TCFS

0.2

Generated on Tue Nov 28 2023 15:14:43 for TCFS by Doxygen 1.9.8

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

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_incoming_messages (void *queue_id)

Handle the case described in note

Member handle_outgoing_messages (void *queue_id)

Remove this function from the code

Member handle_termination (int signum)

: Implement remove_queue() to clear and delete the queue

Member HOST []

This should be passed as a parameter to the daemon

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

Deprecated List

Member get_encrypted_key (char *filepath, unsigned char *encrypted_key)

There is no use currenly for this function. It was once used for debugging

Member print_aes_key (unsigned char *key)

There is currently no use for this function

Member read_file (FILE *file)

Currently it has no use

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

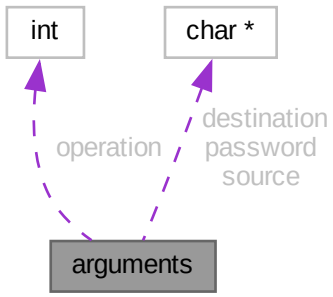
Here is a list of all documented files with brief descriptions:

Chapter 6

Class Documentation

6.1 arguments Struct Reference

Collaboration diagram for arguments:



Public Attributes

- int operation
- char * source
- char * destination
- char * password

6.1.1 Detailed Description

Definition at line 20 of file user_tcfs.c.

6.1.2 Member Data Documentation

6.1.2.1 destination

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

Definition at line 736 of file tcfs.c.

6.1.2.2 operation

```
int arguments::operation
```

Definition at line 22 of file user_tcfs.c.

6.1.2.3 password

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

Definition at line 737 of file tcfs.c.

6.1.2.4 source

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

Definition at line 735 of file tcfs.c.

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

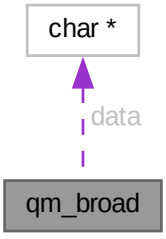
- user/user_tcfs.c
- userspace-module/tcfs.c

6.2 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

6.2.1 Detailed Description

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

Definition at line 86 of file common.h.

6.2.2 Member Data Documentation

6.2.2.1 data

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

The data that is broadcasted.

Definition at line 87 of file common.h.

Referenced by string_to_struct(), and struct_to_json().

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

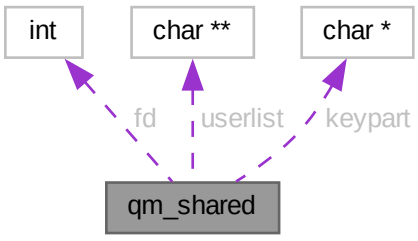
- daemon/daemon_utils/common.h

6.3 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

6.3.1 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 75 of file common.h.

6.3.2 Member Data Documentation

6.3.2.1 fd

```
int qm_shared::fd
```

The file descriptor of the shared file.

Definition at line 76 of file common.h.

Referenced by string_to_struct(), and struct_to_json().

6.3.2.2 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 78 of file common.h.

Referenced by string_to_struct(), and struct_to_json().

6.3.2.3 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 77 of file common.h.

Referenced by string_to_struct(), and struct_to_json().

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

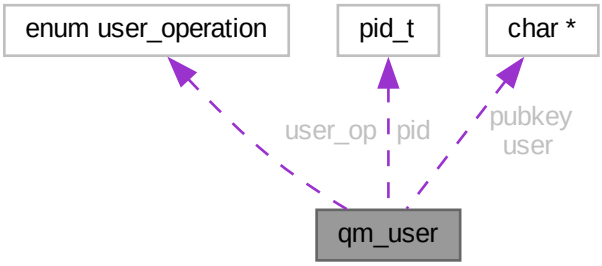
- daemon/daemon_utils/common.h

6.4 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

6.4.1 Detailed Description

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

Definition at line 61 of file common.h.

6.4.2 Member Data Documentation

6.4.2.1 pid

`pid_t qm_user::pid`

The process ID of the user.

Definition at line 63 of file common.h.

Referenced by get_user_by_name(), get_user_by_pid(), handle_outgoing_messages(), insert(), remove_by_user(), string_to_struct(), and struct_to_json().

6.4.2.2 pubkey

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

The public key of the user.

Definition at line 65 of file common.h.

Referenced by handle_outgoing_messages(), string_to_struct(), and struct_to_json().

6.4.2.3 user

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

The username of the user.

Definition at line 64 of file common.h.

Referenced by get_user_by_name(), get_user_by_pid(), handle_outgoing_messages(), insert(), remove_by_pid(), string_to_struct(), and struct_to_json().

6.4.2.4 user_op

```
user_operation qm_user::user_op
```

The operation that the user wants to perform.

Definition at line 62 of file common.h.

Referenced by handle_outgoing_messages(), string_to_struct(), and struct_to_json().

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

- daemon/daemon_utils/common.h

Chapter 7

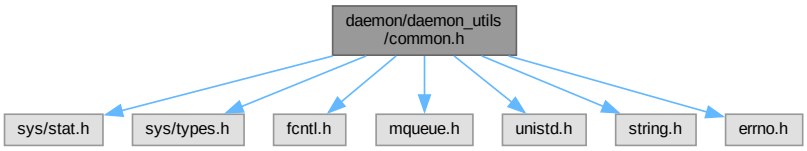
File Documentation

7.1 daemon/daemon_utils/common.h File Reference

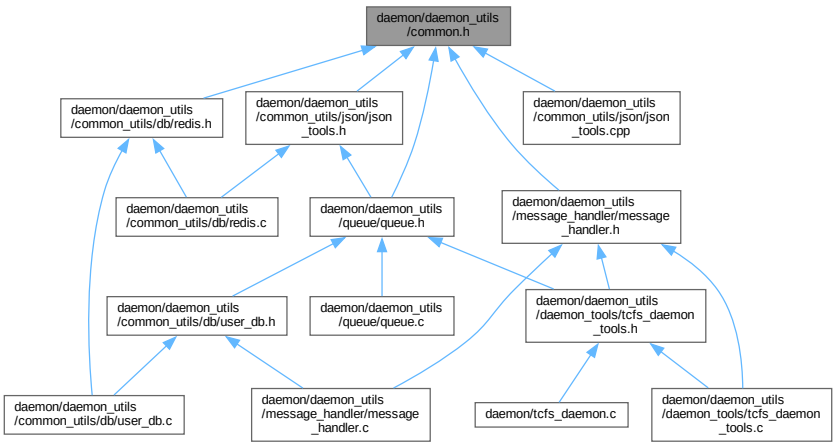
This file contains some common definitions and structs used by the daemon.

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

Include dependency graph for common.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct qm_user

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

  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.
  .
- struct qm_broad

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

Macros

- #define MAX_QM_SIZE 512

  Maximum size of a message in bytes. This definition is marked as internal and should not be used directly by the user.
- #define MAX_QM_N 100

  Maximum number of messages that can be stored on a queue. This definition is marked as internal and should not be used directly by the user.

Typedefs

- typedef enum qm_type **qm_type**
- typedef enum user_operation **user_operation**
- typedef struct qm_user **qm_user**
- typedef struct qm_shared **qm_shared**
- typedef struct qm_broad **qm_broad**

Enumerations

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

  Describes the type of a given message.
  USER refers to qm_user struct
  SHARED refers to user_operation struct
  BROADCAST refers to qm_broad struct
  QM_TYPE_UNDEFINED is set if there was an error and we cannot determinate the type of the struct.
- enum user_operation { REGISTER = 0 , UNREGISTER = 1 }

  Describes the operation that a user can perform.
  REGISTER means that the user wants to register to the system.
  UNREGISTER means that the user wants to unregister from the system.

7.1.1 Detailed Description

This file contains some common definitions and structs used by the daemon.

Definition in file common.h.

7.1.2 Macro Definition Documentation

7.1.2.1 MAX_QM_N

```
#define MAX_QM_N 100
```

Maximum number of messages that can be stored on a queue. This definition is marked as internal and should not be used directly by the user.

Definition at line 25 of file common.h.

7.1.2.2 MAX_QM_SIZE

```
#define MAX_QM_SIZE 512
```

Maximum size of a message in bytes. This definition is marked as internal and should not be used directly by the user.

Definition at line 19 of file common.h.

7.1.3 Enumeration Type Documentation

7.1.3.1 qm_type

```
enum qm_type
```

Describes the type of a given message.
USER refers to qm_user struct
SHARED refers to user_operation struct
BROADCAST refers to qm_broad struct
QM_TYPE_UNDEFINED is set if there was an error and we cannot determinate the type of the struct.

Enumerator

USER	Refers to type qm_user
SHARED	Refers to type qm_shared
BROADCAST	Refers to type qm_broad
QM_TYPE_UNDEFINED	This is set in case of error, it means that the structure it is referring to is invalid

Definition at line 38 of file common.h.

7.1.3.2 user_operation

```
enum user_operation
```

Describes the operation that a user can perform.
REGISTER means that the user wants to register to the system.
UNREGISTER means that the user wants to unregister from the system.

Enumerator

REGISTER	User wants to register
UNREGISTER	User wants to unregister

Definition at line 51 of file common.h.

7.2 common.h

Go to the documentation of this file.

```
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
00019 #define MAX_QM_SIZE 512
00025 #define MAX_QM_N 100
00026
00027 #ifndef QUEUE_STRUCTS
00028 #define QUEUE_STRUCTS
00029
00038 typedef enum qm_type{
00039     USER = 0,
00040     SHARED = 1,
00041     BROADCAST = 2,
00042     QM_TYPE_UNDEFINED = -1,
00043 } qm_type;
00044
00051 typedef enum user_operation{
00052     REGISTER = 0,
00053     UNREGISTER = 1,
00054 } user_operation;
00055
00061 typedef struct qm_user {
00062     user_operation user_op;
00063     pid_t pid;
00064     char *user;
00065     char *pubkey;
00066 } qm_user;
00067
00075 typedef struct qm_shared {
00076     int fd;
```

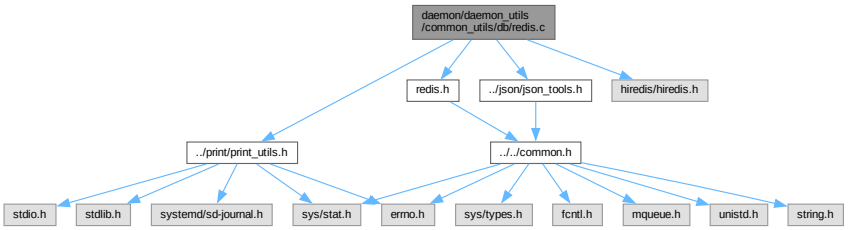
```
00077     char **userlist;
00078     char *keypart;
00079 } qm_shared;
00080
00086 typedef struct qm_broad {
00087     char *data;
00088 } qm_broad;
00089
00090 #endif
```

7.3 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"

  The host address of the redis DB. This variable is marked as internal and should not be used by the user.
- redisContext * context

  Pointer to the context of Redis DB.

7.3.1 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.

7.3.2 Macro Definition Documentation

7.3.2.1 PORT

```
#define PORT 6380
```

The port of the redis DB.

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

Todo This should be passed as a parameter to the daemon

Definition at line 27 of file redis.c.

7.3.3 Function Documentation

7.3.3.1 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:

7.3.3.2 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

name	The key of the row

Returns

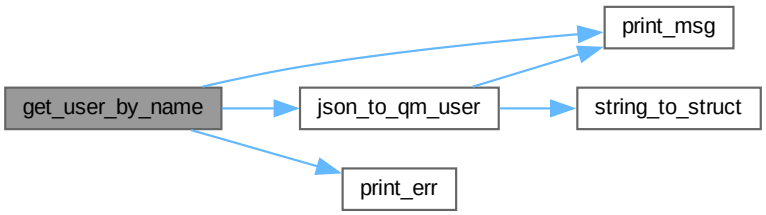
qm_user∗ A pointer to the allocated qm_user∗ struct

Definition at line 165 of file redis.c.

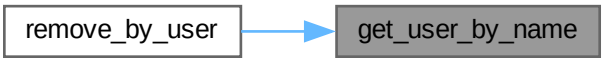
References context, json_to_qm_user(), qm_user::pid, print_err(), print_msg(), and qm_user::user.

Referenced by remove_by_user().

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.3.3 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	The key of the row

Returns

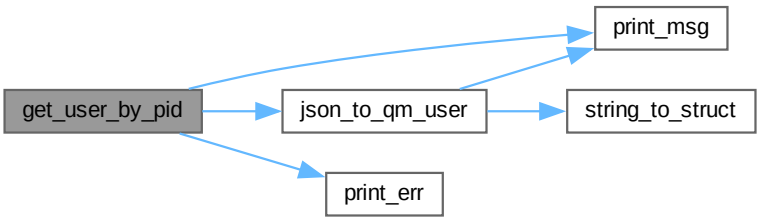
qm_user* A pointer to the allocated qm_user* struct

Definition at line 122 of file redis.c.

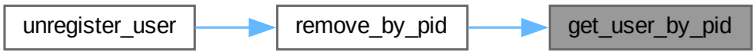
References context, json_to_qm_user(), qm_user::pid, print_err(), print_msg(), and qm_user::user.

Referenced by remove_by_pid().

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.3.4 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, HOST, PORT, and print_err().

Referenced by register_user().

Here is the call graph for this function:

Here is the caller graph for this function:

7.3.3.5 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

user	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

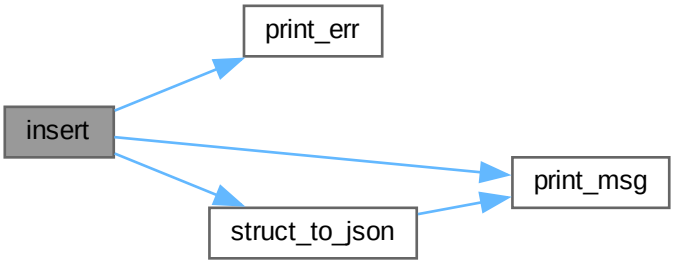
If an error is thrown it will be printed by print_err() function

Definition at line 211 of file redis.c.

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

Referenced by register_user().

Here is the call graph for this function:



Here is the caller graph for this function:

7.3.3.6 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

json	the json string representing the qm_user struct

Returns

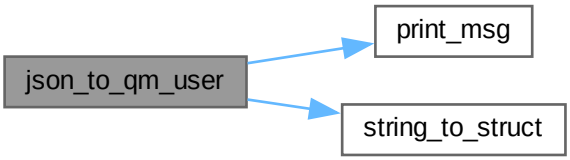
qm_user* A pointer to the allocated qm_user* struct

Definition at line 104 of file redis.c.

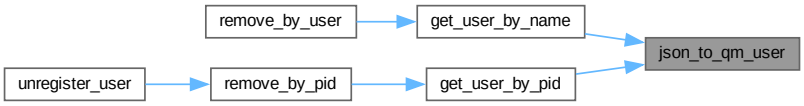
References print_msg(), and string_to_struct().

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:



7.3.3.7 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

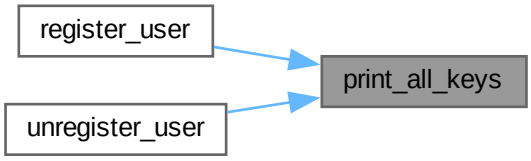
Definition at line 42 of file redis.c.

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:



7.3.3.8 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	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.

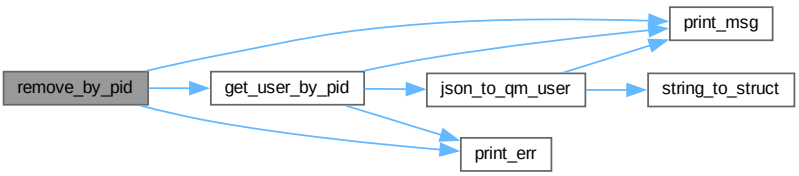
If an error is thrown it will be printed using the print_err() function

Definition at line 256 of file redis.c.

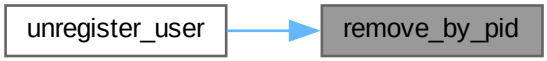
References context, get_user_by_pid(), print_err(), print_msg(), and qm_user::user.

Referenced by unregister_user().

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.3.9 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

name	The key

Returns

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

See also

print_err

Note

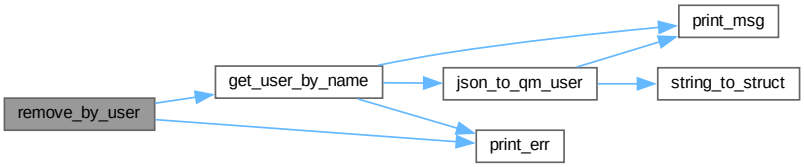
Will also remove the corresponding entry by PID

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:



7.3.4 Variable Documentation

7.3.4.1 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().

7.3.4.2 HOST

```
HOST = "127.0.0.1"
```

The host address of the redis DB. This variable is marked as internal and should not be used by the user.

Todo This should be passed as a parameter to the daemon

Definition at line 20 of file redis.c.

Referenced by init_context().

7.4 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 }
```

7.5 redis.h

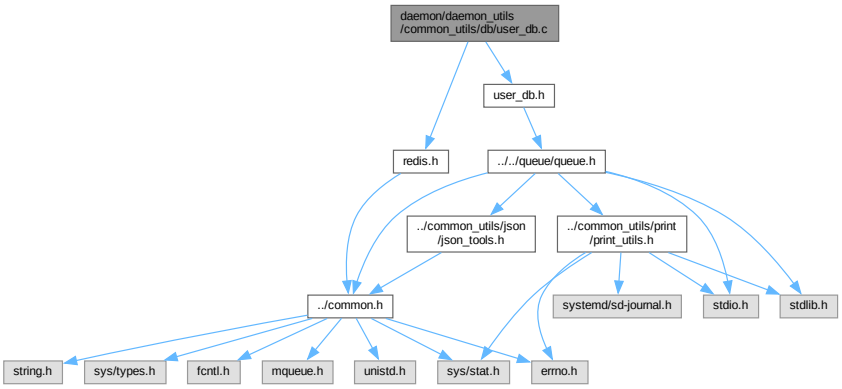
```
00001 #include "../../common.h"
00002
00003 void print_all_keys ();
00004
00005 int init_context ();
00006
00007 qm_user *json_to_qm_user (char *json);
00008
00009 qm_user *get_user_by_pid (pid_t pid);
00010
00011 qm_user *get_user_by_name (const char *name);
00012
00013 int insert (qm_user *user);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

7.6 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.

7.6.1 Detailed Description

This file contains the functions to interact with the database.

Definition in file user_db.c.

7.6.2 Function Documentation

7.6.2.1 disconnect_db()

```
void disconnect_db (
            void  )
```

Free the context of the DB.

Parameters

void	

Returns

void

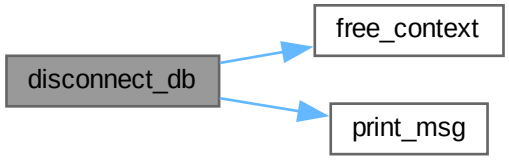
Note

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:



7.6.2.2 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

user_msg	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,

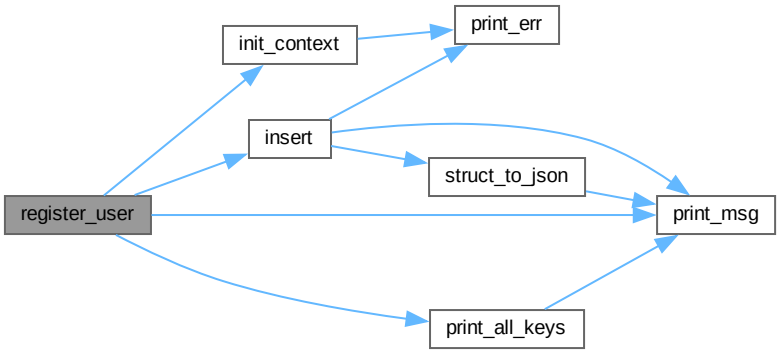
See also

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:



7.6.2.3 unregister_user()

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

Remove a user from the DB.

Parameters

pid	the key

Returns

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

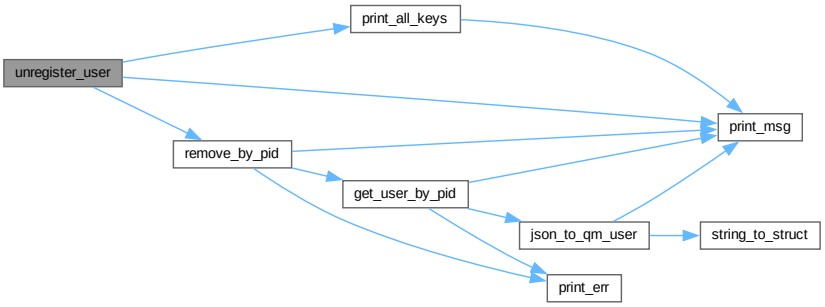
See also

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:



7.7 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 }
```

7.8 user_db.h

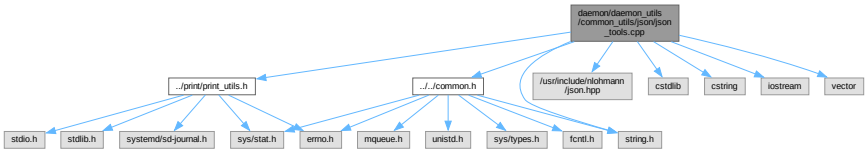
```
00001 #include "../../queue/queue.h"
00002
00003 int register_user (qm_user *user_msg);
00004 int unregister_user (pid_t pid);
00005 void disconnect_db (void);
```

7.9 daemon/daemon_utils/common_utils/json/json_tools.cpp File Reference

This file provides function to cast either qm_user, qm_shared or qm_broad to a json string and vice versa.

```
#include "../../common.h"
#include "../print/print_utils.h"
#include "/usr/include/nlohmann/json.hpp"
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <string.h>
#include <vector>
```

Include dependency graph for json_tools.cpp:



Functions

- char * struct_to_json (qm_type qmt, void *q_mess)

  Cast a qm_user, qm_shared or qm_broad struct to a json string representing the struct.
- void * string_to_struct (const char *json_string, qm_type *type)

  Cast a json string to a struct.

7.9.1 Detailed Description

This file provides function to cast either qm_user, qm_shared or qm_broad to a json string and vice versa.

Note

All the functions here are C++ functions, so we could use nlohmann-json library

Learn more on `nlohmann-json`

Definition in file json_tools.cpp.

7.9.2 Function Documentation

7.9.2.1 string_to_struct()

```
void * string_to_struct (
            const char * json_string,
            qm_type * type )
```

Cast a json string to a struct.

Parameters

json_string	The string containing the json that represents the struct
type	Will be set to the type of the struct

Returns

void* This is the actual allocated structure, casted to void

Note

To cast the returned param to the structure you probably need to use a switch(type) and cast it to a struct

See also

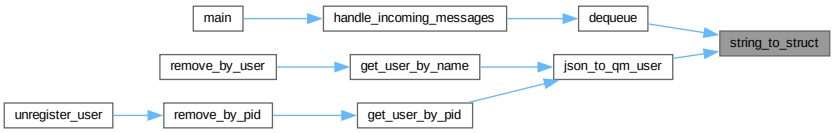
common.h

Definition at line 92 of file json_tools.cpp.

References BROADCAST, qm_broad::data, qm_shared::fd, qm_shared::keypart, qm_user::pid, qm_user::pubkey, QM_TYPE_UNDEFINED, SHARED, USER, qm_user::user, qm_user::user_op, and qm_shared::userlist.

Referenced by dequeue(), and json_to_qm_user().

Here is the caller graph for this function:



7.9.2.2 struct_to_json()

```
char * struct_to_json (
            qm_type qmt,
            void * q_mess )
```

Cast a qm_user, qm_shared or qm_broad struct to a json string representing the struct.

Parameters

qmt	

See also

common.h

Parameters

q_mess	The structure from which the json will be built

Returns

char* The json string

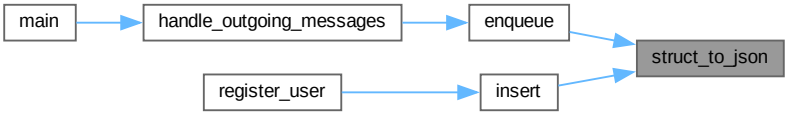
Definition at line 27 of file json_tools.cpp.

References BROADCAST, qm_broad::data, qm_shared::fd, qm_shared::keypart, qm_user::pid, print_msg(), qm_user::pubkey, REGISTER, SHARED, UNREGISTER, USER, qm_user::user, qm_user::user_op, and qm_shared::userlist.

Referenced by enqueue(), and insert().

Here is the call graph for this function:

Here is the caller graph for this function:



7.10 json_tools.cpp

Go to the documentation of this file.

```
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 }
```

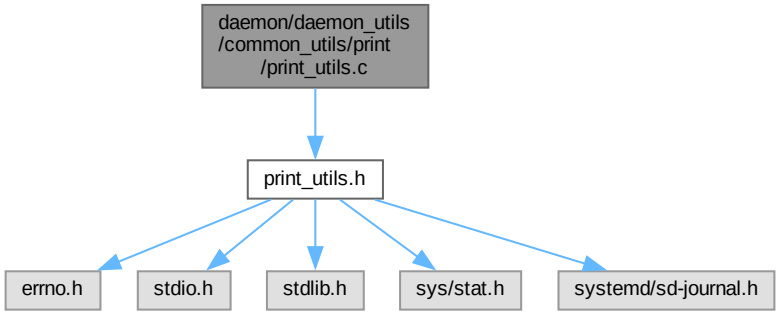
7.11 json_tools.h

```
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);
```

7.12 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.

7.12.1 Detailed Description

This file defines some QoL functions.

Definition in file print_utils.c.

7.12.2 Function Documentation

7.12.2.1 print_debug()

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

Format and print data as a debug.

Parameters

format	the string that will formatted and printed
...	Print optional ARGUMENT(s) according to format

Returns

void

Note

Will also log using systemD

"DEBUG=" will be prepended to format

Definition at line 144 of file print_utils.c.

7.12.2.2 print_err()

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

Format and print data as an error.

Parameters

format	the string that will formatted and printed
...	Print optional ARGUMENT(s) according to format

Returns

void

Note

Will also log using systemD

"ERROR=" will be prepended to format

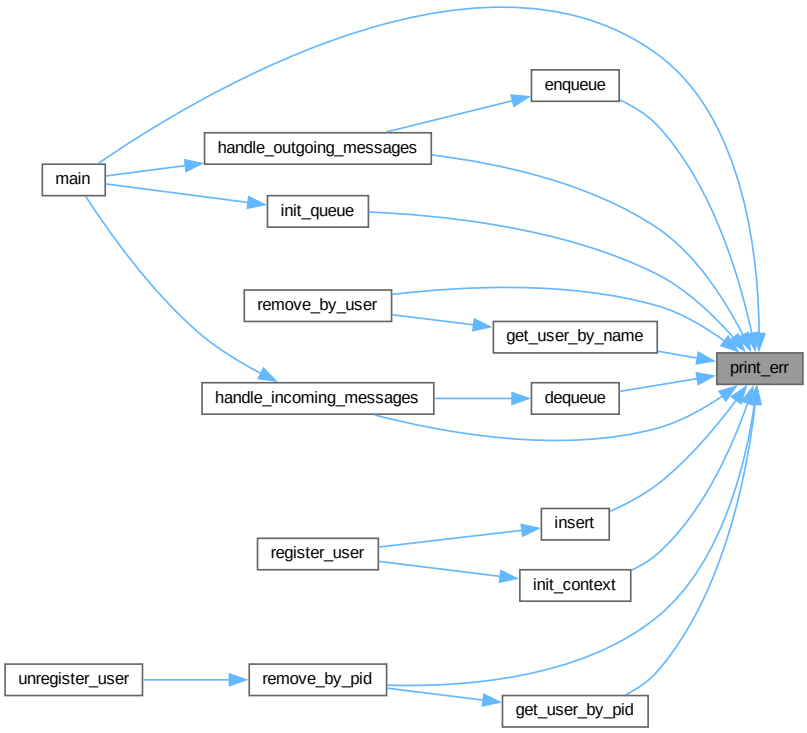
"Err_Numebr:d" will be appended to the formatted string describing the error number

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

Definition at line 78 of file print_utils.c.

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

Here is the caller graph for this function:



7.12.2.3 print_msg()

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

Format and print data as a message.

Parameters

format	the string that will formatted and printed
...	Print optional ARGUMENT(s) according to format

Returns

void

Note

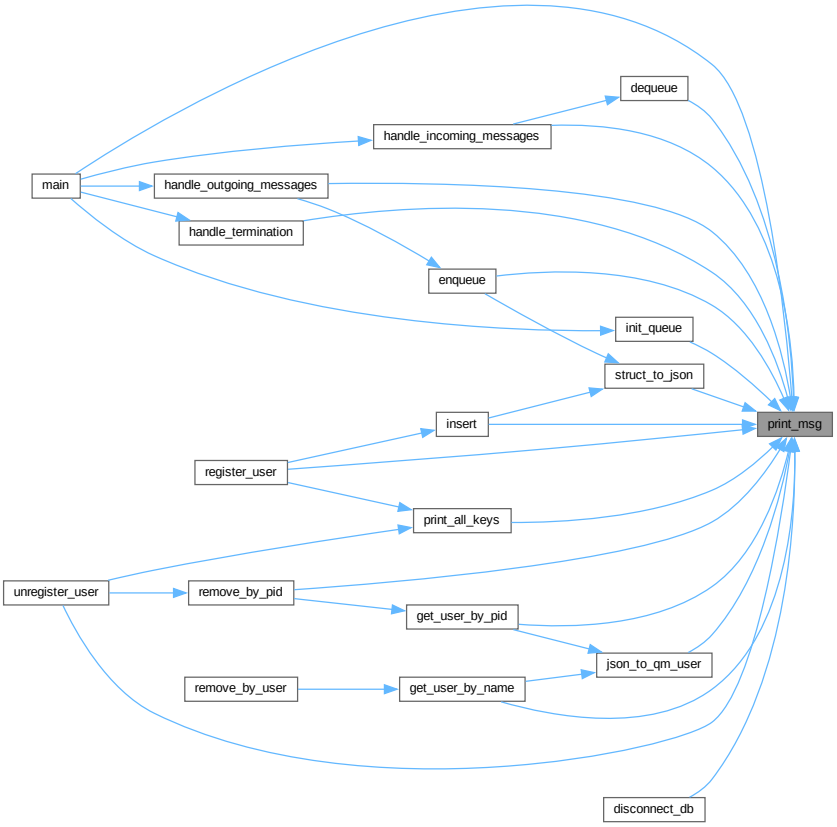
Will also log using systemD

"MESSAGE=" will be prepended to format

Definition at line 100 of file print_utils.c.

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

Here is the caller graph for this function:



7.12.2.4 print_warn()

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

Format and print data as a waring.

Parameters

format	the string that will formatted and printed
...	Print optional ARGUMENT(s) according to format

Returns

void

Note

Will also log using systemD

"WARNING=" will be prepended to format

Definition at line 122 of file print_utils.c.

7.12.3 Variable Documentation

7.12.3.1 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.

7.13 print_utils.c

Go to the documentation of this file.

```
00001 #include "print_utils.h"
00002
00014 int cleared = 0;
00015
00023 void
00024 log_message (const char *log)
00025 {
00026   printf ("%s\n", log);
00027   // Path of the log folder and log file
00028   const char *logFolder = "/var/log/tcfs";
00033   const char *logFile = "/var/log/tcfs/log.txt";
00034
00035   // Check if the folder exists, otherwise create it
00036   struct stat st;
00037   if (stat (logFolder, &st) == -1)
00038     {
00039       mkdir (logFolder, 0700);
00040     }
00041
00042   FILE *file;
00043   if (cleared == 0)
00044     {
00045       cleared = 1;
00046       file = fopen (logFile, "w");
00047     }
00048   else
00049     {
00050       file = fopen (logFile, "a");
00051     }
00052
00053   // Open the log file in append mode
00054   if (file == NULL)
```

```
00055     {
00056       perror ("Error opening the log file");
00057     }
00058
00059   // Write the message to the log file
00060   fprintf (file, "%s\n", log);
00061
00062   // Close the file
00063   fclose (file);
00064 }
00065
00077 void
00078 print_err (const char *format, ...)
00079 {
00080   va_list args;
00081   va_start (args, format);
00082   char buffer[1024];
00083   vsnprintf (buffer, sizeof (buffer), format, args);
00084   va_end (args);
00085
00086   log_message (buffer);
00087
00088   sd_journal_print (LOG_ERR, "ERROR=%s Err_Number:%d -> %s", buffer, errno,
00089                     strerror (errno));
00090 }
00099 void
00100 print_msg (const char *format, ...)
00101 {
00102   va_list args;
00103   va_start (args, format);
00104   char buffer[1024];
00105   vsnprintf (buffer, sizeof (buffer), format, args);
00106   va_end (args);
00107
00108   log_message (buffer);
00109
00110   sd_journal_send ("MESSAGE=%s", buffer, NULL);
00111 }
00112
00121 void
00122 print_warn (const char *format, ...)
00123 {
00124   va_list args;
00125   va_start (args, format);
00126   char buffer[1024];
00127   vsnprintf (buffer, sizeof (buffer), format, args);
00128   va_end (args);
00129
00130   log_message (buffer);
00131
00132   sd_journal_print (LOG_WARNING, "WARNING=%s", buffer, NULL);
00133 }
00134
00143 void
00144 print_debug (const char *format, ...)
00145 {
00146   va_list args;
00147   va_start (args, format);
00148   char buffer[1024];
00149   vsnprintf (buffer, sizeof (buffer), format, args);
00150   va_end (args);
00151
00152   log_message (buffer);
00153
00154   sd_journal_print (LOG_DEBUG, "DEBUG=%s", buffer, NULL);
00155 }
```

7.14 print_utils.h

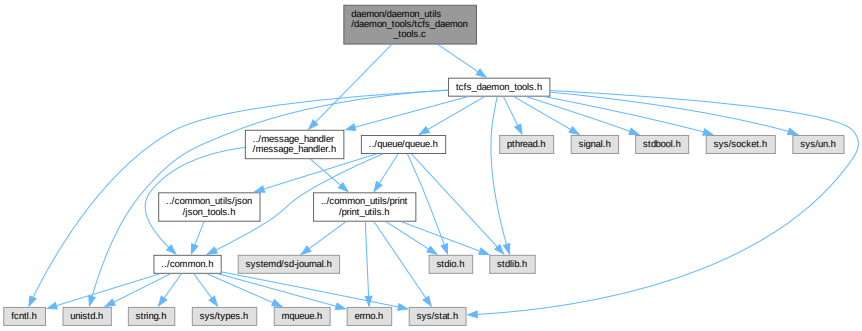
```
00001 #include <errno.h>
00002 #include <stdio.h>
00003 #include <stdlib.h>
00004 #include <sys/stat.h>
00005 #include <systemd/sd-journal.h>
00006
00007 void print_err (const char *format, ...);
00008 void print_msg (const char *format, ...);
00009 void print_warn (const char *format, ...);
00010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

7.15 daemon/daemon_utils/daemon_tools/tcfs_daemon_tools.c File Reference

This file contains the logic for handling the various requests and responses on the message queue.

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

Include dependency graph for tcfs_daemon_tools.c:



Functions

- void * handle_incoming_messages (void *queue_id)

  Dequeue the latest message from the queue and handle it.
- void * handle_outgoing_messages (void *queue_id)

  Test if the daemon is working by sending some messages.

7.15.1 Detailed Description

This file contains the logic for handling the various requests and responses on the message queue.

Definition in file tcfs_daemon_tools.c.

7.15.2 Function Documentation

7.15.2.1 handle_incoming_messages()

```
void * handle_incoming_messages (
            void * queue_id )
```

Dequeue the latest message from the queue and handle it.

Parameters

queue_id	Pointer to mqd_t message queue descriptor

Returns

void

Note

This function must never return. In case of its return the daemon will stall

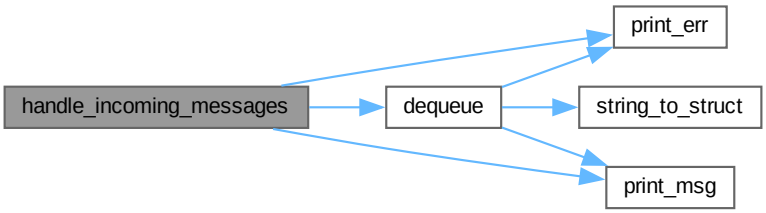
Todo Handle the case described in note

Definition at line 19 of file tcfs_daemon_tools.c.

References BROADCAST, dequeue(), print_err(), print_msg(), QM_TYPE_UNDEFINED, SHARED, and USER.

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

7.15.2.2 handle_outgoing_messages()

```
void * handle_outgoing_messages (
            void * queue_id )
```

Test if the daemon is working by sending some messages.

Parameters

queue_id	Pointer to mqd_t message queue descriptor

Returns

void

Note

THIS FUNCTION IS HERE JUST TEMPORARILY. WILL BE REMOVED, THIS IS NOT WHAT WE WANT THE DAEMON TO DO. PLEASE IGNORE

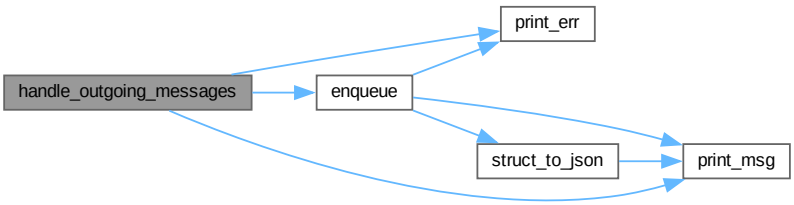
Todo Remove this function from the code

Definition at line 66 of file tcfs_daemon_tools.c.

References enqueue(), qm_user::pid, print_err(), print_msg(), qm_user::pubkey, REGISTER, UNREGISTER, USER, qm_user::user, and qm_user::user_op.

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

7.16 tcfs_daemon_tools.c

Go to the documentation of this file.

```
00001 #include "tcfs_daemon_tools.h"
00002 #include "../message_handler/message_handler.h"
00003
00018 void *
00019 handle_incoming_messages (void *queue_id)
00020 {
00021   qm_type qmt;
00022   qm_user *user_msg;
00023   qm_shared *shared_msg;
00024   qm_broad *broadcast_msg;
00025
00026   print_msg ("Starting handler for incoming messages");
00027   void *tmp_struct;
00028   while (1)
00029     {
00030       tmp_struct = dequeue (*(mqd_t *)queue_id, &qmt);
00031       switch (qmt)
00032         {
00033         case USER:
00034           print_msg ("Handling user message");
00035           user_msg = (qm_user *)tmp_struct;
00036           handle_user_message (user_msg);
00037           break;
00038         case SHARED:
00039           print_msg ("Handling shared message");
00040           shared_msg = (qm_shared *)tmp_struct;
00041           // handle_shared_message()
00042           break;
00043         case BROADCAST:
00044           print_msg ("Handling broadcast message");
00045           broadcast_msg = (qm_broad *)tmp_struct;
00046           // handle_broadcast_message()
00047           break;
00048         case QM_TYPE_UNDEFINED:
00049           print_err ("Received un unknown message type, skipping...");
00050           break;
00051         }
00052       free (tmp_struct);
00053     }
00054   return NULL;
00055 }
00056
00065 void *
00066 handle_outgoing_messages (void *queue_id)
00067 {
00068   print_msg ("Handling outgoing messages");
00069   // sleep(1);
00070
00071   char s1[] = "TEST";
00072   char s2[] = "pubkey";
00073
00074   struct qm_user test_msg;
00075   test_msg.user_op = REGISTER;
00076   test_msg.pid = 104;
00077   test_msg.user = s1;
00078   test_msg.pubkey = s2;
00079
00080   print_msg ("Enqueueing test registration...");
00081   int res = enqueue (*(mqd_t *)queue_id, USER, (void *)&test_msg);
00082   print_msg ("TEST message send with result %d", res);
00083
00084   if (res != 1)
00085     {
00086       print_err ("enqueue err ");
00087     }
00088
00089   struct qm_user test_msg2;
00090   test_msg2.user_op = UNREGISTER;
00091   test_msg2.pid = 104;
00092   test_msg2.user = "";
00093   test_msg2.pubkey = "";
00094
00095   sleep (3);
00096
00097   print_msg ("Enqueueing test remove...");
00098   res = enqueue (*(mqd_t *)queue_id, USER, (void *)&test_msg2);
00099   print_msg ("TEST message send with result %d", res);
00100
00101   if (res != 1)
00102     {
00103       print_err ("enqueue err ");
00104     }
```

```
00105
00106   return NULL;
00107 }
00108
00109 /*
00110  *
00111 void* monitor_termination(void* queue_id) {
00112     while (1) {
00113         pthread_mutex_lock(&terminate_mutex);
00114         if (terminate) {
00115             pthread_mutex_unlock(&terminate_mutex);
00116             break;
00117         }
00118         pthread_mutex_unlock(&terminate_mutex);
00119         sleep(1);
00120     }
00121     print_err("Terminating threads");
00122     remove_empty_queue(*(int *)queue_id);
00123     return NULL;
00124 }*/
```

7.17 tcfs_daemon_tools.h

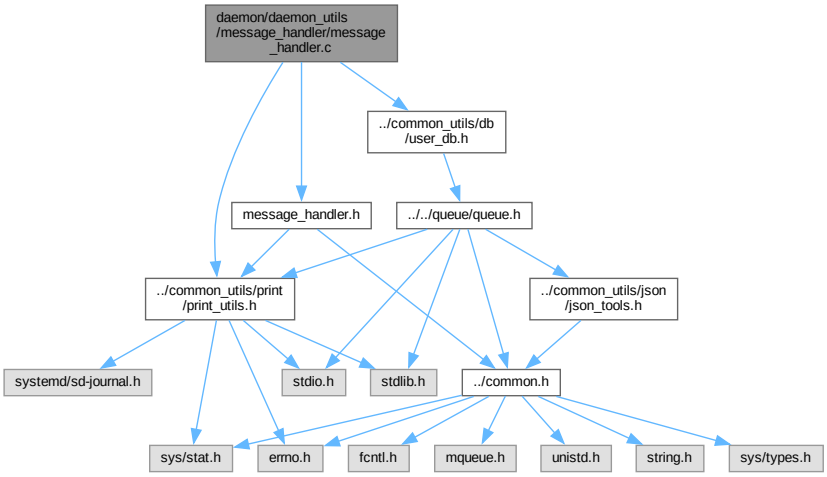
```
00001 #include "../message_handler/message_handler.h"
00002 #include "../queue/queue.h"
00003 #include <fcntl.h>
00004 #include <pthread.h>
00005 #include <signal.h>
00006 #include <stdbool.h>
00007 #include <stdlib.h>
00008 #include <sys/socket.h>
00009 #include <sys/stat.h>
00010 #include <sys/un.h>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

7.18 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/print/print_utils.h"
```

Include dependency graph for message_handler.c:



Functions

- int handle_user_message (qm_user *user_msg)

7.18.1 Detailed Description

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

Definition in file message_handler.c.

7.18.2 Function Documentation

7.18.2.1 handle_user_message()

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

Definition at line 11 of file message_handler.c.

7.19 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/print/print_utils.h"
00004
00010 int
00011 handle_user_message (qm_user *user_msg)
00012 {
00013   if (user_msg->user_op == REGISTER)
00014     {
```

```
00015       register_user (user_msg);
00016     }
00017   else if (user_msg->user_op == UNREGISTER)
00018     {
00019       unregister_user (user_msg->pid);
00020       // TODO: next line is a test, remove it
00021       free_context ();
00022     }
00023   else
00024     {
00025       print_err ("Unknown user operation %d", user_msg->user_op);
00026       return 0;
00027     }
00028
00029   return 1;
00030 }
```

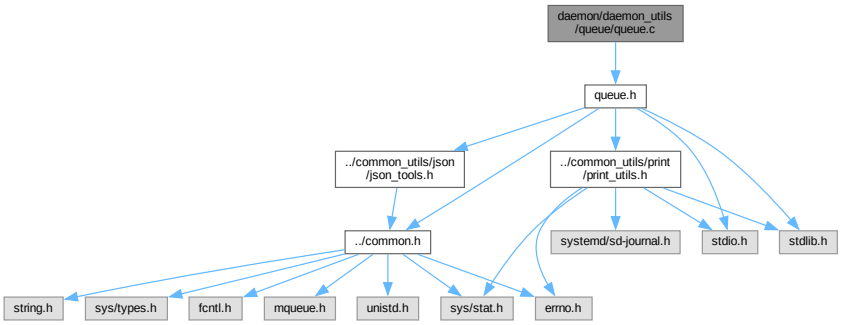
7.20 message_handler.h

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

7.21 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

  This defines the max size of a message on the queue. This definition is marked as internal and should not be used directly by the user.
- #define MQUEUE_N 256;

  Max number of messages on a queue.

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.*

7.21.1 Detailed Description

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

Definition in file queue.c.

7.21.2 Macro Definition Documentation

7.21.2.1 MESSAGE_BUFFER_SIZE

```
#define MESSAGE_BUFFER_SIZE 256
```

This defines the max size of a message on the queue. This definition is marked as internal and should not be used directly by the user.

Definition at line 13 of file queue.c.

7.21.2.2 MQUEUE_N

```
#define MQUEUE_N 256;
```

Max number of messages on a queue.

Definition at line 18 of file queue.c.

7.21.3 Function Documentation

7.21.3.1 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

queue↩ _d	Message queue descriptor type
qmt	Pointer to a struct indicating the type of the returned parameter

See also

qm_type

Returns

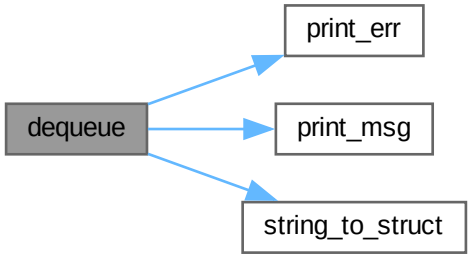
A pointer to a structure containing the structured message data. If an error occurs NULL is returned

Definition at line 94 of file queue.c.

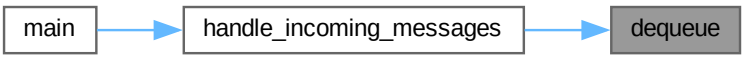
References MAX_QM_SIZE, print_err(), print_msg(), and string_to_struct().

Referenced by handle_incoming_messages().

Here is the call graph for this function:



Here is the caller graph for this function:



7.21.3.2 enqueue()

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

Enqueues a message void* message on the queue.

Parameters

queue↩ _d	Message queue descriptor type
qmt	enum describing the type of the message.

See also

qm_type

Parameters

q_mess	Actual message, this must be either qm_user, qm_shared qm_broad

See also

qm_user

qm_shared

qm_broad

Returns

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

See also

print_err

Note

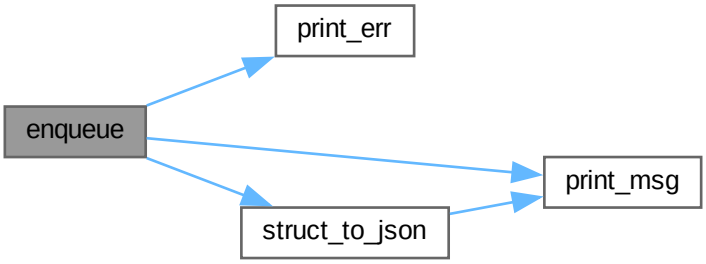
The structure representing the message will be casted to a json and then it will be enqueued

Definition at line 66 of file queue.c.

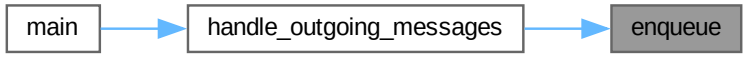
References print_err(), print_msg(), and struct_to_json().

Referenced by handle_outgoing_messages().

Here is the call graph for this function:



Here is the caller graph for this function:



7.21.3.3 init_queue()

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

Initialize the message queue.

Parameters

queue	the path of the queue file

Returns

mqd_t Message queue descriptor

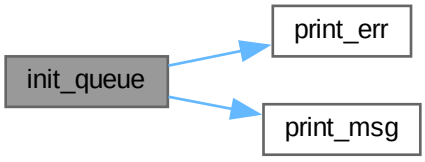
Todo Define permissions for mq_open

Definition at line 27 of file queue.c.

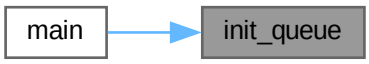
References MAX_QM_N, MAX_QM_SIZE, print_err(), and print_msg().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



7.22 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
00065 int
00066 enqueue (mqd_t queue_d, qm_type qmt, void *q_mess)
00067 {
00068   const char *qm_json = struct_to_json (qmt, q_mess);
00069
00070   if (mq_send (queue_d, qm_json, strlen (qm_json) + 1, 0) == -1)
00071     {
00072       print_err ("mq_send %s", qm_json);
00073       free ((void *)qm_json);
00074       return 0;
00075     }
00076   print_msg ("Message sent successfully!\n");
00077   free ((void *)qm_json);
00078   return 1;
00079 }
00080
00093 void *
00094 dequeue (mqd_t queue_d, qm_type *qmt)
00095 {
00096   char *qm_json = (char *)malloc (sizeof (char) * MAX_QM_SIZE);
00097
00098   if (mq_receive (queue_d, qm_json, MAX_QM_SIZE, 0) == -1)
00099     {
00100       free ((void *)qm_json);
00101       print_err ("mq_rec %d %s", errno, strerror (errno));
00102       return NULL;
00103     }
00104
```

```
00105   print_msg ("Dequeued %s", qm_json);
00106   void *tmp_struct = string_to_struct (qm_json, qmt);
00107
00108   free ((void *)qm_json);
00109   return tmp_struct;
00110 }
```

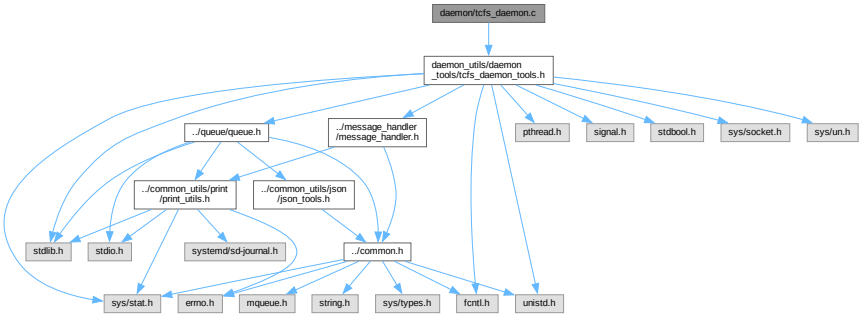
7.23 queue.h

```
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);
```

7.24 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

7.24.1 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.

7.24.2 Function Documentation

7.24.2.1 handle_termination()

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

Handle the termination if SIGTERM is received.

Parameters

signum	Integer corresponding to SIGNUM

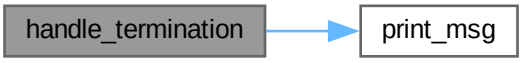
Todo : Implement remove_queue() to clear and delete the queue

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:

7.24.2.2 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

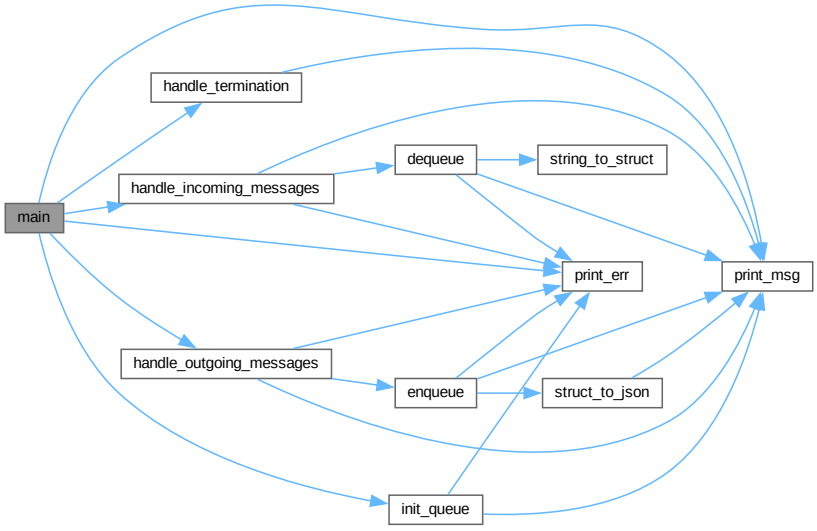
Todo : The brief description is basically false advertisement. It only spawn a thread and hangs infinitely

: 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_incoming_messages(), handle_outgoing_messages(), handle_termination(), init_queue(), MQUEUE, print_err(), print_msg(), and terminate.

Here is the call graph for this function:



7.24.3 Variable Documentation

7.24.3.1 MQUEUE

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

the queue file location

Definition at line 32 of file tcfs_daemon.c.

Referenced by main().

7.24.3.2 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().

7.24.3.3 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.

7.25 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 }
```

7.26 kernel-module/tcfs_kmodule.c File Reference

This will host the kernel module implementation in the future. It is not beeing currently developed.

7.26.1 Detailed Description

This will host the kernel module implementation in the future. It is not beeing currently developed.

Definition in file tcfs_kmodule.c.

7.27 tcfs_kmodule.c

Go to the documentation of this file.

```
00001
00008 /*
00009 #include <linux/kernel.h>
00010 #include <linux/module.h>
00011 #include <linux/syscalls.h>
00012 #include <linux/slab.h>
00013
00014 MODULE_LICENSE("GPL");
00015
00016 static char *key = NULL;
00017 static size_t key_size = 0;
00018
00019 SYSCALL_DEFINE2(putkey, char __user *, user_key, size_t, size)
00020 {
00021 char *new_key = kmalloc(size, GFP_KERNEL);
00022 if (!new_key)
00023 return -ENOMEM;
00024
00025 if (copy_from_user(new_key, user_key, size)) {
00026 kfree(new_key);
00027 return -EFAULT;
00028 }
00029
00030 kfree(key);
00031 key = new_key;
00032 key_size = size;
```

```
00033
00034 return 0;
00035 }
00036
00037 SYSCALL_DEFINE2(getkey, char __user *, user_key, size_t, size)
00038 {
00039 if (size < key_size)
00040 return -EINVAL;
00041
00042 if (copy_to_user(user_key, key, key_size))
00043 return -EFAULT;
00044
00045 return key_size;
00046 }
00047 */
```

7.28 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 }
```

7.29 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 ();
```

7.30 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 }
```

7.31 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
00038 char *root_path;
00043 char *password;
00044
00045 static int tcfs_getxattr (const char *fuse_path, const char *name, char *value,
00046                           size_t size);
00047
00051 static int
00052 tcfs_opendir (const char *fuse_path, struct fuse_file_info *fi)
00053 {
00054   (void)fuse_path;
00055   (void)fi;
00056   printf ("Called opendir UNIMPLEMENTED\n");
00057   /*int res = 0;
00058   DIR *dp;
00059   char path[PATH_MAX];
00060
00061   *path = prefix_path(fuse_path);
00062
00063   dp = opendir(path);
00064   if (dp == NULL)
00065       res = -errno;
00066
00067   fi->fh = (intptr_t) dp;
00068
00069   return res;*/
00070   return 0;
00071 }
00072
00073 static int
00074 tcfs_getattr (const char *fuse_path, struct stat *stbuf)
00075 {
00076   printf ("Called getattr\n");
00077   char *path = prefix_path (fuse_path, root_path);
00078
00079   int res;
00080
00081   res = stat (path, stbuf);
00082   if (res == -1)
00083     return -errno;
00084
00085   return 0;
00086 }
00087
00088 static int
00089 tcfs_access (const char *fuse_path, int mask)
00090 {
00091   printf ("Callen access\n");
00092   char *path = prefix_path (fuse_path, root_path);
00093
00094   int res;
00095
00096   res = access (path, mask);
00097   if (res == -1)
00098     return -errno;
00099
00100   return 0;
00101 }
00102
00103 static int
00104 tcfs_readlink (const char *fuse_path, char *buf, size_t size)
00105 {
00106   char *path = prefix_path (fuse_path, root_path);
00107
00108   int res;
00109
00110   res = readlink (path, buf, size - 1);
00111   if (res == -1)
00112     return -errno;
```

```
00113
00114   buf[res] = '\0';
00115   return 0;
00116 }
00117
00118 static int
00119 tcfs_readdir (const char *fuse_path, void *buf, fuse_fill_dir_t filler,
00120               off_t offset, struct fuse_file_info *fi)
00121 {
00122   (void)offset;
00123   (void)fi;
00124
00125   printf ("Called readdir %s\n", fuse_path);
00126   char *path = prefix_path (fuse_path, root_path);
00127
00128   DIR *dp;
00129   struct dirent *de;
00130
00131   dp = opendir (path);
00132   if (dp == NULL)
00133     {
00134       perror ("Could not open the directory");
00135       return -errno;
00136     }
00137
00138   while ((de = readdir (dp)) != NULL)
00139     {
00140       struct stat st;
00141       memset (&st, 0, sizeof (st));
00142       st.st_ino = de->d_ino;
00143       st.st_mode = de->d_type << 12;
00144       if (filler (buf, de->d_name, &st, 0))
00145         break;
00146     }
00147
00148   closedir (dp);
00149   return 0;
00150 }
00151
00152 static int
00153 tcfs_mknod (const char *fuse_path, mode_t mode, dev_t rdev)
00154 {
00155   printf ("Called mknod\n");
00156   char *path = prefix_path (fuse_path, root_path);
00157
00158   int res;
00159
00160   /* On Linux this could just be 'mknod(path, mode, rdev)' but this
00161      is more portable */
00162   if (S_ISREG (mode))
00163     {
00164       res = open (path, O_CREAT | O_EXCL | O_WRONLY, mode);
00165       if (res >= 0)
00166         res = close (res);
00167     }
00168   else if (S_ISFIFO (mode))
00169     res = mkfifo (path, mode);
00170   else
00171     res = mknod (path, mode, rdev);
00172   if (res == -1)
00173     return -errno;
00174
00175   return 0;
00176 }
00177
00178 static int
00179 tcfs_mkdir (const char *fuse_path, mode_t mode)
00180 {
00181   printf ("Called mkdir\n");
00182   char *path = prefix_path (fuse_path, root_path);
00183
00184   int res;
00185
00186   res = mkdir (path, mode);
00187   if (res == -1)
00188     return -errno;
00189
00190   return 0;
00191 }
00192
00193 static int
00194 tcfs_unlink (const char *fuse_path)
00195 {
00196   printf ("Called unlink\n");
00197   char *path = prefix_path (fuse_path, root_path);
00198
00199   int res;
```

```
00200
00201   res = unlink (path);
00202   if (res == -1)
00203     return -errno;
00204
00205   return 0;
00206 }
00207
00208 static int
00209 tcfs_rmdir (const char *fuse_path)
00210 {
00211   printf ("Called rmdir\n");
00212   char *path = prefix_path (fuse_path, root_path);
00213
00214   int res;
00215
00216   res = rmdir (path);
00217   if (res == -1)
00218     return -errno;
00219
00220   return 0;
00221 }
00222
00223 static int
00224 tcfs_symlink (const char *from, const char *to)
00225 {
00226   printf ("Called symlink\n");
00227   int res;
00228
00229   res = symlink (from, to);
00230   if (res == -1)
00231     return -errno;
00232
00233   return 0;
00234 }
00235
00236 static int
00237 tcfs_rename (const char *from, const char *to)
00238 {
00239   printf ("Called rename\n");
00240   int res;
00241
00242   res = rename (from, to);
00243   if (res == -1)
00244     return -errno;
00245
00246   return 0;
00247 }
00248
00249 static int
00250 tcfs_link (const char *from, const char *to)
00251 {
00252   printf ("Called link\n");
00253   int res;
00254
00255   res = link (from, to);
00256   if (res == -1)
00257     return -errno;
00258
00259   return 0;
00260 }
00261
00262 static int
00263 tcfs_chmod (const char *fuse_path, mode_t mode)
00264 {
00265   printf ("Called chmod\n");
00266   char *path = prefix_path (fuse_path, root_path);
00267
00268   int res;
00269
00270   res = chmod (path, mode);
00271   if (res == -1)
00272     return -errno;
00273
00274   return 0;
00275 }
00276
00277 static int
00278 tcfs_chown (const char *fuse_path, uid_t uid, gid_t gid)
00279 {
00280   printf ("Called chown\n");
00281   char *path = prefix_path (fuse_path, root_path);
00282
00283   int res;
00284
00285   res = lchown (path, uid, gid);
00286   if (res == -1)
```

```
00287     return -errno;
00288
00289   return 0;
00290 }
00291
00292 static int
00293 tcfs_truncate (const char *fuse_path, off_t size)
00294 {
00295   printf ("Called truncate\n");
00296   char *path = prefix_path (fuse_path, root_path);
00297
00298   int res;
00299
00300   res = truncate (path, size);
00301   if (res == -1)
00302     return -errno;
00303
00304   return 0;
00305 }
00306
00307 // #ifdef HAVE_UTIMENSAT
00308 static int
00309 tcfs_utimens (const char *fuse_path, const struct timespec ts[2])
00310 {
00311   printf ("Called utimens\n");
00312   char *path = prefix_path (fuse_path, root_path);
00313
00314   int res;
00315   struct timeval tv[2];
00316
00317   tv[0].tv_sec = ts[0].tv_sec;
00318   tv[0].tv_usec = ts[0].tv_nsec / 1000;
00319   tv[1].tv_sec = ts[1].tv_sec;
00320   tv[1].tv_usec = ts[1].tv_nsec / 1000;
00321
00322   res = utimes (path, tv);
00323   if (res == -1)
00324     return -errno;
00325
00326   return 0;
00327 }
00328 // #endif
00329
00330 static int
00331 tcfs_open (const char *fuse_path, struct fuse_file_info *fi)
00332 {
00333   printf ("Called open\n");
00334   char *path = prefix_path (fuse_path, root_path);
00335   int res;
00336
00337   res = open (path, fi->flags);
00338   if (res == -1)
00339     return -errno;
00340
00341   close (res);
00342   return 0;
00343 }
00344
00345 static inline int
00346 file_size (FILE *file)
00347 {
00348   struct stat st;
00349
00350   if (fstat (fileno (file), &st) == 0)
00351     return st.st_size;
00352
00353   return -1;
00354 }
00355
00356 static int
00357 tcfs_read (const char *fuse_path, char *buf, size_t size, off_t offset,
00358            struct fuse_file_info *fi)
00359 {
00360   (void)size;
00361   (void)fi;
00362
00363   printf ("Calling read\n");
00364   FILE *path_ptr, *tmpf;
00365   char *path;
00366   int res;
00367
00368   // Retrieve the username
00369   char username_buf[1024];
00370   size_t username_buf_size = 1024;
00371   get_user_name (username_buf, username_buf_size);
00372
00373   path = prefix_path (fuse_path, root_path);
```

```
00374
00375   path_ptr = fopen (path, "r");
00376   tmpf = tmpfile ();
00377
00378   // Get key size
00379   char *size_key_char = malloc (sizeof (char) * 20);
00380   if (tcfs_getxattr (fuse_path, "user.key_len", size_key_char, 20) == -1)
00381     {
00382       perror ("Could not get file key size");
00383       return -errno;
00384     }
00385   ssize_t size_key = strtol (size_key_char, NULL, 10);
00386
00387   // Retrive the file key
00388   unsigned char *encrypted_key = malloc ((size_key + 1) * sizeof (char));
00389   encrypted_key[size_key] = '\0';
00390   if (tcfs_getxattr (fuse_path, "user.key", (char *)encrypted_key, size_key)
00391       == -1)
00392     {
00393       perror ("Could not get encrypted key for file in tcfs_read");
00394       return -errno;
00395     }
00396
00397   // Decrypt the file key
00398   unsigned char *decrypted_key;
00399   decrypted_key = decrypt_string (encrypted_key, password);
00400
00401   /* Decrypt*/
00402   if (do_crypt (path_ptr, tmpf, DECRYPT, decrypted_key) != 1)
00403     {
00404       perror ("Err: do_crypt cannot decrypt file");
00405       return -errno;
00406     }
00407
00408   /* Something went terribly wrong if this is the case. */
00409   if (path_ptr == NULL || tmpf == NULL)
00410     return -errno;
00411
00412   if (fflush (tmpf) != 0)
00413     {
00414       perror ("Err: Cannot flush file in read process");
00415       return -errno;
00416     }
00417   if (fseek (tmpf, offset, SEEK_SET) != 0)
00418     {
00419       perror ("Err: cannot fseek while reading file");
00420       return -errno;
00421     }
00422
00423   /* Read our tmpfile into the buffer. */
00424   res = fread (buf, 1, file_size (tmpf), tmpf);
00425   if (res == -1)
00426     {
00427       perror ("Err: cannot fread whine in read");
00428       res = -errno;
00429     }
00430
00431   fclose (tmpf);
00432   fclose (path_ptr);
00433   free (encrypted_key);
00434   free (decrypted_key);
00435   return res;
00436 }
00437
00438 static int
00439 tcfs_write (const char *fuse_path, const char *buf, size_t size, off_t offset,
00440             struct fuse_file_info *fi)
00441 {
00442   (void)fi;
00443   printf ("Called write\n");
00444
00445   FILE *path_ptr, *tmpf;
00446   char *path;
00447   int res;
00448   int tmpf_descriptor;
00449
00450   path = prefix_path (fuse_path, root_path);
00451   path_ptr = fopen (path, "r+");
00452   tmpf = tmpfile ();
00453   tmpf_descriptor = fileno (tmpf);
00454
00455   // Get the key size
00456   char *size_key_char = malloc (sizeof (char) * 20);
00457   if (tcfs_getxattr (fuse_path, "user.key_len", size_key_char, 20) == -1)
00458     {
00459       perror ("Could not get file key size");
00460       return -errno;
```

```
00461     }
00462   ssize_t size_key = strtol (size_key_char, NULL, 10);
00463
00464   // Retrieve the file key
00465   unsigned char *encrypted_key
00466       = malloc (sizeof (unsigned char) * (size_key + 1));
00467   encrypted_key[size_key] = '\0';
00468   if (tcfs_getxattr (fuse_path, "user.key", (char *)encrypted_key, size_key)
00469       == -1)
00470     {
00471       perror ("Could not get file encrypted key in tcfs write");
00472       return -errno;
00473     }
00474
00475   // Decrypt the file key
00476   unsigned char *decrypted_key = malloc (sizeof (unsigned char) * 33);
00477   decrypted_key[32] = '\0';
00478   decrypted_key = decrypt_string (encrypted_key, password);
00479
00480   /* Something went terribly wrong if this is the case. */
00481   if (path_ptr == NULL || tmpf == NULL)
00482     {
00483       fprintf (stderr,
00484                "Something went terribly wrong, cannot create new files\n");
00485       return -errno;
00486     }
00487
00488   /* if the file to write to exists, read it into the tempfile */
00489   if (tcfs_access (fuse_path, R_OK) == 0 && file_size (path_ptr) > 0)
00490     {
00491       if (do_crypt (path_ptr, tmpf, DECRYPT, decrypted_key) == 0)
00492         {
00493           perror ("do_crypt: Cannot cypher file\n");
00494           return --errno;
00495         }
00496       rewind (path_ptr);
00497       rewind (tmpf);
00498     }
00499
00500   /* Read our tmpfile into the buffer. */
00501   res = pwrite (tmpf_descriptor, buf, size, offset);
00502   if (res == -1)
00503     {
00504       printf ("%d\n", res);
00505       perror ("pwrite: cannot read tmpfile into the buffer\n");
00506       res = -errno;
00507     }
00508
00509   /* Encrypt*/
00510   if (do_crypt (tmpf, path_ptr, ENCRYPT, decrypted_key) == 0)
00511     {
00512       perror ("do_crypt 2: cannot cypher file\n");
00513       return -errno;
00514     }
00515
00516   fclose (tmpf);
00517   fclose (path_ptr);
00518   free (encrypted_key);
00519   free (decrypted_key);
00520
00521   return res;
00522 }
00523
00524 static int
00525 tcfs_statfs (const char *fuse_path, struct statvfs *stbuf)
00526 {
00527   printf ("Called statfs\n");
00528   char *path = prefix_path (fuse_path, root_path);
00529
00530   int res;
00531
00532   res = statvfs (path, stbuf);
00533   if (res == -1)
00534     return -errno;
00535
00536   return 0;
00537 }
00538
00539 static int
00540 tcfs_setxattr (const char *fuse_path, const char *name, const char *value,
00541                size_t size, int flags)
00542 {
00543   char *path = prefix_path (fuse_path, root_path);
00544   int res = 1;
00545   if ((res = lsetxattr (path, name, value, size, flags)) == -1)
00546     perror ("tcfs_lsetxattr");
00547   if (res == -1)
```

```
00548     return -errno;
00549   return 0;
00550 }
00551
00552 static int
00553 tcfs_create (const char *fuse_path, mode_t mode, struct fuse_file_info *fi)
00554 {
00555   (void)fi;
00556   (void)mode;
00557   printf ("Called create\n");
00558
00559   FILE *res;
00560   res = fopen (prefix_path (fuse_path, root_path), "w");
00561   if (res == NULL)
00562     return -errno;
00563
00564   // Flag file as encrypted
00565   if (tcfs_setxattr (fuse_path, "user.encrypted", "true", 4, 0)
00566       != 0) //(fsetxattr(fileno(res), "user.encrypted", "true", 4, 0) != 0)
00567     {
00568       fclose (res);
00569       return -errno;
00570     }
00571
00572   // Generate and set a new encrypted key for the file
00573   unsigned char *key = malloc (sizeof (unsigned char) * 33);
00574   key[32] = '\0';
00575   generate_key (key);
00576
00577   if (key == NULL)
00578     {
00579       perror ("cannot generate file key");
00580       return -errno;
00581     }
00582   if (is_valid_key (key) == 0)
00583     {
00584       fprintf (stderr, "Generated key size invalid\n");
00585       return -1;
00586     }
00587
00588   // Encrypt the generated key
00589   int encrypted_key_len;
00590   unsigned char *encrypted_key
00591       = encrypt_string (key, password, &encrypted_key_len);
00592
00593   // Set the file key
00594   if (tcfs_setxattr (fuse_path, "user.key", (const char *)encrypted_key,
00595                      encrypted_key_len, 0)
00596       != 0) //(fsetxattr(fileno(res), "user.key", encrypted_key, 32, 0) != 0)
00597     {
00598       perror ("Err setting key xattr");
00599       return -errno;
00600     }
00601   // Set key size
00602   char encrypted_key_len_char[20];
00603   snprintf (encrypted_key_len_char, sizeof (encrypted_key_len_char), "%d",
00604             encrypted_key_len);
00605   if (tcfs_setxattr (fuse_path, "user.key_len", encrypted_key_len_char,
00606                      sizeof (encrypted_key_len_char), 0)
00607       != 0) //(fsetxattr(fileno(res), "user.key", encrypted_key, 32, 0) != 0)
00608     {
00609       perror ("Err setting key_len xattr");
00610       return -errno;
00611     }
00612
00613   free (encrypted_key);
00614   free (key);
00615   fclose (res);
00616   return 0;
00617 }
00618
00619 static int
00620 tcfs_release (const char *fuse_path, struct fuse_file_info *fi)
00621 {
00622   /* Just a stub.    This method is optional and can safely be left
00623      unimplemented */
00624   char *path = prefix_path (fuse_path, root_path);
00625
00626   (void)path;
00627   (void)fi;
00628   return 0;
00629 }
00630
00631 static int
00632 tcfs_fsync (const char *fuse_path, int isdatasync, struct fuse_file_info *fi)
00633 {
00634   /* Just a stub.    This method is optional and can safely be left
```

```
00635      unimplemented */
00636   char *path = prefix_path (fuse_path, root_path);
00637
00638   (void)path;
00639   (void)isdatasync;
00640   (void)fi;
00641   return 0;
00642 }
00643
00644 static int
00645 tcfs_getxattr (const char *fuse_path, const char *name, char *value,
00646                size_t size)
00647 {
00648   char *path = prefix_path (fuse_path, root_path);
00649   printf ("Called getxattr on %s name:%s size:%zu\n", path, name, size);
00650
00651   if (strcmp (name, "security.capability")
00652       == 0) // TODO: I don't know why this is called every time, understand why
00653             // and handle this
00654     return 0;
00655
00656   int res = (int)lgetxattr (path, name, value, size);
00657   if (res == -1)
00658     {
00659       perror ("Could not get xattr for file");
00660       return -errno;
00661     }
00662   return res;
00663 }
00664
00665 static int
00666 tcfs_listxattr (const char *fuse_path, char *list, size_t size)
00667 {
00668   printf ("Called listxattr\n");
00669   char *path = prefix_path (fuse_path, root_path);
00670
00671   int res = llistxattr (path, list, size);
00672   if (res == -1)
00673     return -errno;
00674   return res;
00675 }
00676
00677 static int
00678 tcfs_removexattr (const char *fuse_path, const char *name)
00679 {
00680   printf ("Called removexattr\n");
00681   char *path = prefix_path (fuse_path, root_path);
00682
00683   int res = lremovexattr (path, name);
00684   if (res == -1)
00685     return -errno;
00686   return 0;
00687 }
00688
00689 static struct fuse_operations tcfs_oper = {
00690   .opendir = tcfs_opendir,
00691   .getattr = tcfs_getattr,
00692   .access = tcfs_access,
00693   .readlink = tcfs_readlink,
00694   .readdir = tcfs_readdir,
00695   .mknod = tcfs_mknod,
00696   .mkdir = tcfs_mkdir,
00697   .symlink = tcfs_symlink,
00698   .unlink = tcfs_unlink,
00699   .rmdir = tcfs_rmdir,
00700   .rename = tcfs_rename,
00701   .link = tcfs_link,
00702   .chmod = tcfs_chmod,
00703   .chown = tcfs_chown,
00704   .truncate = tcfs_truncate,
00705   .utimens = tcfs_utimens,
00706   .open = tcfs_open,
00707   .read = tcfs_read,
00708   .write = tcfs_write,
00709   .statfs = tcfs_statfs,
00710   .create = tcfs_create,
00711   .release = tcfs_release,
00712   .fsync = tcfs_fsync,
00713   .setxattr = tcfs_setxattr,
00714   .getxattr = tcfs_getxattr,
00715   .listxattr = tcfs_listxattr,
00716   .removexattr = tcfs_removexattr,
00717 };
00718
00719 const char *argp_program_version = "TCFS Alpha";
00720 const char *argp_program_bug_address = "carloalbertogiordano@duck.com";
00721
```

```
00722 static char doc[] = "This is an implementation on TCFS\ntcfs -s <source_path> "
00723                     "-d <dest_path> -p <password> [fuse arguments]";
00724
00725 static char args_doc[] = "";
00726
00727 static struct argp_option options[]
00728     = { { "source", 's', "SOURCE", 0, "Source file path", -1 },
00729         { "destination", 'd', "DESTINATION", 0, "Destination file path", -1 },
00730         { "password", 'p', "PASSWORD", 0, "Password", -1 },
00731         { NULL } };
00732
00733 struct arguments
00734 {
00735   char *source;
00736   char *destination;
00737   char *password;
00738 };
00739
00740 static error_t
00741 parse_opt (int key, char *arg, struct argp_state *state)
00742 {
00743   struct arguments *arguments = state->input;
00744
00745   switch (key)
00746     {
00747     case 's':
00748       arguments->source = arg;
00749       break;
00750     case 'd':
00751       arguments->destination = arg;
00752       break;
00753     case 'p':
00754       arguments->password = arg;
00755       break;
00756     case ARGP_KEY_ARG:
00757       return ARGP_ERR_UNKNOWN;
00758     default:
00759       return ARGP_ERR_UNKNOWN;
00760     }
00761
00762   return 0;
00763 }
00764
00765 static struct argp argp = { options, parse_opt, args_doc, doc, 0, NULL, NULL };
00766
00767 int
00768 main (int argc, char *argv[])
00769 {
00770   umask (0);
00771
00772   struct arguments arguments;
00773
00774   arguments.source = NULL;
00775   arguments.destination = NULL;
00776   arguments.password = NULL;
00777
00778   argp_parse (&argp, argc, argv, 0, 0, &arguments);
00779
00780   if (arguments.source == NULL || arguments.destination == NULL
00781       || arguments.password == NULL)
00782     {
00783       printf ("Err: You need to specify at least 3 arguments\n");
00784       return -1;
00785     }
00786
00787   printf ("Source: %s\n", arguments.source);
00788   printf ("Destination: %s\n", arguments.destination);
00789   root_path = arguments.source;
00790
00791   if (is_valid_key ((unsigned char *)arguments.password) == 0)
00792     {
00793       fprintf (stderr, "Inserted key not valid\n");
00794       return 1;
00795     }
00796
00797   struct fuse_args args_fuse = FUSE_ARGS_INIT (0, NULL);
00798   fuse_opt_add_arg (&args_fuse, "./tcfs");
00799   fuse_opt_add_arg (&args_fuse, arguments.destination);
00800   fuse_opt_add_arg (&args_fuse,
00801                     "-f"); // TODO: this is forced for now, but will be passed
00802                            // via options in the future
00803   fuse_opt_add_arg (&args_fuse,
00804                     "-s"); // TODO: this is forced for now, but will be passed
00805                            // via options in the future
00806
00807   // Print what we are passing to fuse TODO: This will be removed
00808   for (int i = 0; i < args_fuse.argc; i++)
```

```
00809     {
00810       printf ("%s ", args_fuse.argv[i]);
00811     }
00812   printf ("\n");
00813 
00814   // Get username
00815   /*
00816   char buf[1024];
00817   size_t buf_size = 1024;
00818   get_user_name(buf, buf_size);
00819   */
00820 
00821   password = arguments.password;
00822 
00823   return fuse_main (args_fuse.argc, args_fuse.argv, &tcfs_oper, NULL);
00824 }
```

7.32 crypt-utils.c

```
00001 /***
00002 
00003  *
00004  **/
00005 #include "crypt-utils.h"
00006 
00012 #define BLOCKSIZE 1024
00019 #define IV_SIZE 32
00025 #define KEY_SIZE 32
00026 
00053 extern int
00054 do_crypt (FILE *in, FILE *out, int action, unsigned char *key_str)
00055 {
00056   /* Local Vars */
00057 
00058   /* Buffers */
00059   unsigned char inbuf[BLOCKSIZE];
00060   int inlen;
00061   /* Allow enough space in output buffer for additional cipher block */
00062   unsigned char outbuf[BLOCKSIZE + EVP_MAX_BLOCK_LENGTH];
00063   int outlen;
00064   int writelen;
00065 
00066   /* OpenSSL libcrypto vars */
00067   EVP_CIPHER_CTX *ctx;
00068   ctx = EVP_CIPHER_CTX_new ();
00069 
00070   unsigned char key[KEY_SIZE];
00071   unsigned char iv[IV_SIZE];
00072   int nrounds = 5;
00073 
00074   /* tmp vars */
00075   int i;
00076   /* Setup Encryption Key and Cipher Engine if in cipher mode */
00077   if (action >= 0)
00078     {
00079       if (!key_str)
00080         {
00081           /* Error */
00082           fprintf (stderr, "Key_str must not be NULL\n");
00083           return 0;
00084         }
00085       /* Build Key from String */
00086       i = EVP_BytesToKey (EVP_aes_256_cbc (), EVP_sha1 (), NULL, key_str,
00087                           (int)strlen ((const char *)key_str), nrounds, key,
00088                           iv);
00089       if (i != 32)
00090         {
00091           /* Error */
00092           fprintf (stderr, "Key size is %d bits - should be 256 bits\n",
00093                    i * 8);
00094           return 0;
00095         }
00096       /* Init Engine */
00097       EVP_CIPHER_CTX_init (ctx);
00098       EVP_CipherInit_ex (ctx, EVP_aes_256_cbc (), NULL, key, iv, action);
00099     }
00100 
00101   /* Loop through Input File*/
00102   for (;;)
00103     {
00104       /* Read Block */
00105       inlen = fread (inbuf, sizeof (*inbuf), BLOCKSIZE, in);
00106       if (inlen <= 0)
```

```
00107         {
00108           /* EOF -> Break Loop */
00109           break;
00110         }
00111
00112       /* If in cipher mode, perform cipher transform on block */
00113       if (action >= 0)
00114         {
00115           if (!EVP_CipherUpdate (ctx, outbuf, &outlen, inbuf, inlen))
00116             {
00117               /* Error */
00118               EVP_CIPHER_CTX_cleanup (ctx);
00119               return 0;
00120             }
00121         }
00122       /* If in pass-through mode. copy block as is */
00123       else
00124         {
00125           memcpy (outbuf, inbuf, inlen);
00126           outlen = inlen;
00127         }
00128
00129       /* Write Block */
00130       writelen = fwrite (outbuf, sizeof (*outbuf), outlen, out);
00131       if (writelen != outlen)
00132         {
00133           /* Error */
00134           perror ("fwrite error");
00135           EVP_CIPHER_CTX_cleanup (ctx);
00136           return 0;
00137         }
00138     }
00139
00140   /* If in cipher mode, handle necessary padding */
00141   if (action >= 0)
00142     {
00143       /* Handle remaining cipher block + padding */
00144       if (!EVP_CipherFinal_ex (ctx, outbuf, &outlen))
00145         {
00146           /* Error */
00147           EVP_CIPHER_CTX_cleanup (ctx);
00148           return 0;
00149         }
00150       /* Write remainign cipher block + padding*/
00151       fwrite (outbuf, sizeof (*inbuf), outlen, out);
00152       EVP_CIPHER_CTX_cleanup (ctx);
00153     }
00154
00155   /* Success */
00156   return 1;
00157 }
00158
00168 int
00169 check_entropy (void)
00170 {
00171   FILE *entropy_file = fopen ("/proc/sys/kernel/random/entropy_avail", "r");
00172   if (entropy_file == NULL)
00173     {
00174       perror ("Err: Cannot open entropy file");
00175       return -1;
00176     }
00177
00178   int entropy_value;
00179   if (fscanf (entropy_file, "%d", &entropy_value) != 1)
00180     {
00181       perror ("Err: Cannot estimate entropy");
00182       fclose (entropy_file);
00183       return -1;
00184     }
00185
00186   fclose (entropy_file);
00187   return entropy_value;
00188 }
00189
00198 void
00199 add_entropy (void)
00200 {
00201   FILE *urandom = fopen ("/dev/urandom", "rb");
00202   if (urandom == NULL)
00203     {
00204       perror ("Err: Cannot open /dev/urandom");
00205       exit (EXIT_FAILURE);
00206     }
00207
00208   unsigned char random_data[32];
00209   size_t bytes_read = fread (random_data, 1, sizeof (random_data), urandom);
00210   fclose (urandom);
```

```
00211
00212   if (bytes_read != sizeof (random_data))
00213     {
00214       fprintf (stderr, "Err: Cannot read data\n");
00215       exit (EXIT_FAILURE);
00216     }
00217
00218   // Usa i dati casuali per aggiungere entropia
00219   RAND_add (random_data, sizeof (random_data),
00220             0.5); // 0.5 è un peso arbitrario
00221
00222   fprintf (stdout, "Entropy added successfully!\n");
00223 }
00224
00231 void
00232 generate_key (unsigned char *destination)
00233 {
00234   fprintf (stdout, "Generating a new key...\n");
00235
00236   // Why? Because if we try to create a large number of files there might not
00237   // be enough random bytes in the system to generate a key
00238   for (int i = 0; i < 10; i++)
00239     {
00240       int entropy = check_entropy ();
00241       if (entropy < 128)
00242         {
00243           fprintf (stderr, "WARN: not enough entropy, creating some...\n");
00244           add_entropy ();
00245         }
00246
00247       if (RAND_bytes (destination, 32) != 1)
00248         {
00249           fprintf (stderr, "Err: Cannot generate key\n");
00250           destination = NULL;
00251         }
00252
00253       if (strlen ((const char *)destination) == 32)
00254         break;
00255     }
00256
00257   if (is_valid_key (destination) == 0)
00258     {
00259       fprintf (stderr, "Err: Generated key is invalid\n");
00260       print_aes_key (destination);
00261       destination = NULL;
00262     }
00263 }
00264
00274 unsigned char *
00275 encrypt_string (unsigned char *plaintext, const char *key,
00276                 int *encrypted_key_len)
00277 {
00278   EVP_CIPHER_CTX *ctx;
00279   const EVP_CIPHER *cipher = EVP_aes_256_cbc ();
00280   unsigned char iv[AES_BLOCK_SIZE];
00281   memset (iv, 0, AES_BLOCK_SIZE);
00282
00283   ctx = EVP_CIPHER_CTX_new ();
00284   if (!ctx)
00285     {
00286       return NULL;
00287     }
00288
00289   EVP_EncryptInit_ex (ctx, cipher, NULL, (const unsigned char *)key, iv);
00290
00291   size_t plaintext_len = strlen ((const char *)plaintext);
00292   unsigned char ciphertext[plaintext_len + AES_BLOCK_SIZE];
00293   memset (ciphertext, 0, sizeof (ciphertext));
00294
00295   int len;
00296   EVP_EncryptUpdate (ctx, ciphertext, &len, plaintext, plaintext_len);
00297   EVP_EncryptFinal_ex (ctx, ciphertext + len, &len);
00298   EVP_CIPHER_CTX_free (ctx);
00299
00300   unsigned char *encoded_string = malloc (len * 2 + 1);
00301   if (!encoded_string)
00302     {
00303       return NULL;
00304     }
00305
00306   for (int i = 0; i < len; i++)
00307     {
00308       sprintf ((char *)&encoded_string[i * 2], "%02x", ciphertext[i]);
00309     }
00310   encoded_string[len * 2] = '\0';
00311
00312   *encrypted_key_len = len * 2;
```

```
00313   return encoded_string;
00314 }
00315
00324 unsigned char *
00325 decrypt_string (unsigned char *ciphertext, const char *key)
00326 {
00327   EVP_CIPHER_CTX *ctx;
00328   const EVP_CIPHER *cipher
00329       = EVP_aes_256_cbc (); // Choose the correct algorithm
00330   unsigned char iv[AES_BLOCK_SIZE];
00331   memset (iv, 0, AES_BLOCK_SIZE);
00332
00333   ctx = EVP_CIPHER_CTX_new ();
00334   EVP_DecryptInit_ex (ctx, cipher, NULL, (const unsigned char *)key, iv);
00335
00336   size_t decoded_len = strlen ((const char *)ciphertext);
00337
00338   unsigned char plaintext[decoded_len];
00339   memset (plaintext, 0, sizeof (plaintext));
00340
00341   int len;
00342   EVP_DecryptUpdate (ctx, plaintext, &len, ciphertext, (int)decoded_len);
00343   EVP_DecryptFinal_ex (ctx, plaintext + len, &len);
00344   EVP_CIPHER_CTX_free (ctx);
00345
00346   unsigned char *decrypted_string = (unsigned char *)malloc (decoded_len + 1);
00347   memcpy (decrypted_string, plaintext, decoded_len);
00348   decrypted_string[decoded_len] = '\0';
00349
00350   return decrypted_string;
00351 }
00352
00359 int
00360 is_valid_key (const unsigned char *key)
00361 {
00362   char str[33];
00363   memcpy (str, key, 32);
00364   str[32] = '\0';
00365   size_t key_length = strlen (str);
00366   return key_length != 32 ? 0 : 1;
00367 }
00368
00369 /*
00370 int rebuild_key(char *key, char *cert, char *dest){
00371     return -1;
00372 }*/
```

7.33 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
00020 #define ENCRYPT 1
00025 #define DECRYPT 0
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
00031 unsigned char *encrypt_string (unsigned char *plaintext, const char *key,
00032                                int *encrypted_len);
00033
00034 unsigned char *decrypt_string (unsigned char *base64_ciphertext,
00035                                const char *key);
00036
00037 int is_valid_key (const unsigned char *key);
00038
00039 /*
00040 int rebuild_key(char *key, char *cert, char *dest);
00041 */
```

7.34 userspace-module/utils/password_manager/password_manager.c File Reference

This file will handle key exchanges with the kernel module. This is not being currently developed.

7.34.1 Detailed Description

This file will handle key exchanges with the kernel module. This is not being currently developed.

Definition in file password_manager.c.

7.35 password_manager.c

Go to the documentation of this file.

```
00001 // TODO: This util will handle requesting keys to kernel
00002
00009 /*
00010 #include "password_manager.h"
00011 #include "../crypt-utils/crypt-utils.h"
00012
00013 char *true_key;
00014
00015 int insert_key(char* key, char* cert, int is_sys_call)
00016 {
00017     if (is_sys_call == WITH_SYS_CALL)
00018     {
00019         fprintf(stderr, "The kernal module has not been implemented yet, saving
00020 key in userspace\n \ This will change in the future"); insert_key(key, cert,
00021 WITHOUT_SYS_CALL);
00022     }
00023     return rebuild_key(key, cert, true_key);
00024 }
00025
00026 char *request_key(int is_sys_call){
00027     return NULL;
00028 }
00029 int delete_key(int is_sys_call){
00030     return -1;
00031 }*/
```

7.36 password_manager.h

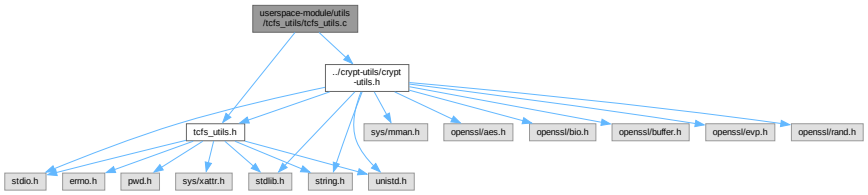
```
00001 #include <stddef.h>
00002 #include <stdio.h>
00003
00010 #define WITH_SYS_CALL 1
00017 #define WITHOUT_SYS_CALL 0
00018 /*
00019 int insert_key(char* key, char* cert, int is_sys_call);
00020 char *request_key(int is_sys_call);
00021 int delete_key(int is_sys_call);*/
```

7.37 userspace-module/utils/tcfs_utils/tcfs_utils.c File Reference

This file contains an assortment of functions used by tcfs.c.

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

Include dependency graph for tcfs_utils.c:



Functions

- void get_user_name (char *buf, size_t size)

  Fetch the username of the current user.
- int is_encrypted (const char *path)

  Check if a file is encrypted by TCFS.
- char * prefix_path (const char *path, const char *realpath)

  Prefix the realpath to the fuse path.
- int read_file (FILE *file)

  Read a file, useful for debugging tmpfiles.
- int get_encrypted_key (char *filepath, unsigned char *encrypted_key)

  Get the xattr value describing the key of a file.
- void print_aes_key (unsigned char *key)

  Print the value of an aes key.

7.37.1 Detailed Description

This file contains an assortment of functions used by tcfs.c.

See also

tcfs.c

Definition in file tcfs_utils.c.

7.37.2 Function Documentation

7.37.2.1 get_encrypted_key()

```
int get_encrypted_key (
            char * filepath,
            unsigned char * encrypted_key )
```

Get the xattr value describing the key of a file.

Deprecated There is no use currenly for this function. It was once used for debugging

Parameters

filepath	The full-path of the file
encrypted_key	The buffer to save the encrypted key to

Returns

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

See also

print_err

Definition at line 130 of file tcfs_utils.c.

References is_encrypted().

Here is the call graph for this function:

7.37.2.2 get_user_name()

```
void get_user_name (
            char * buf,
            size_t size )
```

Fetch the username of the current user.

Parameters

buf	The username will be written to this buffer
size	The size of the buffer

Returns

void

Note

If an error occurs it will be printed and the buffer will not be modified

Definition at line 17 of file tcfs_utils.c.

7.37.2.3 is_encrypted()

```
int is_encrypted (
            const char * path )
```

Check if a file is encrypted by TCFS.

Parameters

path	The fullpath of the file

Returns

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

See also

print_err

Definition at line 33 of file tcfs_utils.c.

Referenced by get_encrypted_key().

Here is the caller graph for this function:

7.37.2.4 prefix_path()

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

Prefix the realpath to the fuse path.

Parameters

path	The fuse path
realpath	The realpath to the directory mounted by TCFS

Returns

char * An allocated string containing the fullpath to the file

Note

Please free the result after use

Definition at line 57 of file tcfs_utils.c.

7.37.2.5 print_aes_key()

```
void print_aes_key (
            unsigned char * key )
```

Print the value of an aes key.

Deprecated There is currently no use for this function

Warning

THIS WILL PRINT THE AES KEY TO STDOUT. TCFS trusts the user by design, but this is excessive

Parameters

key	The string containing the key

Returns

void

Definition at line 170 of file tcfs_utils.c.

7.37.2.6 read_file()

```
int read_file (
            FILE * file )
```

Read a file, useful for debugging tmpfiles.

Deprecated Currently it has no use

Parameters

file	The file to read

Returns

0

Note

It will print "file was empty" if the file was empty

Definition at line 95 of file tcfs_utils.c.

7.38 tcfs_utils.c

Go to the documentation of this file.

```
00001 #include "tcfs_utils.h"
00002 #include "../crypt-utils/crypt-utils.h"
00003
00016 void
00017 get_user_name (char *buf, size_t size)
00018 {
00019   uid_t uid = geteuid ();
00020   struct passwd *pw = getpwuid (uid);
00021   if (pw)
00022     snprintf (buf, size, "%s", pw->pw_name);
00023   else
00024     perror ("Error: Could not retrieve username.\n");
00025 }
00026
00032 int
00033 is_encrypted (const char *path)
00034 {
00035   int ret;
00036   char xattr_val[5];
00037   getxattr (path, "user.encrypted", xattr_val, sizeof (char) * 5);
00038   xattr_val[4] == '\n';
00039
00040   return strcmp (xattr_val, "true") == 0 ? 1 : 0;
00041 }
00042
00043 /* char *prefix_path(const char *path))
00044  * Purpose:
00045  * Args:
00046  *
00047  * Return: NULL on error, char* on success
00048  */
00056 char *
00057 prefix_path (const char *path, const char *realpath)
00058 {
00059   if (path == NULL || realpath == NULL)
00060     {
00061       perror ("Err: path or realpath is NULL");
00062       return NULL;
00063     }
00064
00065   size_t len = strlen (path) + strlen (realpath) + 1;
00066   char *root_dir = malloc (len * sizeof (char));
00067
00068   if (root_dir == NULL)
00069     {
00070       perror ("Err: Could not allocate memory while in prefix_path");
00071       return NULL;
00072     }
00073
00074   if (strcpy (root_dir, realpath) == NULL)
00075     {
00076       perror ("strcpy: Cannot copy path");
00077       return NULL;
00078     }
00079   if (strcat (root_dir, path) == NULL)
00080     {
00081       perror ("strcat: in prefix_path cannot concatenate the paths");
00082       return NULL;
00083     }
00084   return root_dir;
00085 }
00086
00094 int
```

```
00095 read_file (FILE *file)
00096 {
00097   int c;
00098   int file_contains_something = 0;
00099   FILE *read = file; /* don't move original file pointer */
00100   if (read)
00101     {
00102       while ((c = getc (read)) != EOF)
00103         {
00104           file_contains_something = 1;
00105           putc (c, stderr);
00106         }
00107     }
00108   if (!file_contains_something)
00109     fprintf (stderr, "file was empty\n");
00110   rewind (file);
00111   /* fseek(tmpf, offset, SEEK_END); */
00112   return 0;
00113 }
00114
00115 /*
00116  * */
00117 /* int get_encrypted_key(char *filepath, void *encrypted_key)
00118  * Purpose: Get the encrypted file key from its xattrs
00119  * Args:
00120  *
00121  */
00129 int
00130 get_encrypted_key (char *filepath, unsigned char *encrypted_key)
00131 {
00132   printf ("\tGet Encrypted key for file %s\n", filepath);
00133   if (is_encrypted (filepath) == 1)
00134     {
00135       printf ("\t\tencrypted file\n");
00136
00137       FILE *src_file = fopen (filepath, "r");
00138       if (src_file == NULL)
00139         {
00140           fclose (src_file);
00141           perror ("Could not open the file to get the key");
00142           return -errno;
00143         }
00144       int src_fd;
00145       src_fd = fileno (src_file);
00146       if (src_fd == -1)
00147         {
00148           fclose (src_file);
00149           perror ("Could not get fd for the file");
00150           return -errno;
00151         }
00152
00153       if (fgetxattr (src_fd, "user.key", encrypted_key, 33) != -1)
00154         {
00155           fclose (src_file);
00156           return 1;
00157         }
00158     }
00159   return 0;
00160 }
00161
00169 void
00170 print_aes_key (unsigned char *key)
00171 {
00172   printf ("AES HEX:%s -> ", key);
00173   for (int i = 0; i < 32; i++)
00174     {
00175       printf ("%02x", key[i]);
00176     }
00177   printf ("\n");
00178 }
```

7.39 tcfs_utils.h

```
00001 #include <errno.h>
00002 #include <pwd.h>
00003 #include <stdio.h>
00004 #include <stdlib.h>
00005 #include <string.h>
00006 #include <sys/xattr.h>
00007 #include <unistd.h>
00008
00009 void get_user_name (char *buf, size_t size);
00010
```

```
00011 int is_encrypted (const char *path);
00012
00013 char *prefix_path (const char *path, const char *realpath);
00014
00015 int read_file (FILE *file);
00016
00017 int get_encrypted_key (char *filepath, unsigned char *encrypted_key);
00018
00019 void print_aes_key (unsigned char *key);
```

Index