buf, size, offset);
00491   if (res == -1)
00492     {
00493       printf ("%d\n", res);
00494       perror ("pwrite: cannot read tmpfile into the buffer\n");
00495       res = -errno;
00496     }
00497
00498   /* Encrypt*/
00499   if (do_crypt (tmpf, path_ptr, ENCRYPT, decrypted_key) == 0)
00500     {
00501       perror ("do_crypt 2: cannot cypher file\n");
00502       return -errno;
00503     }
00504
00505   fclose (tmpf);
00506   fclose (path_ptr);
00507   free (encrypted_key);
00508   free (decrypted_key);
00509
00510   return res;
00511 }
00512
00513 static int
00514 tcfs_statfs (const char *fuse_path, struct statvfs *stbuf)
00515 {
00516   printf ("Called statfs\n");
00517   char *path = prefix_path (fuse_path, root_path);
00518
00519   int res;
00520
00521   res = statvfs (path, stbuf);
00522   if (res == -1)
00523     return -errno;
00524
00525   return 0;
00526 }
00527
00528 static int
00529 tcfs_setxattr (const char *fuse_path, const char *name, const char *value,
00530                size_t size, int flags)
00531 {
00532   char *path = prefix_path (fuse_path, root_path);
00533   int res = 1;
00534   if ((res = lsetxattr (path, name, value, size, flags)) == -1)
00535     perror ("tcfs_lsetxattr");
00536   if (res == -1)
00537     return -errno;
00538   return 0;
00539 }
00540
00541 static int
00542 tcfs_create (const char *fuse_path, mode_t mode, struct fuse_file_info *fi)
00543 {
00544   (void)fi;
00545   (void)mode;
00546   printf ("Called create\n");
00547
00548   FILE *res;
00549   res = fopen (prefix_path (fuse_path, root_path), "w");
00550   if (res == NULL)
00551     return -errno;
00552
00553   // Flag file as encrypted
00554   if (tcfs_setxattr (fuse_path, "user.encrypted", "true", 4, 0)
00555       != 0) //(fsetxattr(fileno(res), "user.encrypted", "true", 4, 0) != 0)
00556     {
00557       fclose (res);
00558       return -errno;
00559     }
00560
00561   // Generate and set a new encrypted key for the file
00562   unsigned char *key = malloc (sizeof (unsigned char) * 33);
```

```
00563   key[32] = '\0';
00564   generate_key (key);
00565
00566   if (key == NULL)
00567     {
00568       perror ("cannot generate file key");
00569       return -errno;
00570     }
00571   if (is_valid_key (key) == 0)
00572     {
00573       fprintf (stderr, "Generated key size invalid\n");
00574       return -1;
00575     }
00576
00577   // Encrypt the generated key
00578   int encrypted_key_len;
00579   unsigned char *encrypted_key
00580       = encrypt_string (key, password, &encrypted_key_len);
00581
00582   // Set the file key
00583   if (tcfs_setxattr (fuse_path, "user.key", (const char *)encrypted_key,
00584                      encrypted_key_len, 0)
00585       != 0) //(fsetxattr(fileno(res), "user.key", encrypted_key, 32, 0) != 0)
00586     {
00587       perror ("Err setting key xattr");
00588       return -errno;
00589     }
00590   // Set key size
00591   char encrypted_key_len_char[20];
00592   snprintf (encrypted_key_len_char, sizeof (encrypted_key_len_char), "%d",
00593             encrypted_key_len);
00594   if (tcfs_setxattr (fuse_path, "user.key_len", encrypted_key_len_char,
00595                      sizeof (encrypted_key_len_char), 0)
00596       != 0) //(fsetxattr(fileno(res), "user.key", encrypted_key, 32, 0) != 0)
00597     {
00598       perror ("Err setting key_len xattr");
00599       return -errno;
00600     }
00601
00602   free (encrypted_key);
00603   free (key);
00604   fclose (res);
00605   return 0;
00606 }
00607
00608 static int
00609 tcfs_release (const char *fuse_path, struct fuse_file_info *fi)
00610 {
00611   /* Just a stub.    This method is optional and can safely be left
00612      unimplemented */
00613   char *path = prefix_path (fuse_path, root_path);
00614
00615   (void)path;
00616   (void)fi;
00617   return 0;
00618 }
00619
00620 static int
00621 tcfs_fsync (const char *fuse_path, int isdatasync, struct fuse_file_info *fi)
00622 {
00623   /* Just a stub.    This method is optional and can safely be left
00624      unimplemented */
00625   char *path = prefix_path (fuse_path, root_path);
00626
00627   (void)path;
00628   (void)isdatasync;
00629   (void)fi;
00630   return 0;
00631 }
00632
00633 static int
00634 tcfs_getxattr (const char *fuse_path, const char *name, char *value,
00635                size_t size)
00636 {
00637   char *path = prefix_path (fuse_path, root_path);
00638   printf ("Called getxattr on %s name:%s size:%zu\n", path, name, size);
00639
00640   if (strcmp (name, "security.capability")
00641       == 0) // TODO: I don't know why this is called every time, understand why
00642             // and handle this
00643     return 0;
00644
00645   int res = (int)lgetxattr (path, name, value, size);
00646   if (res == -1)
00647     {
00648       perror ("Could not get xattr for file");
00649       return -errno;
```

```
00650     }
00651   return res;
00652 }
00653
00654 static int
00655 tcfs_listxattr (const char *fuse_path, char *list, size_t size)
00656 {
00657   printf ("Called listxattr\n");
00658   char *path = prefix_path (fuse_path, root_path);
00659
00660   int res = llistxattr (path, list, size);
00661   if (res == -1)
00662     return -errno;
00663   return res;
00664 }
00665
00666 static int
00667 tcfs_removexattr (const char *fuse_path, const char *name)
00668 {
00669   printf ("Called removexattr\n");
00670   char *path = prefix_path (fuse_path, root_path);
00671
00672   int res = lremovexattr (path, name);
00673   if (res == -1)
00674     return -errno;
00675   return 0;
00676 }
00677
00678 static struct fuse_operations tcfs_oper = {
00679   .opendir = tcfs_opendir,
00680   .getattr = tcfs_getattr,
00681   .access = tcfs_access,
00682   .readlink = tcfs_readlink,
00683   .readdir = tcfs_readdir,
00684   .mknod = tcfs_mknod,
00685   .mkdir = tcfs_mkdir,
00686   .symlink = tcfs_symlink,
00687   .unlink = tcfs_unlink,
00688   .rmdir = tcfs_rmdir,
00689   .rename = tcfs_rename,
00690   .link = tcfs_link,
00691   .chmod = tcfs_chmod,
00692   .chown = tcfs_chown,
00693   .truncate = tcfs_truncate,
00694   .utimens = tcfs_utimens,
00695   .open = tcfs_open,
00696   .read = tcfs_read,
00697   .write = tcfs_write,
00698   .statfs = tcfs_statfs,
00699   .create = tcfs_create,
00700   .release = tcfs_release,
00701   .fsync = tcfs_fsync,
00702   .setxattr = tcfs_setxattr,
00703   .getxattr = tcfs_getxattr,
00704   .listxattr = tcfs_listxattr,
00705   .removexattr = tcfs_removexattr,
00706 };
00707
00708 const char *argp_program_version = "TCFS Alpha";
00709 const char *argp_program_bug_address = "carloalbertogiordano@duck.com";
00710
00711 static char doc[] = "This is an implementation on TCFS\ntcfs -s <source_path> "
00712                     "-d <dest_path> -p <password> [fuse arguments]";
00713
00714 static char args_doc[] = "";
00715
00716 static struct argp_option options[]
00717     = { { "source", 's', "SOURCE", 0, "Source file path", -1 },
00718         { "destination", 'd', "DESTINATION", 0, "Destination file path", -1 },
00719         { "password", 'p', "PASSWORD", 0, "Password", -1 },
00720         { NULL } };
00721
00722 struct arguments
00723 {
00724   char *source;
00725   char *destination;
00726   char *password;
00727 };
00728
00729 static error_t
00730 parse_opt (int key, char *arg, struct argp_state *state)
00731 {
00732   struct arguments *arguments = state->input;
00733
00734   switch (key)
00735     {
00736     case 's':
```

```
00737       arguments->source = arg;
00738       break;
00739     case 'd':
00740       arguments->destination = arg;
00741       break;
00742     case 'p':
00743       arguments->password = arg;
00744       break;
00745     case ARGP_KEY_ARG:
00746       return ARGP_ERR_UNKNOWN;
00747     default:
00748       return ARGP_ERR_UNKNOWN;
00749     }
00750
00751   return 0;
00752 }
00753
00754 static struct argp argp = { options, parse_opt, args_doc, doc, 0, NULL, NULL };
00755
00756 int
00757 main (int argc, char *argv[])
00758 {
00759   umask (0);
00760
00761   struct arguments arguments;
00762
00763   arguments.source = NULL;
00764   arguments.destination = NULL;
00765   arguments.password = NULL;
00766
00767   argp_parse (&argp, argc, argv, 0, 0, &arguments);
00768
00769   if (arguments.source == NULL || arguments.destination == NULL
00770       || arguments.password == NULL)
00771     {
00772       printf ("Err: You need to specify at least 3 arguments\n");
00773       return -1;
00774     }
00775
00776   printf ("Source: %s\n", arguments.source);
00777   printf ("Destination: %s\n", arguments.destination);
00778   root_path = arguments.source;
00779
00780   if (is_valid_key ((unsigned char *)arguments.password) == 0)
00781     {
00782       fprintf (stderr, "Inserted key not valid\n");
00783       return 1;
00784     }
00785
00786   struct fuse_args args_fuse = FUSE_ARGS_INIT (0, NULL);
00787   fuse_opt_add_arg (&args_fuse, "./tcfs");
00788   fuse_opt_add_arg (&args_fuse, arguments.destination);
00789   fuse_opt_add_arg (&args_fuse,
00790                     "-f"); // TODO: this is forced for now, but will be passed
00791                            // via options in the future
00792   fuse_opt_add_arg (&args_fuse,
00793                     "-s"); // TODO: this is forced for now, but will be passed
00794                            // via options in the future
00795
00796   // Print what we are passing to fuse TODO: This will be removed
00797   for (int i = 0; i < args_fuse.argc; i++)
00798     {
00799       printf ("%s ", args_fuse.argv[i]);
00800     }
00801   printf ("\n");
00802
00803   // Get username
00804   /*
00805   char buf[1024];
00806   size_t buf_size = 1024;
00807   get_user_name(buf, buf_size);
00808   */
00809
00810   password = arguments.password;
00811
00812   return fuse_main (args_fuse.argc, args_fuse.argv, &tcfs_oper, NULL);
00813 }
```

6.28 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
00024 do_crypt (FILE *in, FILE *out, int action, unsigned char *key_str)
00025 {
00026   /* Local Vars */
00027
00028   /* Buffers */
00029   unsigned char inbuf[BLOCKSIZE];
00030   int inlen;
00031   /* Allow enough space in output buffer for additional cipher block */
00032   unsigned char outbuf[BLOCKSIZE + EVP_MAX_BLOCK_LENGTH];
00033   int outlen;
00034   int writelen;
00035
00036   /* OpenSSL libcrypto vars */
00037   EVP_CIPHER_CTX *ctx;
00038   ctx = EVP_CIPHER_CTX_new ();
00039
00040   unsigned char key[KEY_SIZE];
00041   unsigned char iv[IV_SIZE];
00042   int nrounds = 5;
00043
00044   /* tmp vars */
00045   int i;
00046   /* Setup Encryption Key and Cipher Engine if in cipher mode */
00047   if (action >= 0)
00048     {
00049       if (!key_str)
00050         {
00051           /* Error */
00052           fprintf (stderr, "Key_str must not be NULL\n");
00053           return 0;
00054         }
00055       /* Build Key from String */
00056       i = EVP_BytesToKey (EVP_aes_256_cbc (), EVP_sha1 (), NULL, key_str,
00057                           (int)strlen ((const char *)key_str), nrounds, key,
00058                           iv);
00059       if (i != 32)
00060         {
00061           /* Error */
00062           fprintf (stderr, "Key size is %d bits - should be 256 bits\n",
00063                    i * 8);
00064           return 0;
00065         }
00066       /* Init Engine */
00067       EVP_CIPHER_CTX_init (ctx);
00068       EVP_CipherInit_ex (ctx, EVP_aes_256_cbc (), NULL, key, iv, action);
00069     }
00070
00071   /* Loop through Input File*/
00072   for (;;)
00073     {
00074       /* Read Block */
00075       inlen = fread (inbuf, sizeof (*inbuf), BLOCKSIZE, in);
00076       if (inlen <= 0)
00077         {
00078           /* EOF -> Break Loop */
00079           break;
00080         }
00081
00082       /* If in cipher mode, perform cipher transform on block */
00083       if (action >= 0)
00084         {
00085           if (!EVP_CipherUpdate (ctx, outbuf, &outlen, inbuf, inlen))
00086             {
00087               /* Error */
00088               EVP_CIPHER_CTX_cleanup (ctx);
00089               return 0;
00090             }
```

```
00091         }
00092       /* If in pass-through mode. copy block as is */
00093       else
00094         {
00095           memcpy (outbuf, inbuf, inlen);
00096           outlen = inlen;
00097         }
00098
00099       /* Write Block */
00100       writelen = fwrite (outbuf, sizeof (*outbuf), outlen, out);
00101       if (writelen != outlen)
00102         {
00103           /* Error */
00104           perror ("fwrite error");
00105           EVP_CIPHER_CTX_cleanup (ctx);
00106           return 0;
00107         }
00108     }
00109
00110   /* If in cipher mode, handle necessary padding */
00111   if (action >= 0)
00112     {
00113       /* Handle remaining cipher block + padding */
00114       if (!EVP_CipherFinal_ex (ctx, outbuf, &outlen))
00115         {
00116           /* Error */
00117           EVP_CIPHER_CTX_cleanup (ctx);
00118           return 0;
00119         }
00120       /* Write remainign cipher block + padding*/
00121       fwrite (outbuf, sizeof (*inbuf), outlen, out);
00122       EVP_CIPHER_CTX_cleanup (ctx);
00123     }
00124
00125   /* Success */
00126   return 1;
00127 }
00128
00129 // Verify the entropy
00130 int
00131 check_entropy (void)
00132 {
00133   FILE *entropy_file = fopen ("/proc/sys/kernel/random/entropy_avail", "r");
00134   if (entropy_file == NULL)
00135     {
00136       perror ("Err: Cannot open entropy file");
00137       return -1;
00138     }
00139
00140   int entropy_value;
00141   if (fscanf (entropy_file, "%d", &entropy_value) != 1)
00142     {
00143       perror ("Err: Cannot estimate entropy");
00144       fclose (entropy_file);
00145       return -1;
00146     }
00147
00148   fclose (entropy_file);
00149   return entropy_value;
00150 }
00151
00152 // Add new entropy
00153 void
00154 add_entropy (void)
00155 {
00156   FILE *urandom = fopen ("/dev/urandom", "rb");
00157   if (urandom == NULL)
00158     {
00159       perror ("Err: Cannot open /dev/urandom");
00160       exit (EXIT_FAILURE);
00161     }
00162
00163   unsigned char random_data[32];
00164   size_t bytes_read = fread (random_data, 1, sizeof (random_data), urandom);
00165   fclose (urandom);
00166
00167   if (bytes_read != sizeof (random_data))
00168     {
00169       fprintf (stderr, "Err: Cannot read data\n");
00170       exit (EXIT_FAILURE);
00171     }
00172
00173   // Usa i dati casuali per aggiungere entropia
00174   RAND_add (random_data, sizeof (random_data),
00175             0.5); // 0.5 è un peso arbitrario
00176
00177   fprintf (stdout, "Entropy added successfully!\n");
```

```
00178 }
00179
00180 void
00181 generate_key (unsigned char *destination)
00182 {
00183   fprintf (stdout, "Generating a new key...\n");
00184
00185   // Why? Because if we try to create a large number of files there might not
00186   // be enough random bytes in the system to generate a key
00187   for (int i = 0; i < 10; i++)
00188     {
00189       int entropy = check_entropy ();
00190       if (entropy < 128)
00191         {
00192           fprintf (stderr, "WARN: not enough entropy, creating some...\n");
00193           add_entropy ();
00194         }
00195
00196       if (RAND_bytes (destination, 32) != 1)
00197         {
00198           fprintf (stderr, "Err: Cannot generate key\n");
00199           destination = NULL;
00200         }
00201
00202       if (strlen ((const char *)destination) == 32)
00203         break;
00204     }
00205
00206   if (is_valid_key (destination) == 0)
00207     {
00208       fprintf (stderr, "Err: Generated key is invalid\n");
00209       print_aes_key (destination);
00210       destination = NULL;
00211     }
00212 }
00213
00214 unsigned char *
00215 encrypt_string (unsigned char *plaintext, const char *key,
00216                 int *encrypted_key_len)
00217 {
00218   EVP_CIPHER_CTX *ctx;
00219   const EVP_CIPHER *cipher = EVP_aes_256_cbc ();
00220   unsigned char iv[AES_BLOCK_SIZE];
00221   memset (iv, 0, AES_BLOCK_SIZE);
00222
00223   ctx = EVP_CIPHER_CTX_new ();
00224   if (!ctx)
00225     {
00226       return NULL;
00227     }
00228
00229   EVP_EncryptInit_ex (ctx, cipher, NULL, (const unsigned char *)key, iv);
00230
00231   size_t plaintext_len = strlen ((const char *)plaintext);
00232   unsigned char ciphertext[plaintext_len + AES_BLOCK_SIZE];
00233   memset (ciphertext, 0, sizeof (ciphertext));
00234
00235   int len;
00236   EVP_EncryptUpdate (ctx, ciphertext, &len, plaintext, plaintext_len);
00237   EVP_EncryptFinal_ex (ctx, ciphertext + len, &len);
00238   EVP_CIPHER_CTX_free (ctx);
00239
00240   unsigned char *encoded_string = malloc (len * 2 + 1);
00241   if (!encoded_string)
00242     {
00243       return NULL;
00244     }
00245
00246   for (int i = 0; i < len; i++)
00247     {
00248       sprintf ((char *)&encoded_string[i * 2], "%02x", ciphertext[i]);
00249     }
00250   encoded_string[len * 2] = '\0';
00251
00252   *encrypted_key_len = len * 2;
00253   return encoded_string;
00254 }
00255
00256 unsigned char *
00257 decrypt_string (unsigned char *ciphertext, const char *key)
00258 {
00259   EVP_CIPHER_CTX *ctx;
00260   const EVP_CIPHER *cipher
00261       = EVP_aes_256_cbc (); // Choose the correct algorithm
00262   unsigned char iv[AES_BLOCK_SIZE];
00263   memset (iv, 0, AES_BLOCK_SIZE);
00264
```

```
00265   ctx = EVP_CIPHER_CTX_new ();
00266   EVP_DecryptInit_ex (ctx, cipher, NULL, (const unsigned char *)key, iv);
00267
00268   size_t decoded_len = strlen ((const char *)ciphertext);
00269
00270   unsigned char plaintext[decoded_len];
00271   memset (plaintext, 0, sizeof (plaintext));
00272
00273   int len;
00274   EVP_DecryptUpdate (ctx, plaintext, &len, ciphertext, (int)decoded_len);
00275   EVP_DecryptFinal_ex (ctx, plaintext + len, &len);
00276   EVP_CIPHER_CTX_free (ctx);
00277
00278   unsigned char *decrypted_string = (unsigned char *)malloc (decoded_len + 1);
00279   memcpy (decrypted_string, plaintext, decoded_len);
00280   decrypted_string[decoded_len] = '\0';
00281
00282   return decrypted_string;
00283 }
00284
00285 int
00286 is_valid_key (const unsigned char *key)
00287 {
00288   char str[33];
00289   memcpy (str, key, 32);
00290   str[32] = '\0';
00291   size_t key_length = strlen (str);
00292   return key_length != 32 ? 0 : 1;
00293 }
00294
00295 /*
00296 int rebuild_key(char *key, char *cert, char *dest){
00297     return -1;
00298 }*/
```

6.29 crypt-utils.h

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <sys/mman.h>
00005 #include <unistd.h>
00006
00007 #include <openssl/aes.h>
00008 #include <openssl/bio.h>
00009 #include <openssl/buffer.h>
00010 #include <openssl/evp.h>
00011 #include <openssl/rand.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
00024  *(copy)) unsigned char *key_str : C-string containing passphrase from which
00025  *key is derived 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.
00032  * it must be 33 bytes long to account for a \0 Return: void, if the generation
00033  * failed an error will be thrown
00034  */
00035 void generate_key (unsigned char *destination);
00036
00037 /*unsigned char* encrypt_string(unsigned char* plaintext, const char* key, int
00038  * *encrypted_len) Purpose: Encrypt a string with AES-256 Args: unsigned char*
00039  * plaintext   : The plaintext to be encrypted const char* key            : The
00040  * key for the encryption int *encrypted_len         : This will be filled with
00041  * the encrypted text length Return: The encrypted string + \0. On error null
00042  * is returned
00043  * */
00044 unsigned char *encrypt_string (unsigned char *plaintext, const char *key,
00045                                int *encrypted_len);
00046
```

```
00047 /*unsigned char* decrypt_string(unsigned char* base64_ciphertext, const char*
00048  * key); Purpose: Decrypt a string with AES-256 Args: unsigned char*
00049  * base64_ciphertext   : The cyphertext to be decrypted const char* key : The
00050  * key for the decryption Return: The decrypted string + \0. On error null is
00051  * returned
00052  * */
00053 unsigned char *decrypt_string (unsigned char *base64_ciphertext,
00054                                const char *key);
00055
00056 /*int is_valid_key(const unsigned char* key);
00057  * Purpose: Check if a AES-256 key is valid
00058  * Args: unsigned char* key   : The key to be checked
00059  * Return: 1 if the key is valid, 0 if it is invalid
00060  * */
00061 int is_valid_key (const unsigned char *key);
00062
00063 /*
00064 int rebuild_key(char *key, char *cert, char *dest);
00065 */
```

6.30 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
00013 key in userspace\n \ This will change in the future"); insert_key(key, cert,
00014 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 }*/
```

6.31 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);*/
```

6.32 tcfs_utils.c

```
00001 #include "tcfs_utils.h"
00002 #include "../crypt-utils/crypt-utils.h"
00003
00004 void
00005 get_user_name (char *buf, size_t size)
00006 {
00007   uid_t uid = geteuid ();
00008   struct passwd *pw = getpwuid (uid);
00009   if (pw)
00010     snprintf (buf, size, "%s", pw->pw_name);
00011   else
00012     perror ("Error: Could not retrieve username.\n");
00013 }
00014
```

```
00015 /* is_encrypted: returns 1 if file is encrypted, 0 otherwise*/
00016 int
00017 is_encrypted (const char *path)
00018 {
00019   int ret;
00020   char xattr_val[5];
00021   getxattr (path, "user.encrypted", xattr_val, sizeof (char) * 5);
00022   xattr_val[4] == '\n';
00023
00024   return strcmp (xattr_val, "true") == 0 ? 1 : 0;
00025 }
00026
00027 char *
00028 prefix_path (const char *path, const char *realpath)
00029 {
00030   if (path == NULL || realpath == NULL)
00031     {
00032       perror ("Err: path or realpath is NULL");
00033       return NULL;
00034     }
00035
00036   size_t len = strlen (path) + strlen (realpath) + 1;
00037   char *root_dir = malloc (len * sizeof (char));
00038
00039   if (root_dir == NULL)
00040     {
00041       perror ("Err: Could not allocate memory while in prefix_path");
00042       return NULL;
00043     }
00044
00045   if (strcpy (root_dir, realpath) == NULL)
00046     {
00047       perror ("strcpy: Cannot copy path");
00048       return NULL;
00049     }
00050   if (strcat (root_dir, path) == NULL)
00051     {
00052       perror ("strcat: in prefix_path cannot concatenate the paths");
00053       return NULL;
00054     }
00055   return root_dir;
00056 }
00057
00058 /* read_file: for debugging tempfiles */
00059 int
00060 read_file (FILE *file)
00061 {
00062   int c;
00063   int file_contains_something = 0;
00064   FILE *read = file; /* don't move original file pointer */
00065   if (read)
00066     {
00067       while ((c = getc (read)) != EOF)
00068         {
00069           file_contains_something = 1;
00070           putc (c, stderr);
00071         }
00072     }
00073   if (!file_contains_something)
00074     fprintf (stderr, "file was empty\n");
00075   rewind (file);
00076   /* fseek(tmpf, offset, SEEK_END); */
00077   return 0;
00078 }
00079 /* Get the xattr value describing the key of a file
00080  * return 1 on success else 0
00081  * */
00082 int
00083 get_encrypted_key (char *filepath, unsigned char *encrypted_key)
00084 {
00085   printf ("\tGet Encrypted key for file %s\n", filepath);
00086   if (is_encrypted (filepath) == 1)
00087     {
00088       printf ("\t\tencrypted file\n");
00089
00090       FILE *src_file = fopen (filepath, "r");
00091       if (src_file == NULL)
00092         {
00093           fclose (src_file);
00094           perror ("Could not open the file to get the key");
00095           return -errno;
00096         }
00097       int src_fd;
00098       src_fd = fileno (src_file);
00099       if (src_fd == -1)
00100         {
00101           fclose (src_file);
```

```
00102           perror ("Could not get fd for the file");
00103           return -errno;
00104         }
00105
00106       if (fgetxattr (src_fd, "user.key", encrypted_key, 33) != -1)
00107         {
00108           fclose (src_file);
00109           return 1;
00110         }
00111     }
00112   return 0;
00113 }
00114 /*For debugging only*/
00115 void
00116 print_aes_key (unsigned char *key)
00117 {
00118   printf ("AES HEX:%s -> ", key);
00119   for (int i = 0; i < 32; i++)
00120     {
00121       printf ("%02x", key[i]);
00122     }
00123   printf ("\n");
00124 }
```

6.33 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