TCFS

0.2

Generated on Sun Jan 21 2024 18:41:36 for TCFS by Doxygen 1.9.8

Sun Jan 21 2024 18:41:36

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 Technologies used

To achieve our goal many different auxiliary programs and tech has found its way in TCFS

- Securing the encryption Key
  - GPG
- Database management
  - MariaDB
- Documentation
  - Generated using Doxygen
  - Some documentation is currently missing
- Versioning
  - GitHub
- Code analysis
  - See the GitHub actions
- Code formatting
  - clang-format for C/C++ files

1.3 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. Now, the encryption keys are managed by a REST server that integrates with the database and publishes the public keys of the users.
- 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.

1.4 Getting Started

1.4.1 Documentation

Documentation is lacking but it can be found here

1.4.2 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.
- MariaDB: Install and start MariaDB
- Go: Install a compiler for go

1.4.3 Build

- Clone the TCFS repository to your local machine:

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

```
##
```

1.5 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"
```

#

1.5.1 Build and run the REST server

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

```
go build server
```

#

1.5.2 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
```

#

1.5.3 Kernel module

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

1.6 Usage of the fuse module

1.6.1 This is not raccomended, consider using the tcfs_helper program

1.6.2 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.

1.6.3 Unmount the NFS share when you're done:

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

then unmount the NFS share.

1.7 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.

1.8 License

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

1.9 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.

1.10 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 (done in the server, fuse module is missing this feature at the moment)
- Implement threshold sharing files (done in the server, fuse module is missing this feature at the moment).
- Server:
  - ~~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

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 3

Todo List

Member get_pass (char *pw)

This will be changed when a public/private key model will be avilable to TCFS userspace module

Member tcfs_opendir (const char *fuse_path, struct fuse_file_info *fi)

Implement the opendir function

Chapter 4

Namespace Index

4.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 5

Class Index

5.1 Class List

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

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

Chapter 7

Namespace Documentation

7.1 command_handler Namespace Reference

Namespaces

- namespace enviroinment

7.2 command_handler.enviroinment Namespace Reference

Functions

- countdown (msg, delta)
- init_env ()

7.2.1 Function Documentation

7.2.1.1 countdown()

```
command_handler.enviroinment.countdown (
              msg,
              delta )
```

Definition at line 6 of file enviroinment.py.

Referenced by command_handler.enviroinment.init_env().

Here is the caller graph for this function:

7.2.1.2 init_env()

```
command_handler.enviroinment.init_env ( )
```

Definition at line 12 of file enviroinment.py.

References command_handler.enviroinment.countdown().

Here is the call graph for this function:

7.3 main Namespace Reference

Functions

- foo ()

Variables

- win = Window("TCFS user helper", "200x200")
- init_env_butt = win.add_button("Initialize the environment", foo, row=0, col=0)
- mount_butt = win.add_button("Mount TCFS", foo, row=1, col=0)
- umount_butt = win.add_button("Umount TCFS", foo, row=2, col=0)
- shared_butt = win.add_button("Threshold share", foo, row=3, col=0)
- logout_butt = win.add_button("Logout", foo, row=4, col=0)

7.3.1 Function Documentation

7.3.1.1 foo()

```
main.foo ( )
```

Definition at line 4 of file main.py.

7.3.2 Variable Documentation

7.3.2.1 init_env_butt

```
main.init_env_butt = win.add_button("Initialize the environment", foo, row=0, col=0)
```

Definition at line 10 of file main.py.

7.3.2.2 logout_butt

```
main.logout_butt = win.add_button("Logout", foo, row=4, col=0)
```

Definition at line 14 of file main.py.

7.3.2.3 mount_butt

```
main.mount_butt = win.add_button("Mount TCFS", foo, row=1, col=0)
```

Definition at line 11 of file main.py.

7.3.2.4 shared_butt

```
main.shared_butt = win.add_button("Threshold share", foo, row=3, col=0)
```

Definition at line 13 of file main.py.

7.3.2.5 umount_butt

```
main.umount_butt = win.add_button("Umount TCFS", foo, row=2, col=0)
```

Definition at line 12 of file main.py.

7.3.2.6 win

```
main.win = Window("TCFS user helper", "200x200")
```

Definition at line 8 of file main.py.

7.4 ui Namespace Reference

Namespaces

- namespace main_window

7.5 ui.main_window Namespace Reference

Classes

- class Window

Functions

- modify_button_allign (button, row, column)

7.5.1 Function Documentation

7.5.1.1 modify_button_allign()

```
ui.main_window.modify_button_allign (
              button,
              row,
              column )
```

Definition at line 4 of file main_window.py.

Chapter 8

Class Documentation

8.1 argp Struct Reference

The struct used by argp.

8.1.1 Detailed Description

The struct used by argp.

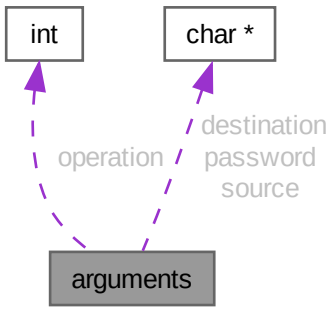
The documentation for this struct was generated from the following file:

- user/old_stuff/user_tcfs.c

8.2 arguments Struct Reference

Structure to hold the parsed arguments.

Collaboration diagram for arguments:



Public Attributes

- int operation

  Decribes the operation that will be executed by the main function.
- char * source
- char * destination
- char * password

8.2.1 Detailed Description

Structure to hold the parsed arguments.

Definition at line 48 of file user_tcfs.c.

8.2.2 Member Data Documentation

8.2.2.1 destination

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

Definition at line 736 of file tcfs.c.

Referenced by main(), and parse_opt().

8.2.2.2 operation

```
int arguments::operation
```

Decribes the operation that will be executed by the main function.

Definition at line 50 of file user_tcfs.c.

Referenced by main(), and parse_opt().

8.2.2.3 password

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

Definition at line 737 of file tcfs.c.

Referenced by main(), and parse_opt().

8.2.2.4 source

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

Definition at line 735 of file tcfs.c.

Referenced by main(), and parse_opt().

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

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

8.3 ui.main_window.Window Class Reference

Public Member Functions

- __init__ (self, str title, str geometry)
- add_button (self, str text, function, row=0, col=0)
- start_window (self)

Public Attributes

- window

8.3.1 Detailed Description

Definition at line 8 of file main_window.py.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 __init__()

```
ui.main_window.Window.__init__ (
              self,
             str title,
             str geometry )
```

Definition at line 10 of file main_window.py.

8.3.3 Member Function Documentation

8.3.3.1 add_button()

```
ui.main_window.Window.add_button (
              self,
             str text,
              function,
              row = 0,
              col = 0 )
```

Definition at line 15 of file main_window.py.

References ui.main_window.Window.window.

8.3.3.2 start_window()

```
ui.main_window.Window.start_window (
              self )
```

Definition at line 20 of file main_window.py.

References ui.main_window.Window.window.

8.3.4 Member Data Documentation

8.3.4.1 window

`ui.main_window.Window.window`

Definition at line 11 of file main_window.py.

Referenced by ui.main_window.Window.add_button(), and ui.main_window.Window.start_window().

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

- user/ui/main_window.py

Chapter 9

File Documentation

9.1 kernel-module/tcfs_kmodule.c File Reference

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

9.1.1 Detailed Description

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

Definition in file tcfs_kmodule.c.

9.2 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 */
```

9.3 README.md File Reference

9.4 ServerREST/crypt-utils/key-tools.go File Reference

9.5 key-tools.go

Go to the documentation of this file.

```
00001 package KeyTools
00002
00003 import (
00004     "crypto/rand"
00005     "crypto/rsa"
00006     "crypto/sha256"
00007     "crypto/x509"
00008     "encoding/hex"
00009     "encoding/pem"
00010     "errors"
00011     "fmt"
00012     "github.com/corvus-ch/shamir"
00013     TCFSTypes "serverTCFS/types"
00014 )
00015
00016 // GenerateKey Generate a AES 256 key
00017 func GenerateKey() ([]byte, error) {
00018     key := make([]byte, 32)
00019     _, err := rand.Read(key)
00020     if err != nil {
00021         return nil, err
00022     }
00023     return key, nil
00024 }
00025
00026 // SplitKey splits a key using Shamir's secret sharing
00027 func SplitKey(key []byte, n int, k int) (map[byte][]byte, error) {
00028     shares, err := shamir.Split(key, n, k)
00029     if err != nil {
00030         return nil, err
00031     }
00032     return shares, nil
00033 }
00034
00035 // parsePublicKeyFromPEMString Returns an rsa key froma pem string in PKIX format
00036 func parsePublicKeyFromPEMString(pubPEM string) (*rsa.PublicKey, error) {
00037     block, _ := pem.Decode([]byte(pubPEM))
00038     if block == nil {
00039         return nil, errors.New("failed to parse PEM block containing public key")
00040     }
00041
00042     pub, err := x509.ParsePKIXPublicKey(block.Bytes)
00043     if err != nil {
00044         return nil, err
00045     }
00046
00047     rsaPub, ok := pub.(*rsa.PublicKey)
00048     if !ok {
00049         return nil, errors.New("key type is not RSA")
00050     }
00051
00052     return rsaPub, nil
00053 }
00054
00055 // EncryptKeyPart Encrypts a keypart from shamir alg. with a public key
00056 func EncryptKeyPart(keyPart []byte, publicKey string) (string, error) {
00057     // Parse the public key
00058     pubKeyToRSA, err := parsePublicKeyFromPEMString(publicKey)
00059     if err != nil {
00060         return "", fmt.Errorf("failed to parse string to rsa key: %w", err)
00061     }
00062
00063     // Encrypt the key part using RSA-OAEP with SHA-256 hash function
00064     label := []byte("")
00065     hash := sha256.New()
00066     encryptedKeyPart, err := rsa.EncryptOAEP(hash, rand.Reader, pubKeyToRSA, keyPart, label)
00067     if err != nil {
00068         return "", fmt.Errorf("failed to encrypt key part: %w", err)
00069     }
00070
00071     return hex.EncodeToString(encryptedKeyPart), nil
00072 }
```

```
00073
00074 // EncryptSharesForSharedFile Encrypts all the keyparts from a slice of SharedFile structs
00075 func EncryptSharesForSharedFile(sharedFile *TCFSTypes.SharedFile) error {
00076     encryptedShare, err := EncryptKeyPart(sharedFile.Share, sharedFile.User.PublicKey)
00077     if err != nil {
00078         return err
00079     }
00080     sharedFile.EncryptedShare = encryptedShare
00081     return nil
00082 }
00083
00084 // EncryptSharesForSharedFileList same as EncryptSharesForSharedFile but works with slices
00085 func EncryptSharesForSharedFileList(list *[]TCFSTypes.SharedFile) error {
00086     for i := range *list {
00087         fmt.Printf("Encrypting share for %v\n", (*list)[i].User.Username)
00088         err := EncryptSharesForSharedFile(&(*list)[i])
00089         if err != nil {
00090             return err
00091         }
00092     }
00093     return nil
00094 }
```

9.6 ServerREST/db/db.go File Reference

9.7 db.go

Go to the documentation of this file.

```
00001 package DB
00002
00003 import (
00004     "database/sql"
00005     "errors"
00006     "fmt"
00007     _ "github.com/go-sql-driver/mysql"
00008     TCFSTypes "serverTCFS/types"
00009 )
00010
00011 var db *sql.DB
00012
00013 // Init initializes the MariaDB client with the specified options.
00014 func Init(host string, port string, dbname string, username string, password string) error {
00015     var err error
00016     dbConnectionString := username + ":" + password + "@tcp(" + host + ":" + port + ")/" + dbname
00017     db, err = sql.Open("mysql", dbConnectionString)
00018     if err != nil {
00019         fmt.Printf("ERR: %v\n", err)
00020         return err
00021     }
00022
00023     // Check if the connection is valid
00024     err = db.Ping()
00025     if err != nil {
00026         fmt.Printf("ERR: %v\n", err)
00027         return err
00028     }
00029
00030     fmt.Println("DB initialized")
00031     return nil
00032 }
00033
00034 // Close is a method to close the database connection
00035 func Close() error {
00036     err := db.Close()
00037     if err != nil {
00038         fmt.Printf("ERR: %v", err)
00039         return err
00040     }
00041     return nil
00042 }
00043
00044 // InsertRegisteredUser inserts a new registered user into the RegisteredUsers table.
00045 func InsertRegisteredUser(username string, passwordHash string) error {
00046     _, err := db.Exec("INSERT INTO RegisteredUsers (username, password_hash) VALUES (?, ?)", username,
    passwordHash)
00047     if err != nil {
00048         fmt.Printf("ERR: %v", err)
00049         return err
```

```
00050     }
00051     return nil
00052 }
00053
00054 // InsertLoggedUser inserts a new logged user into the LoggedUsers table.
00055 func InsertLoggedUser(username string, publicKey string) error {
00056     _, err := db.Exec("INSERT INTO LoggedUsers (username, public_key) VALUES (?, ?)", username,
      publicKey)
00057     if err != nil {
00058         fmt.Printf("ERR: %v", err)
00059         return err
00060     }
00061     return nil
00062 }
00063
00064 func DeleteLoggedUser(username string) error {
00065     _, err := db.Exec("DELETE FROM LoggedUsers WHERE username=?", username)
00066     if err != nil {
00067         fmt.Printf("ERR: %v", err)
00068         return err
00069     }
00070     return nil
00071 }
00072
00073 func GetPasswordHash(username string) (string, error) {
00074     // Query to obtain password hash
00075     var passwordHash string
00076     err := db.QueryRow("SELECT password_hash FROM RegisteredUsers WHERE username = ?",
      username).Scan(&passwordHash)
00077     if err != nil {
00078         fmt.Printf("ERR: %v", err)
00079         return "", err
00080     }
00081
00082     return passwordHash, nil
00083 }
00084
00085 // InsertSharedFile inserts a new shared file into the SharedFiles table.
00086 func InsertSharedFile(username string, fileID int, keypart string) error {
00087     _, err := db.Exec("INSERT INTO SharedFiles (username, fileID, keypart) VALUES (?, ?, ?)",
00088         username, fileID, keypart)
00089     if err != nil {
00090         fmt.Printf("ERR: %v", err)
00091         return err
00092     }
00093     return nil
00094 }
00095
00096 func GetNewFileID() (int, error) {
00097     var lastFileID int
00098
00099     // Esegui la stored procedure GetLastFileID
00100     _, err := db.Exec("CALL GetLastFileID(@output);")
00101     if err != nil {
00102         return 0, fmt.Errorf("failed to execute GetLastFileID: %w", err)
00103     }
00104
00105     // Recupera il valore di output
00106     err = db.QueryRow("SELECT @output").Scan(&lastFileID)
00107     if err != nil {
00108         return 0, fmt.Errorf("failed to get last file ID: %w", err)
00109     }
00110
00111     // Esegui la stored procedure IncrementLastFileID
00112     _, err = db.Exec("CALL IncrementLastFileID();")
00113     if err != nil {
00114         return 0, fmt.Errorf("failed to increment last file ID: %w", err)
00115     }
00116
00117     return lastFileID, nil
00118 }
00119
00120 // InsertMultipleSharedFiles Saves the shared files described by a slice of SharedFile structs in the
      DB
00121 func InsertMultipleSharedFiles(sharedFilesList []TCFSTypes.SharedFile) error {
00122
00123     for _, sharedFile := range sharedFilesList {
00124         err := InsertSharedFile(sharedFile.User.Username, sharedFile.FileID,
      sharedFile.EncryptedShare)
00125         if err != nil {
00126             return err
00127         }
00128     }
00129     return nil
00130 }
00131
00132 // LoadUserInfoByName retrieves user information from the LoggedUsers table based on the provided
```

```
    username.
00133 func LoadUserInfoByName(user *TCFSTypes.TCFSUser) error {
00134     // SQL query to retrieve information from LoggedUsers based on the username
00135     query := "SELECT username, public_key FROM LoggedUsers WHERE username = ?"
00136 
00137     // Execute the query
00138     row := db.QueryRow(query, user.Username)
00139 
00140     // Variables to store the query results
00141     var username string
00142     var publicKey string
00143 
00144     // Scan the results into the corresponding variable
00145     if err := row.Scan(&username, &publicKey); err != nil {
00146         if errors.Is(err, sql.ErrNoRows) {
00147             return fmt.Errorf("user not found: %s", user.Username)
00148         }
00149         return err
00150     }
00151 
00152     // Update the TCFSUser object with the retrieved information
00153     user.PublicKey = publicKey
00154 
00155     return nil
00156 }
```

9.8 ServerREST/main_test.go File Reference

9.9 main_test.go

Go to the documentation of this file.

```
00001 package main
00002 
00003 import (
00004     "bytes"
00005     "encoding/json"
00006     "fmt"
00007     "io/ioutil"
00008     "net/http"
00009     "testing"
00010 )
00011 
00012 func TestRegister(t *testing.T) {
00013     user := map[string]string{
00014         "username": "testUser",
00015         "password": "pass",
00016     }
00017 
00018     requestBody, err := json.Marshal(user)
00019     if err != nil {
00020         t.Fatal(err)
00021     }
00022     request, err := http.NewRequest("POST", "http://127.0.0.1:1234/register",
    bytes.NewBuffer(requestBody))
00023     if err != nil {
00024         t.Fatal(err)
00025     }
00026 
00027     // Esegui la richiesta HTTP
00028     client := &http.Client{}
00029     response, err := client.Do(request)
00030     if err != nil {
00031         t.Fatal(err)
00032     }
00033     defer response.Body.Close()
00034 
00035     // Leggi il corpo della risposta
00036     body, err := ioutil.ReadAll(response.Body)
00037     if err != nil {
00038         t.Fatal(err)
00039     }
00040 
00041     // Verifica che lo status code sia 200
00042     if response.StatusCode != http.StatusOK {
00043         t.Errorf("handler returned wrong status code: got %v want %v", response.StatusCode,
    http.StatusOK)
00044     }
00045 
```

```
00046     // Verifica che il messaggio di successo sia corretto
00047     expectedResponse := "User registered successfully"
00048     if string(body) != expectedResponse {
00049         t.Errorf("handler returned unexpected body: got %v want %v", string(body), expectedResponse)
00050     }
00051
00052 }
00053
00054 func TestLogin(t *testing.T) {
00055     // Crea una richiesta HTTP POST
00056     user := map[string]string{
00057         "username": "testUser",
00058         "password": "pass",
00059     }
00060     requestBody, err := json.Marshal(user)
00061     if err != nil {
00062         t.Fatal(err)
00063     }
00064     request, err := http.NewRequest("POST", "http://localhost:1234/login",
      bytes.NewBuffer(requestBody))
00065     if err != nil {
00066         t.Fatal(err)
00067     }
00068
00069     // Esegui la richiesta HTTP
00070     client := &http.Client{}
00071     response, err := client.Do(request)
00072     if err != nil {
00073         t.Fatal(err)
00074     }
00075     defer response.Body.Close()
00076
00077     // Leggi il corpo della risposta
00078     body, err := ioutil.ReadAll(response.Body)
00079     if err != nil {
00080         t.Fatal(err)
00081     }
00082
00083     // Verifica che lo status code sia 200
00084     if response.StatusCode != http.StatusOK {
00085         t.Errorf("handler returned wrong status code: got %v want %v", response.StatusCode,
      http.StatusOK)
00086     }
00087
00088     // Verifica che il messaggio di successo sia corretto
00089     expectedResponse := "User logged in successfully"
00090     if string(body) != expectedResponse {
00091         t.Errorf("handler returned unexpected body: got %v want %v", string(body), expectedResponse)
00092     }
00093 }
00094
00095 func TestLogout(t *testing.T) {
00096     // Crea una richiesta HTTP POST
00097     user := map[string]string{
00098         "username": "testUser",
00099     }
00100     requestBody, err := json.Marshal(user)
00101     if err != nil {
00102         t.Fatal(err)
00103     }
00104     request, err := http.NewRequest("POST", "http://localhost:1234/logout",
      bytes.NewBuffer(requestBody))
00105     if err != nil {
00106         t.Fatal(err)
00107     }
00108
00109     // Esegui la richiesta HTTP
00110     client := &http.Client{}
00111     response, err := client.Do(request)
00112     if err != nil {
00113         t.Fatal(err)
00114     }
00115     defer response.Body.Close()
00116
00117     // Leggi il corpo della risposta
00118     body, err := ioutil.ReadAll(response.Body)
00119     if err != nil {
00120         t.Fatal(err)
00121     }
00122
00123     // Verifica che lo status code sia 200
00124     if response.StatusCode != http.StatusOK {
00125         t.Errorf("handler returned wrong status code: got %v want %v", response.StatusCode,
      http.StatusOK)
00126     }
00127
00128     // Verifica che il messaggio di successo sia corretto
```

```
00129     expectedResponse := "User logged out successfully"
00130     if string(body) != expectedResponse {
00131         t.Errorf("handler returned unexpected body: got %v want %v", string(body), expectedResponse)
00132     }
00133 }
00134
00135 func TestShareFile(t *testing.T) {
00136     //register some users
00137     for i := 0; i < 10; i++ {
00138         user := map[string]string{
00139             "username": fmt.Sprintf("testUser%v", i),
00140             "password": "pass",
00141         }
00142
00143         requestBody, err := json.Marshal(user)
00144         if err != nil {
00145             t.Fatal(err)
00146         }
00147         request, err := http.NewRequest("POST", "http://127.0.0.1:1234/register",
      bytes.NewBuffer(requestBody))
00148         if err != nil {
00149             t.Fatal(err)
00150         }
00151
00152         // Esegui la richiesta HTTP
00153         client := &http.Client{}
00154         _, err = client.Do(request)
00155         if err != nil {
00156             t.Fatal(err)
00157         }
00158
00159     }
00160     //Log in the users
00161     //register some users
00162     for i := 0; i < 10; i++ {
00163         user := map[string]string{
00164             "username":  fmt.Sprintf("testUser%v", i),
00165             "password":  "pass",
00166             "publickey": "-----BEGIN PUBLIC
      KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzJy21YMY4pGMWRJwQvZe\n7gXIuvPS5JeCxXXn/6xCsC5aeHlojP+/nLd+f1339Vr
      PUBLIC KEY-----",
00167         }
00168
00169         requestBody, err := json.Marshal(user)
00170         if err != nil {
00171             t.Fatal(err)
00172         }
00173         request, err := http.NewRequest("POST", "http://127.0.0.1:1234/login",
      bytes.NewBuffer(requestBody))
00174         if err != nil {
00175             t.Fatal(err)
00176         }
00177
00178         // Esegui la richiesta HTTP
00179         client := &http.Client{}
00180         _, err = client.Do(request)
00181         if err != nil {
00182             t.Fatal(err)
00183         }
00184
00185     }
00186
00187     //Now share a file
00188     sharedFilesRequest := map[string]interface{}{
00189         "users": []map[string]string{
00190             {"username": "testUser0"},
00191             {"username": "testUser1"},
00192             {"username": "testUser2"},
00193             {"username": "testUser3"},
00194             {"username": "testUser4"},
00195             {"username": "testUser5"},
00196             {"username": "testUser6"},
00197             {"username": "testUser7"},
00198             {"username": "testUser8"},
00199             {"username": "testUser9"},
00200             // Aggiungi altri utenti qui se necessario
00201         },
00202         "k": 4,
00203     }
00204
00205     requestBody, err := json.Marshal(sharedFilesRequest)
00206     if err != nil {
00207         t.Fatal(err)
00208     }
00209     request, err := http.NewRequest("POST", "http://localhost:1234/createSharedFile",
      bytes.NewBuffer(requestBody))
00210
```

```
00211     // Esegui la richiesta HTTP
00212     client := &http.Client{}
00213     response, err := client.Do(request)
00214     if err != nil {
00215         t.Fatal(err)
00216     }
00217     defer response.Body.Close()
00218 
00219     // Leggi il corpo della risposta
00220     body, err := ioutil.ReadAll(response.Body)
00221     if err != nil {
00222         t.Fatal(err)
00223     }
00224 
00225     // Verifica che lo status code sia 200
00226     if response.StatusCode != http.StatusOK {
00227         t.Errorf("handler returned wrong status code: got %v want %v", response.StatusCode,
      http.StatusOK)
00228     }
00229 
00230     fmt.Printf("%v\n", string(body))
00231 
00232 }
```

9.10 ServerREST/serverTools/REST_functions.go File Reference

9.11 REST_functions.go

Go to the documentation of this file.

```
00001 package REST_Functions
00002 
00003 import (
00004     "encoding/json"
00005     "fmt"
00006     "golang.org/x/crypto/bcrypt"
00007     "net/http"
00008     KeyTools "serverTCFS/crypt-utils"
00009     DB "serverTCFS/db"
00010     TCFSTypes "serverTCFS/types"
00011 )
00012 
00013 /**
00014  * @brief Descrizione della funzione.
00015  * @param param1 Descrizione del parametro 1.
00016  * @param param2 Descrizione del parametro 2.
00017  * @return Descrizione del valore di ritorno.
00018  */
00019 func deserializeUser(r *http.Request) (TCFSTypes.TCFSUser, error) {
00020     var user TCFSTypes.TCFSUser
00021     err := json.NewDecoder(r.Body).Decode(&user)
00022     if err != nil {
00023         return user, err
00024     }
00025     return user, nil
00026 }
00027 
00028 func Register(w http.ResponseWriter, r *http.Request) {
00029     fmt.Printf("Called Register")
00030     var user TCFSTypes.TCFSUser
00031     var err error = nil
00032     user, err = deserializeUser(r)
00033     if err != nil {
00034         http.Error(w, err.Error(), http.StatusBadRequest)
00035         return
00036     }
00037 
00038     // Hash the user's password
00039     hashedPassword, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost)
00040     if err != nil {
00041         http.Error(w, err.Error(), http.StatusInternalServerError)
00042         return
00043     }
00044 
00045     // Insert the user into the RegisteredUsers table
00046     err = DB.InsertRegisteredUser(user.Username, string(hashedPassword))
00047     if err != nil {
00048         http.Error(w, err.Error(), http.StatusInternalServerError)
00049         return
```

```
00050     }
00051     fmt.Printf("New user inserted \nSUCCESS\n")
00052
00053     // Return a success message
00054     w.WriteHeader(http.StatusOK)
00055     fmt.Fprintf(w, "User registered successfully")
00056 }
00057
00058 func Login(w http.ResponseWriter, r *http.Request) {
00059     fmt.Println("Called Login\n")
00060     var user TCFSTypes.TCFSUser
00061     var err error = nil
00062     user, err = deserializeUser(r)
00063     if err != nil {
00064         http.Error(w, err.Error(), http.StatusBadRequest)
00065         return
00066     }
00067
00068     // Retrieve the user's hashed password from the RegisteredUsers table
00069     hashedPassword, err := DB.GetPasswordHash(user.Username)
00070     if err != nil {
00071         http.Error(w, err.Error(), http.StatusInternalServerError)
00072         return
00073     }
00074
00075     // Compare the user's password with the hashed password
00076     err = bcrypt.CompareHashAndPassword([]byte(hashedPassword), []byte(user.Password))
00077     if err != nil {
00078         http.Error(w, "Invalid credentials", http.StatusUnauthorized)
00079         return
00080     }
00081     fmt.Println("Password match")
00082
00083     // Insert the user into the LoggedUsers table
00084     err = DB.InsertLoggedUser(user.Username, user.PublicKey)
00085     if err != nil {
00086         http.Error(w, err.Error(), http.StatusInternalServerError)
00087         return
00088     }
00089     fmt.Println("Inserted in logged users")
00090
00091     // Return a success message
00092     w.WriteHeader(http.StatusOK)
00093     fmt.Fprintf(w, "User logged in successfully")
00094 }
00095
00096 func Logout(w http.ResponseWriter, r *http.Request) {
00097     fmt.Println("Called Unregister")
00098     var user TCFSTypes.TCFSUser
00099     var err error = nil
00100     user, err = deserializeUser(r)
00101     if err != nil {
00102         http.Error(w, err.Error(), http.StatusBadRequest)
00103         return
00104     }
00105
00106     err = DB.DeleteLoggedUser(user.Username)
00107     if err != nil {
00108         http.Error(w, err.Error(), http.StatusInternalServerError)
00109     }
00110     fmt.Printf("User %v unregistered\n SUCCESS\n", user.Username)
00111     w.WriteHeader(http.StatusOK)
00112     fmt.Fprintf(w, "User logged out successfully")
00113 }
00114
00115 /*
00116 CreateSharedFile The request contains a username list and the k number for Shamir
00117 A new key for the file will be generated and then Shamir generates all the key-parts
00118 Each key-part is cyphered with the public key of the user and saved in the relative entry in the DB
00119 A fileID is returned in the response. This will identify the file
00120 */
00121 func CreateSharedFile(w http.ResponseWriter, r *http.Request) {
00122
00123     type User struct {
00124         Username string `json:"username"`
00125     }
00126
00127     type Request struct {
00128         Users []User `json:"users"`
00129         K     int    `json:"k"`
00130     }
00131
00132     var req Request
00133
00134     decoder := json.NewDecoder(r.Body)
00135     err := decoder.Decode(&req)
00136     if err != nil {
```

```
00137         http.Error(w, err.Error(), http.StatusBadRequest)
00138         return
00139     }
00140
00141     var users []TCFSTypes.TCFSUser
00142     for _, user := range req.Users {
00143         tmpUser := TCFSTypes.TCFSUser{Username: user.Username}
00144         err := DB.LoadUserInfoByName(&tmpUser)
00145         if err != nil {
00146             http.Error(w, err.Error(), http.StatusBadRequest)
00147             fmt.Printf("Could not load user %v info %v\n", user, err)
00148             return
00149         }
00150         users = append(users, tmpUser)
00151     }
00152
00153     k := req.K
00154
00155     fmt.Printf("Got users and k: %v\n", k)
00156
00157     // Generate a new key
00158     key, err := KeyTools.GenerateKey()
00159     if err != nil {
00160         fmt.Printf("Err: cannot not generate new key %v\n", err)
00161         http.Error(w, err.Error(), http.StatusInternalServerError)
00162         return
00163     }
00164
00165     // Split the key using Shamir's secret sharing
00166     shares, err := KeyTools.SplitKey(key, len(users), k)
00167     if err != nil {
00168         fmt.Printf("Cannot split the key %v\n", err)
00169         http.Error(w, err.Error(), http.StatusInternalServerError)
00170         return
00171     }
00172     fmt.Printf("Got %v keyparts for %v users\n", len(shares), len(users))
00173
00174     fileID, err := DB.GetNewFileID()
00175     if err != nil {
00176         fmt.Printf("Cannot generate a new fileID %v\n", err)
00177         http.Error(w, err.Error(), http.StatusInternalServerError)
00178         return
00179     }
00180     fmt.Printf("Got new file id %v\n", fileID)
00181
00182     var sharedFilesList []TCFSTypes.SharedFile
00183     // Couple the shares with the user in the sharedFilesList
00184     j := 0
00185     for _, share := range shares {
00186         if share == nil {
00187             fmt.Printf("This share is nil\n")
00188             http.Error(w, err.Error(), http.StatusInternalServerError)
00189             return
00190         }
00191         sharedFile := TCFSTypes.SharedFile{
00192             User:           users[j],
00193             FileID:         fileID,
00194             Share:          share,
00195             EncryptedShare: "",
00196         }
00197         sharedFilesList = append(sharedFilesList, sharedFile)
00198         j++
00199     }
00200     fmt.Printf("Created %v shared files \n", len(sharedFilesList))
00201
00202     for _, s := range sharedFilesList {
00203         fmt.Printf("%v\n", s)
00204     }
00205
00206     err = KeyTools.EncryptSharesForSharedFileList(&sharedFilesList)
00207     if err != nil {
00208         fmt.Printf("Err: cannot Encrypt share list: %v\n", err)
00209         http.Error(w, err.Error(), http.StatusInternalServerError)
00210         return
00211     }
00212
00213     err = DB.InsertMultipleSharedFiles(sharedFilesList)
00214     if err != nil {
00215         fmt.Printf("Err: cannot save list in DB  %v\n", err)
00216         http.Error(w, err.Error(), http.StatusInternalServerError)
00217         return
00218     }
00219     fmt.Printf("New share saved to DB\n")
00220
00221     // Return a success message
00222     w.WriteHeader(http.StatusOK)
00223     fmt.Fprintf(w, fmt.Sprintf("fileID:%v\n", fileID))
```

```
00224     fmt.Fprintf(w, "Shared file created successfully")
00225 }
```

9.12 ServerREST/tcfs-daemon.go File Reference

9.13 tcfs-daemon.go

Go to the documentation of this file.

```
00001 package main
00002
00003 /**
00004  * @file main.go
00005  * @brief Main file for the TCFS server.
00006  */
00007
00008 import (
00009     "flag"
00010     "fmt"
00011     "gopkg.in/yaml.v2"
00012     "io"
00013     "io/ioutil"
00014     "log"
00015     "net/http"
00016     "os"
00017     DB "serverTCFS/db"
00018     restfunctions "serverTCFS/serverTools"
00019 )
00020
00021 /**
00022  * @struct serverConfig
00023  * @brief Configuration structure for the server.
00024  */
00025 type serverConfig struct {
00026     Server struct {
00027         Port string `yaml:"port"`
00028     } `yaml:"Server"`
00029     DB struct {
00030         Host     string `yaml:"host"`
00031         Port     string `yaml:"port"`
00032         DBname   string `yaml:"dbname"`
00033         Username string `yaml:"username"`
00034         Password string `yaml:"password"`
00035     } `yaml:"db"`
00036 }
00037
00038 /**
00039  * @brief Main function to start the TCFS server.
00040  */
00041 func main() {
00042     // Parse command-line flags for the Server port
00043     var configFile string
00044     flag.StringVar(&configFile, "config-file", "config.yaml", "The location of the rest server config
      file")
00045     flag.Parse()
00046
00047     // Read the YAML file
00048     data, err := ioutil.ReadFile(configFile)
00049     if err != nil {
00050         log.Fatal(err)
00051     }
00052
00053     // Unmarshal the YAML data into a Config struct
00054     var config serverConfig
00055     err = yaml.Unmarshal(data, &config)
00056     if err != nil {
00057         log.Fatal(err)
00058     }
00059
00060     // Create a new log file
00061     file, err := os.OpenFile("/tmp/tcfs-daemon.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
00062     if err != nil {
00063         log.Fatal(err)
00064     }
00065     defer file.Close()
00066
00067     // Create a multi-writer that writes to both stdout and the log file
00068     multiWriter := io.MultiWriter(os.Stdout, file)
00069
```

```
00070     // Set the logger to write to the multi-writer
00071     logger := log.New(multiWriter, "", log.LstdFlags)
00072
00073     err = DB.Init(config.DB.Host, config.DB.Port, config.DB.DBname, config.DB.Username,
      config.DB.Password)
00074     if err != nil {
00075         fmt.Printf("Err initializing the DB: %v", err)
00076         return
00077     }
00078     http.HandleFunc("/register", restfunctions.Register)
00079     http.HandleFunc("/login", restfunctions.Login)
00080     http.HandleFunc("/logout", restfunctions.Logout)
00081     http.HandleFunc("/createSharedFile", restfunctions.CreateSharedFile)
00082     fmt.Printf("serving on %v\n", config.Server.Port)
00083     log.Fatal(http.ListenAndServe(":"+config.Server.Port, nil))
00084
00085     // Terminate the program
00086     logger.Println("Server is exiting")
00087 }
```

9.14 ServerREST/types/tcfs-user.go File Reference

9.15 tcfs-user.go

Go to the documentation of this file.

```
00001 package TCFSTypes
00002
00003 import "fmt"
00004
00005 type TCFSUser struct {
00006     Username  string
00007     Password  string
00008     PublicKey string
00009 }
00010
00011 type SharedFile struct {
00012     User           TCFSUser
00013     FileID         int
00014     Share          []byte
00015     EncryptedShare string
00016 }
00017
00018 func (s SharedFile) String() string {
00019     return fmt.Sprintf(""+
00020         "------------------------------------------------------------\n"+
00021         "User:\n"+
00022         "   Name%v\n"+
00023         "   Pubkey%v\n"+
00024         "   Pass%v\n"+
00025         "FileID: %v\n"+
00026         "Share: %v\n"+
00027         "EncryptedShare %v\n"+
00028         "------------------------------------------------------------"+
00029         "", s.User.Username, s.User.PublicKey, s.User.Password, s.FileID, s.Share, s.EncryptedShare)
00030 }
```

9.16 user/command_handler/enviroinment.py File Reference

Namespaces

- namespace command_handler
- namespace command_handler.enviroinment

Functions

- command_handler.enviroinment.countdown (msg, delta)
- command_handler.enviroinment.init_env ()

9.17 enviroinment.py

Go to the documentation of this file.

```
00001 import os
00002 import shutil
00003 import time
00004
00005
00006 def countdown(msg, delta):
00007     for i in range(delta, 0, -1):
00008         print(f"{msg} {delta}")
00009         time.sleep(1)
00010
00011
00012 def init_env():
00013     tcfs_folder = "~/.tcfs"
00014     data_folder = tcfs_folder + "/data"
00015
00016     if os.path.exists(tcfs_folder) and os.path.isdir(tcfs_folder):
00017         print("WARN: Deleting main tcfs folder, all your data will be lost")
00018         countdown("\r Deleting in ....", 5)
```

9.18 user/main.py File Reference

Namespaces

- namespace main

Functions

- main.foo ()

Variables

- main.win = Window("TCFS user helper", "200x200")
- main.init_env_butt = win.add_button("Initialize the environment", foo, row=0, col=0)
- main.mount_butt = win.add_button("Mount TCFS", foo, row=1, col=0)
- main.umount_butt = win.add_button("Umount TCFS", foo, row=2, col=0)
- main.shared_butt = win.add_button("Threshold share", foo, row=3, col=0)
- main.logout_butt = win.add_button("Logout", foo, row=4, col=0)

9.19 main.py

Go to the documentation of this file.

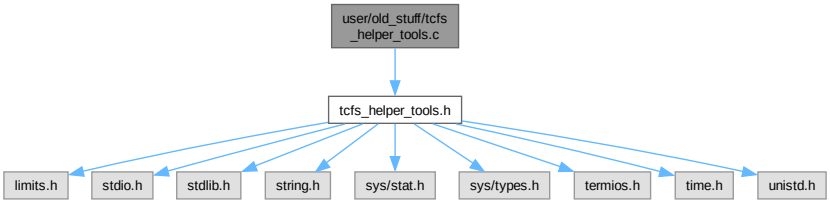
```
00001 from ui.main_window import Window
00002
00003
00004 def foo():
00005     print("Hello World")
00006
00007
00008 win = Window("TCFS user helper", "200x200")
00009
00010 init_env_butt = win.add_button("Initialize the environment", foo, row=0, col=0) #init enviroinment
00011 mount_butt = win.add_button("Mount TCFS", foo, row=1, col=0) #mount tcfs
00012 umount_butt = win.add_button("Umount TCFS", foo, row=2, col=0) #umount
00013 shared_butt = win.add_button("Threshold share", foo, row=3, col=0) #create shared file
00014 logout_butt = win.add_button("Logout", foo, row=4, col=0) #logout
00015
00016
00017 win.start_window()
```

9.20 user/old_stuff/tcfs_helper_tools.c File Reference

This file contains the logic and implementation of functions needed to perform the operations requested in user_tcfs.c.

```
#include "tcfs_helper_tools.h"
```

Include dependency graph for tcfs_helper_tools.c:



Macros

- #define PASS_SIZE 33

  Set the password size to 33 as AES 256 uses a 32 byte key. An \0 is added for safety. This definition is marked as internal and should not be used directly by the user.

Functions

- int handle_local_mount ()
- int handle_remote_mount ()
- int handle_folder_mount ()
- int do_mount ()

  Execute the mount of either a Network FS (for ex NFS), Local FS (for ex a block device), Local folder (a folder of the system)
- int generate_random_string (char ∗str)

  Generate a random string that will be used as a folder name for the mount. This function is marked as internal and should not be used by the user.
- int directory_exists (const char ∗path)

  Check if a directory already exists. This function is marked as internal and should not be used by the user.
- char ∗ setup_tcfs_mount_folder ()

  Creates the .tcfs folder in the user HOME. Then this creates a folder with a random name so that create_tcfs_mount↩ _local_folder function can use it. create_tcfs_mount_local_folder This function is marked as internal and should not be used by the user.
- void get_pass (char ∗pw)

  fetch the password of the current user. This function is marked as internal and should not be used by the user
- void get_source_dest (char ∗source, char ∗dest)
- char ∗ create_tcfs_mount_local_folder ()

  Create a temporary folder in HOME/.tcfs.
- int mount_tcfs_folder (char ∗tmp_path, char ∗destination)
- void clearKeyboardBuffer ()
- int setup_tcfs_env ()

9.20.1 Detailed Description

This file contains the logic and implementation of functions needed to perform the operations requested in user_tcfs.c.

See also

user_tcfs.c

Definition in file tcfs_helper_tools.c.

9.20.2 Macro Definition Documentation

9.20.2.1 PASS_SIZE

```
#define PASS_SIZE 33
```

Set the password size to 33 as AES 256 uses a 32 byte key. An \0 is added for safety. This definition is marked as internal and should not be used directly by the user.

Definition at line 14 of file tcfs_helper_tools.c.

9.20.3 Function Documentation

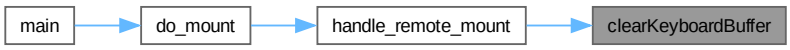
9.20.3.1 clearKeyboardBuffer()

```
void clearKeyboardBuffer ( )
```

Definition at line 339 of file tcfs_helper_tools.c.

Referenced by handle_remote_mount().

Here is the caller graph for this function:



9.20.3.2 create_tcfs_mount_local_folder()

```
char * create_tcfs_mount_local_folder ( )
```

Create a temporary folder in HOME/.tcfs.

Returns

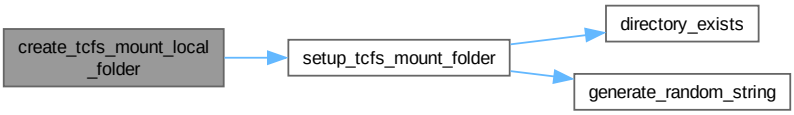
char * the fullpath to the generated folder

Definition at line 219 of file tcfs_helper_tools.c.

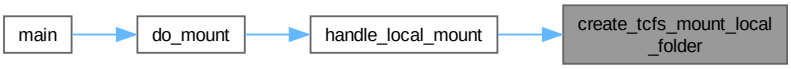
References setup_tcfs_mount_folder().

Referenced by handle_local_mount().

Here is the call graph for this function:



Here is the caller graph for this function:



9.20.3.3 directory_exists()

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

Check if a directory already exists. This function is marked as internal and should not be used by the user.

Parameters

path	Fullpath to the dir

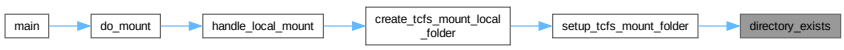
Returns

int _ret

Definition at line 83 of file tcfs_helper_tools.c.

Referenced by setup_tcfs_mount_folder().

Here is the caller graph for this function:



9.20.3.4 do_mount()

```
int do_mount ( )
```

Execute the mount of either a Network FS (for ex NFS), Local FS (for ex a block device), Local folder (a folder of the system)

Returns

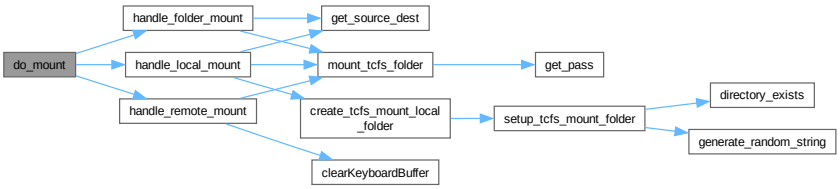
_ret

Definition at line 25 of file tcfs_helper_tools.c.

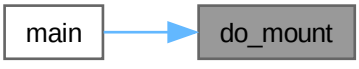
References handle_folder_mount(), handle_local_mount(), and handle_remote_mount().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



9.20.3.5 generate_random_string()

```
int generate_random_string (
            char * str )
```

Generate a random string that will be used as a folder name for the mount. This function is marked as internal and should not be used by the user.

Parameters

str	The result will be saved here

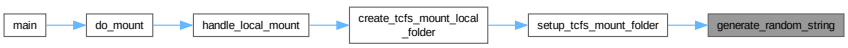
Returns

int _ret

Definition at line 64 of file tcfs_helper_tools.c.

Referenced by setup_tcfs_mount_folder().

Here is the caller graph for this function:



9.20.3.6 get_pass()

```
void get_pass (
            char * pw )
```

fetch the password of the current user. This function is marked as internal and should not be used by the user

Parameters

pw	String to save the password to

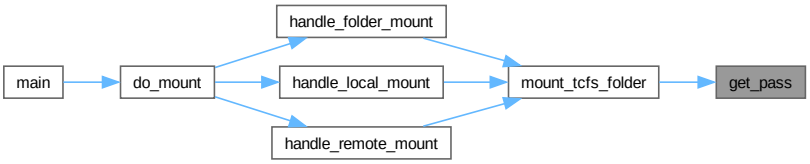
Todo This will be changed when a public/private key model will be avilable to TCFS userspace module

Definition at line 162 of file tcfs_helper_tools.c.

References PASS_SIZE.

Referenced by mount_tcfs_folder().

Here is the caller graph for this function:



9.20.3.7 get_source_dest()

```
void get_source_dest (
            char * source,
            char * dest )
```

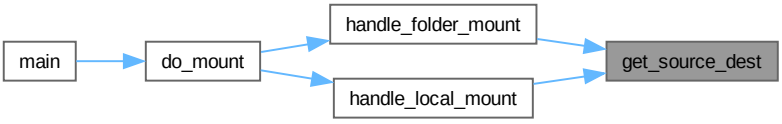
Parameters

Ask	the user to insert the source and the destination of the mount. This function is marked as internal and should not be used by the user

Definition at line 204 of file tcfs_helper_tools.c.

Referenced by handle_folder_mount(), and handle_local_mount().

Here is the caller graph for this function:



9.20.3.8 handle_folder_mount()

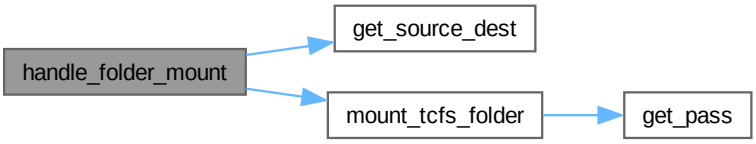
```
int handle_folder_mount ( )
```

Definition at line 318 of file tcfs_helper_tools.c.

References get_source_dest(), and mount_tcfs_folder().

Referenced by do_mount().

Here is the call graph for this function:



Here is the caller graph for this function:

9.20.3.9 handle_local_mount()

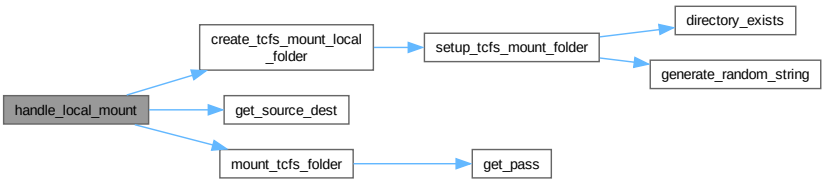
```
int handle_local_mount ( )
```

Definition at line 274 of file tcfs_helper_tools.c.

References create_tcfs_mount_local_folder(), get_source_dest(), and mount_tcfs_folder().

Referenced by do_mount().

Here is the call graph for this function:



Here is the caller graph for this function:

9.20.3.10 handle_remote_mount()

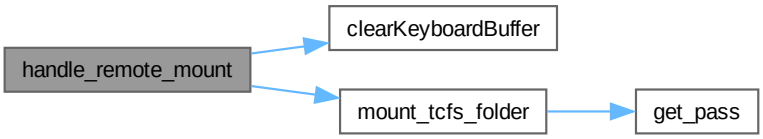
```
int handle_remote_mount ( )
```

Definition at line 347 of file tcfs_helper_tools.c.

References clearKeyboardBuffer(), and mount_tcfs_folder().

Referenced by do_mount().

Here is the call graph for this function:



Here is the caller graph for this function:

9.20.3.11 mount_tcfs_folder()

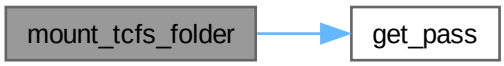
```
int mount_tcfs_folder (
            char * tmp_path,
            char * destination )
```

Definition at line 235 of file tcfs_helper_tools.c.

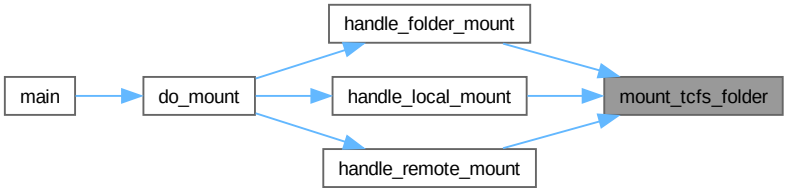
References get_pass(), and PASS_SIZE.

Referenced by handle_folder_mount(), handle_local_mount(), and handle_remote_mount().

Here is the call graph for this function:



Here is the caller graph for this function:



9.20.3.12 setup_tcfs_env()

```
int setup_tcfs_env ( )
```

Definition at line 405 of file tcfs_helper_tools.c.

Referenced by main().

Here is the caller graph for this function:

9.20.3.13 setup_tcfs_mount_folder()

```
char * setup_tcfs_mount_folder ( )
```

Creates the .tcfs folder in the user HOME. Then this creates a folder with a random name so that create_tcfs↩ _mount_local_folder function can use it. create_tcfs_mount_local_folder This function is marked as internal and should not be used by the user.

Returns

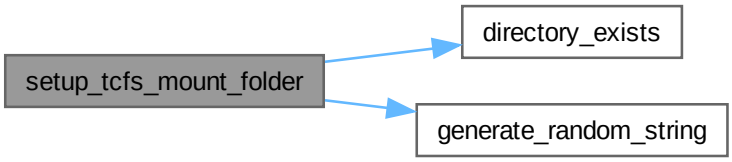
a folder with a random name inside the $HOME/.tcfs folder

Definition at line 95 of file tcfs_helper_tools.c.

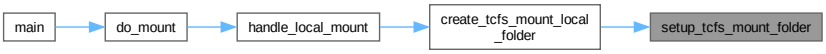
References directory_exists(), and generate_random_string().

Referenced by create_tcfs_mount_local_folder().

Here is the call graph for this function:



Here is the caller graph for this function:



9.21 tcfs_helper_tools.c

Go to the documentation of this file.

```
00001 #include "tcfs_helper_tools.h"
00002
00014 #define PASS_SIZE 33
00015
00016 int handle_local_mount ();
00017 int handle_remote_mount ();
00018 int handle_folder_mount ();
00019
00024 int
00025 do_mount ()
00026 {
00027   int choice = -1;
00028   do
```

```
00029     {
00030       printf ("Chose between:\n"
00031               "\t1. Network FS\n"
00032               "\t2. Local FS\n"
00033               "\t3. Local folder");
00034       scanf ("%d", &choice);
00035       if (choice != 1 && choice != 2 && choice != 3)
00036         printf ("Err: Select 1 or 2\n");
00037     }
00038   while (choice != 1 && choice != 2 && choice != 3);
00039   printf ("You chose %d\n", choice);
00040
00041   if (choice == 1)
00042     {
00043       return handle_remote_mount ();
00044     }
00045   else if (choice == 2)
00046     {
00047       return handle_local_mount ();
00048     }
00049   else if (choice == 3)
00050     {
00051       return handle_folder_mount ();
00052     }
00053   printf ("Unrecoverable error\n");
00054   return 0;
00055 }
00056
00063 int
00064 generate_random_string (char *str)
00065 {
00066   srand (time (NULL));
00067   if (str == NULL)
00068     return 0;
00069   for (int i = 0; i < 10; i++)
00070     str[i] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
00071         [rand () % 62];
00072   str[10] = '\0';
00073   return 1;
00074 }
00075
00082 int
00083 directory_exists (const char *path)
00084 {
00085   struct stat sb;
00086   return stat (path, &sb) == 0 && S_ISDIR (sb.st_mode);
00087 }
00088
00094 char *
00095 setup_tcfs_mount_folder ()
00096 {
00097   printf ("SETUP ENV\n");
00098   char *home = getenv ("HOME");
00099   printf ("$HOME=%s\n", home);
00100
00101   char *tcfs_path
00102       = malloc ((strlen (home) + strlen ("/.tcfs\0")) * sizeof (char));
00103   char rand_path_name[11];
00104   char *new_path = NULL;
00105
00106   if (home == NULL)
00107     {
00108       perror ("Could not get $HOME\n");
00109       return 0;
00110     }
00111
00112   if (tcfs_path == NULL)
00113     {
00114       perror ("Could not allocate string tcfs_path");
00115       return 0;
00116     }
00117   sprintf (tcfs_path, "%s/%s", home, ".tcfs");
00118
00119   //$HOME/.tcfs does not exist if this is true
00120   if (directory_exists (tcfs_path) == 0)
00121     {
00122       if (mkdir (tcfs_path, 0770) == -1)
00123         {
00124           perror ("Cannot create .tcfs directory");
00125           return 0;
00126         }
00127     }
00128   // Create a folder to mount the source to
00129   // Generate a random path name
00130   if (generate_random_string (rand_path_name) == 0)
00131     {
00132       fprintf (stderr, "Err: Name generation for temp folder failed\n");
```

```
00133       return 0;
00134     }
00135   // Build the path from / to the generated path
00136   new_path = malloc ((strlen (rand_path_name) + strlen (tcfs_path) + 1)
00137                      * sizeof (char));
00138   if (new_path == NULL)
00139     {
00140       perror ("Cannot allocate new memory for path name");
00141       return 0;
00142     }
00143   sprintf (new_path, "%s/%s", tcfs_path, rand_path_name);
00144   if (mkdir (new_path, 0770) == -1)
00145     {
00146       perror ("Cannot create the tmp folder inside .tcfs");
00147       return 0;
00148     }
00149
00150   printf ("New path %s\n", new_path);
00151   free (tcfs_path);
00152   return new_path;
00153 }
00154
00161 void
00162 get_pass (char *pw)
00163 {
00164   struct termios old, new;
00165   int i = 0;
00166   int ch = 0;
00167
00168   // Disable character echo
00169   tcgetattr (STDIN_FILENO, &old);
00170   new = old;
00171   new.c_lflag &= ~ECHO;
00172   tcsetattr (STDIN_FILENO, TCSANOW, &new);
00173
00174   printf ("Please enter a password exactly %d characters long:\n", PASS_SIZE);
00175
00176   while (strlen (pw) * sizeof (char) < (PASS_SIZE - 1) * sizeof (char))
00177     {
00178       while (1)
00179         {
00180           ch = getchar ();
00181           if (ch == '\r' || ch == '\n' || ch == EOF)
00182             {
00183               break;
00184             }
00185           if (i < PASS_SIZE - 1)
00186             {
00187               pw[i] = ch;
00188               pw[i + 1] = '\0';
00189             }
00190           i++;
00191         }
00192     }
00193
00194   // Restore terminal settings
00195   tcsetattr (STDIN_FILENO, TCSANOW, &old);
00196   printf ("\nPassword successfully entered!\n");
00197 }
00198
00203 void
00204 get_source_dest (char *source, char *dest)
00205 {
00206   printf ("Please type the path to the source\n");
00207   scanf ("%s", source);
00208
00209   printf ("Please type where it should be mounted\n");
00210   scanf ("%s", dest);
00211 }
00212
00218 char *
00219 create_tcfs_mount_local_folder ()
00220 {
00221   char *tmp_path = NULL;
00222
00223   // Create a folder to mount to
00224   tmp_path = setup_tcfs_mount_folder ();
00225   if (tmp_path == NULL)
00226     {
00227       fprintf (stderr, "Err: could not get temp path\n");
00228       return NULL;
00229     }
00230   printf ("Creating dir: %s\n", tmp_path);
00231   return tmp_path;
00232 }
00233
00234 int
```

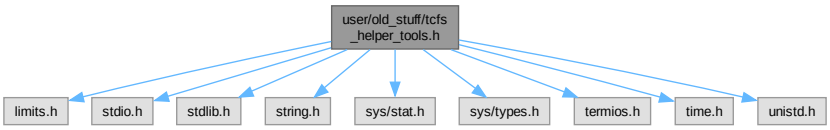
```
00235 mount_tcfs_folder (char *tmp_path, char *destination)
00236 {
00237   char pass[PASS_SIZE] = "\0";
00238   struct termios old, new;
00239
00240   // Disable character echo
00241   tcgetattr (STDIN_FILENO, &old);
00242   new = old;
00243   new.c_lflag &= ~ECHO;
00244   tcsetattr (STDIN_FILENO, TCSANOW, &new);
00245
00246   get_pass (pass);
00247   if (pass[0] == '\0')
00248     {
00249       tcsetattr (STDIN_FILENO, TCSANOW, &old);
00250       fprintf (stderr, "Could not get password\n");
00251       return 0;
00252     }
00253
00254   // Mount tmpfolder to the destination
00255   char *tcfs_command
00256       = malloc ((strlen ("tcfs -s ") + strlen (tmp_path) + strlen (" -d ")
00257                  + strlen (destination) + strlen (" -p ") + strlen (pass)));
00258   sprintf (tcfs_command, "tcfs -s %s -d %s -p %s", tmp_path, destination,
00259            pass);
00260
00261   int status_tcfs_mount = system (tcfs_command);
00262   if (!(WIFEXITED (status_tcfs_mount) && WEXITSTATUS (status_tcfs_mount) == 0))
00263     {
00264       tcsetattr (STDIN_FILENO, TCSANOW, &old);
00265       perror ("Could not execute the command");
00266       return 0;
00267     }
00268   free (tcfs_command);
00269   tcsetattr (STDIN_FILENO, TCSANOW, &old);
00270   return 1;
00271 }
00272
00273 int
00274 handle_local_mount ()
00275 {
00276   char source[PATH_MAX];
00277   char destination[PATH_MAX];
00278   char *tmp_path = NULL;
00279
00280   get_source_dest (source, destination);
00281
00282   tmp_path = create_tcfs_mount_local_folder ();
00283   if (tmp_path == NULL)
00284     {
00285       printf ("Err: could not get tmp folder path\n");
00286       return 0;
00287     }
00288
00289   // Mount block device to temp folder
00290   char *command = malloc (
00291       (strlen ("mount ") + strlen (source) + strlen (" ") + strlen (tmp_path))
00292       * sizeof (char));
00293   if (command == NULL)
00294     {
00295       perror ("cannot allocate memoty for the command");
00296       return 0;
00297     }
00298   sprintf (command, "sudo mount -o umask=0755,gid=1000,uid=1000 %s %s", source,
00299            tmp_path);
00300   printf ("executing: %s\n", command);
00301   int status_tmp_mount = system (command);
00302   if (!(WIFEXITED (status_tmp_mount) && WEXITSTATUS (status_tmp_mount) == 0))
00303     {
00304       perror ("Could not execute the command");
00305       return 0;
00306     }
00307
00308   int res = mount_tcfs_folder (tmp_path, destination);
00309   if (res == 0)
00310     return 0;
00311
00312   free (tmp_path);
00313   free (command);
00314   return 1;
00315 }
00316
00317 int
00318 handle_folder_mount ()
00319 {
00320   char source[PATH_MAX];
00321   char destination[PATH_MAX];
```

```
00322
00323   get_source_dest (source, destination);
00324   if (source[0] == '\0' || destination[0] == '\0')
00325     {
00326       printf ("Err: Could not get source or destination\n");
00327       return 0;
00328     }
00329   printf ("Source:%s\tdestination:%s\n", source, destination);
00330
00331   int res = mount_tcfs_folder (source, destination);
00332   if (res == 0)
00333     return 0;
00334
00335   return 1;
00336 }
00337
00338 void
00339 clearKeyboardBuffer ()
00340 {
00341   int ch;
00342   while ((ch = getchar ()) != EOF && ch != '\n')
00343     ;
00344 }
00345
00346 int
00347 handle_remote_mount ()
00348 {
00349   char source[PATH_MAX] = "\0";
00350   char destination[PATH_MAX] = "\0";
00351   char command[100] = "\0";
00352
00353   printf ("WARN: This function is not complete, I don't know how many remote "
00354           "FileSystems support extended "
00355           "attributes, please mount it manually. "
00356           "\nEX:sudo mount -t nfs -o umask=0755,gid=1000,uid=1000 "
00357           "10.10.10.10:/NFS /mnt\n");
00358
00359   clearKeyboardBuffer ();
00360   printf ("Enter the command: ");
00361   int ch;
00362   int loop = 0;
00363   while (loop < 99 && (ch = getc (stdin)) != EOF && ch != '\n')
00364     {
00365       command[loop] = ch;
00366       ++loop;
00367     }
00368   command[loop] = '\0'; // Null-terminate the string
00369
00370   printf ("Command: %s\n", command);
00371   int status = system (command);
00372   if (!(WIFEXITED (status) && WEXITSTATUS (status) == 0))
00373     {
00374       perror ("Could not execute the command");
00375       return 0;
00376     }
00377
00378   printf ("Where has it been mounted? ");
00379   loop = 0;
00380   while (loop < PATH_MAX - 1 && (ch = getc (stdin)) != EOF && ch != '\n')
00381     {
00382       source[loop] = ch;
00383       ++loop;
00384     }
00385   source[loop] = '\0'; // Null-terminate the string
00386
00387   printf ("Source: %s\n", source);
00388
00389   printf ("Where should TCFS mount it? ");
00390   loop = 0;
00391   while (loop < PATH_MAX - 1 && (ch = getc (stdin)) != EOF && ch != '\n')
00392     {
00393       destination[loop] = ch;
00394       ++loop;
00395     }
00396   destination[loop] = '\0'; // Null-terminate the string
00397
00398   printf ("Destination: %s\n", destination);
00399
00400   int res = mount_tcfs_folder (source, destination);
00401   return res;
00402 }
00403
00404 int
00405 setup_tcfs_env ()
00406 {
00407   return 0;
00408 }
```

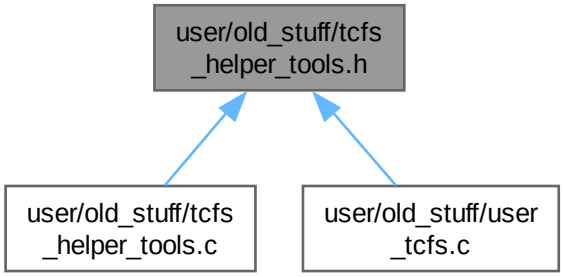
9.22 user/old_stuff/tcfs_helper_tools.h File Reference

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

Include dependency graph for tcfs_helper_tools.h:



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



Functions

- int do_mount ()

  Execute the mount of either a Network FS (for ex NFS), Local FS (for ex a block device), Local folder (a folder of the system)
- int setup_tcfs_env ()

9.22.1 Function Documentation

9.22.1.1 do_mount()

```
int do_mount ( )
```

Execute the mount of either a Network FS (for ex NFS), Local FS (for ex a block device), Local folder (a folder of the system)

Returns

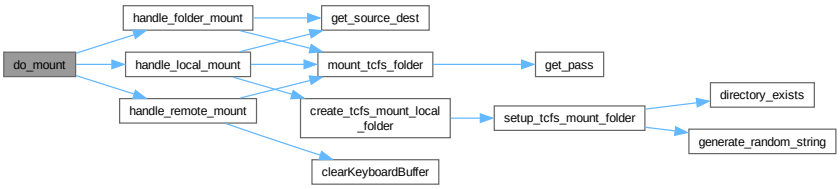
_ret

Definition at line 25 of file tcfs_helper_tools.c.

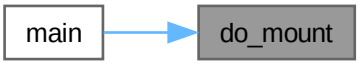
References handle_folder_mount(), handle_local_mount(), and handle_remote_mount().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



9.22.1.2 setup_tcfs_env()

```
int setup_tcfs_env ( )
```

Definition at line 405 of file tcfs_helper_tools.c.

Referenced by main().

Here is the caller graph for this function:

9.23 tcfs_helper_tools.h

Go to the documentation of this file.

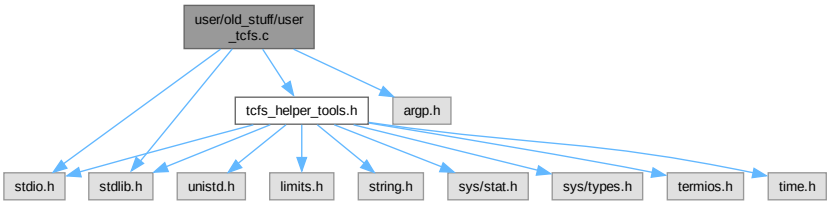
```
00001 #include <limits.h>
00002 #include <stdio.h>
00003 #include <stdlib.h>
00004 #include <string.h>
00005 #include <sys/stat.h>
00006 #include <sys/types.h>
00007 #include <termios.h>
00008 #include <time.h>
00009 #include <unistd.h>
00010
00011 int do_mount ();
00012 int setup_tcfs_env ();
```

9.24 user/old_stuff/user_tcfs.c File Reference

Help the user that wants to use TCFS.

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

Include dependency graph for user_tcfs.c:



Classes

- struct arguments

  Structure to hold the parsed arguments.

Functions

- static error_t parse_opt (int key, char ∗arg, struct argp_state ∗state)

  Parse the operation, used by argp. This function is marked as internal and should not be used by the user.
- int main (int argc, char ∗argv[ ])

  main function for the TCFS user helper program.

Variables

- const char * argp_program_version = "TCFS user helper program"

  Program version. This variable is marked as internal and should not be used by the user.
- const char * argp_program_bug_address = "carloalbertogiordano@duck.com"

  Mail for bug reports. This variable is marked as internal and should not be used by the user.
- static char doc [ ]

  Documentation for argp. This variable is marked as internal and should not be used by the user.
- static struct argp_option options [ ]

  Option accepted by tcfs helper program. This variable is marked as internal and should not be used by the user.
- static struct argp argp

9.24.1 Detailed Description

Help the user that wants to use TCFS.

Definition in file user_tcfs.c.

9.24.2 Function Documentation

9.24.2.1 main()

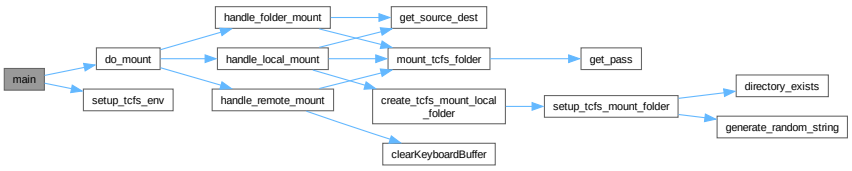
```
int main (
            int argc,
            char * argv[] )
```

main function for the TCFS user helper program.

Definition at line 104 of file user_tcfs.c.

References do_mount(), arguments::operation, and setup_tcfs_env().

Here is the call graph for this function:



9.24.2.2 parse_opt()

```
static error_t parse_opt (
            int key,
            char * arg,
            struct argp_state * state )  [static]
```

Parse the operation, used by argp. This function is marked as internal and should not be used by the user.

Parameters

key	The option character
arg	The argument string (unused)
state	The state object of argp

Returns

static error_t The error code (0 for success, ARGP_ERR_UNKNOWN for unknown option)

Definition at line 64 of file user_tcfs.c.

References arguments::operation.

9.24.3 Variable Documentation

9.24.3.1 argp

```
struct argp argp  [static]
```

Initial value:

```
= { .options = options,
                            .parser = parse_opt,
                            .doc = doc,
                            .args_doc = NULL,
                            .children = NULL,
                            .help_filter = NULL }
```

Definition at line 93 of file user_tcfs.c.

9.24.3.2 argp_program_bug_address

```
argp_program_bug_address = "carloalbertogiordano@duck.com"
```

Mail for bug reports. This variable is marked as internal and should not be used by the user.

Definition at line 22 of file user_tcfs.c.

9.24.3.3 argp_program_version

```
argp_program_version = "TCFS user helper program"
```

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

Definition at line 16 of file user_tcfs.c.

9.24.3.4 doc

```
doc  [static]
```

Initial value:
```
= "TCFS user accepts one of three arguments: mount, "
                    "create-shared, or umount."
```

Documentation for argp. This variable is marked as internal and should not be used by the user.

Definition at line 28 of file user_tcfs.c.

9.24.3.5 options

```
options  [static]
```

Initial value:
```
= { { "mount", 'm', 0, 0, "Perform mount operation", -1 },
        { "create-shared", 'c', 0, 0, "Perform create-shared operation", -1 },
        { "umount", 'u', 0, 0, "Perform umount operation", -1 },
        { "setup-env", 's', 0, 0, "Perform the setup of the .tcfs folder",
          -1 },
        { NULL } }
```

Option accepted by tcfs helper program. This variable is marked as internal and should not be used by the user.

Definition at line 36 of file user_tcfs.c.

9.25 user_tcfs.c

Go to the documentation of this file.

```
00001 #include "tcfs_helper_tools.h"
00002 #include <argp.h>
00003 #include <stdio.h>
00004 #include <stdlib.h>
00005
00016 const char *argp_program_version = "TCFS user helper program";
00022 const char *argp_program_bug_address = "carloalbertogiordano@duck.com";
00028 static char doc[] = "TCFS user accepts one of three arguments: mount, "
00029                     "create-shared, or umount.";
00030
00036 static struct argp_option options[]
00037     = { { "mount", 'm', 0, 0, "Perform mount operation", -1 },
00038         { "create-shared", 'c', 0, 0, "Perform create-shared operation", -1 },
00039         { "umount", 'u', 0, 0, "Perform umount operation", -1 },
00040         { "setup-env", 's', 0, 0, "Perform the setup of the .tcfs folder",
00041           -1 },
00042         { NULL } };
00043
00048 struct arguments
00049 {
00050   int operation;
00052 };
00053
00063 static error_t
00064 parse_opt (int key, char *arg, struct argp_state *state)
00065 {
00066   (void)arg;
00067
00068   struct arguments *arguments = state->input;
00069   switch (key)
00070     {
00071     case 'm':
00072       arguments->operation = 1; // Mount
00073       break;
00074     case 'c':
00075       arguments->operation = 2; // Create-shared
00076       break;
00077     case 'u':
```

```
00078       arguments->operation = 3; // Umount
00079       break;
00080     case 's':
00081       arguments->operation = 4; // Set up the env
00082       break;
00083     default:
00084       return ARGP_ERR_UNKNOWN;
00085     }
00086   return 0;
00087 }
00088
00093 static struct argp argp = { .options = options,
00094                             .parser = parse_opt,
00095                             .doc = doc,
00096                             .args_doc = NULL,
00097                             .children = NULL,
00098                             .help_filter = NULL };
00099
00103 int
00104 main (int argc, char *argv[])
00105 {
00106   struct arguments arguments;
00107   arguments.operation = 0; // Default value
00108   int result = 0;
00109
00110   // Parse the arguments
00111   argp_parse (&argp, argc, argv, 0, 0, &arguments);
00112
00113   switch (arguments.operation)
00114     {
00115     case 1:
00116       printf ("Mounting your FS, Please specify the location\n");
00117       result = do_mount ();
00118       if (result == 0)
00119         {
00120           fprintf (stderr, "An error occurred\n");
00121           exit (-1);
00122         }
00123       break;
00124     case 2:
00125       printf ("You chose the 'create-shared' operation.\n");
00126       // Add specific logic for 'create-shared' here.
00127       break;
00128     case 3:
00129       printf ("You chose the 'umount' operation.\n");
00130       // Add specific logic for 'umount' here.
00131       break;
00132     case 4:
00133       printf ("You chose the 'setup environment' option\n");
00134       result = setup_tcfs_env ();
00135       if (result == 0)
00136         {
00137           fprintf (stderr, "An error occurred\n");
00138           exit (-1);
00139         }
00140     default:
00141       printf ("Invalid argument. Choose from 'mount', 'create-shared', or "
00142               "'umount'.\n");
00143       return 1;
00144     }
00145
00146   return 0;
00147 }
```

9.26 user/command_handler/__init__.py File Reference

9.27 __init__.py

Go to the documentation of this file.

9.28 user/ui/__init__.py File Reference

9.29 __init__.py

Go to the documentation of this file.

9.30 user/ui/main_window.py File Reference

Classes

- class ui.main_window.Window

Namespaces

- namespace ui
- namespace ui.main_window

Functions

- ui.main_window.modify_button_allign (button, row, column)

9.31 main_window.py

Go to the documentation of this file.

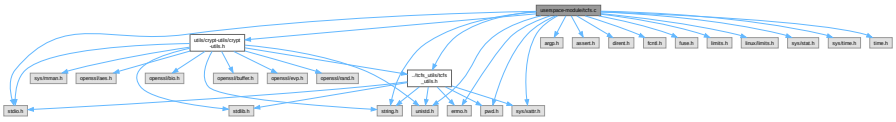
```
00001 import tkinter as tk
00002
00003
00004 def modify_button_allign(button, row, column):
00005     button.grid(row=row, column=column)
00006
00007
00008 class Window:
00009
00010     def __init__(self, title: str, geometry: str):
00011         self.window = tk.Tk()
00012         self.window.title(title)
00013         self.window.geometry(geometry)
00014
00015     def add_button(self, text: str, function, row=0, col=0):
00016         button = tk.Button(self.window, text=text, command=function)
00017         button.grid(row=row, column=col)
00018         return button
00019
00020     def start_window(self):
00021         self.window.mainloop()
00022
```

9.32 userspace-module/tcfs.c File Reference

```
#include "utils/crypt-utils/crypt-utils.h"
#include "utils/tcfs_utils/tcfs_utils.h"
#include <argp.h>
#include <assert.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <fuse.h>
#include <limits.h>
#include <linux/limits.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
```

```
#include <sys/time.h>
#include <sys/xattr.h>
#include <time.h>
#include <unistd.h>
```

Include dependency graph for tcfs.c:



Classes

- struct arguments

  Structure to hold the parsed arguments.

Macros

- #define FUSE_USE_VERSION 30
- #define HAVE_SETXATTR
- #define _XOPEN_SOURCE 500

Functions

- static int tcfs_getxattr (const char *fuse_path, const char *name, char *value, size_t size)
- static int tcfs_opendir (const char *fuse_path, struct fuse_file_info *fi)
- static int tcfs_getattr (const char *fuse_path, struct stat *stbuf)
- static int tcfs_access (const char *fuse_path, int mask)
- static int tcfs_readlink (const char *fuse_path, char *buf, size_t size)
- static int tcfs_readdir (const char *fuse_path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi)
- static int tcfs_mknod (const char *fuse_path, mode_t mode, dev_t rdev)
- static int tcfs_mkdir (const char *fuse_path, mode_t mode)
- static int tcfs_unlink (const char *fuse_path)
- static int tcfs_rmdir (const char *fuse_path)
- static int tcfs_symlink (const char *from, const char *to)
- static int tcfs_rename (const char *from, const char *to)
- static int tcfs_link (const char *from, const char *to)
- static int tcfs_chmod (const char *fuse_path, mode_t mode)
- static int tcfs_chown (const char *fuse_path, uid_t uid, gid_t gid)
- static int tcfs_truncate (const char *fuse_path, off_t size)
- static int tcfs_utimens (const char *fuse_path, const struct timespec ts[2])
- static int tcfs_open (const char *fuse_path, struct fuse_file_info *fi)
- static int file_size (FILE *file)
- static int tcfs_read (const char *fuse_path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
- static int tcfs_write (const char *fuse_path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
- static int tcfs_statfs (const char *fuse_path, struct statvfs *stbuf)
- static int tcfs_setxattr (const char *fuse_path, const char *name, const char *value, size_t size, int flags)
- static int tcfs_create (const char *fuse_path, mode_t mode, struct fuse_file_info *fi)
- static int tcfs_release (const char *fuse_path, struct fuse_file_info *fi)
- static int tcfs_fsync (const char *fuse_path, int isdatasync, struct fuse_file_info *fi)
- static int tcfs_listxattr (const char *fuse_path, char *list, size_t size)
- static int tcfs_removexattr (const char *fuse_path, const char *name)
- static error_t parse_opt (int key, char *arg, struct argp_state *state)
- int main (int argc, char *argv[])

Variables

- char * root_path
  Contains the fullpath to the mounted directory.
- char * password
  Contains the password passed to TCFS when started.
- static struct fuse_operations tcfs_oper
- const char * argp_program_version = "TCFS Alpha"
- const char * argp_program_bug_address = "carloalbertogiordano@duck.com"
- static char doc [ ]
- static char args_doc [ ] = ""
- static struct argp_option options [ ]
- static struct argp argp = { options, parse_opt, args_doc, doc, 0, NULL, NULL }

9.32.1 Macro Definition Documentation

9.32.1.1 _XOPEN_SOURCE

```
#define _XOPEN_SOURCE 500
```

Definition at line 12 of file tcfs.c.

9.32.1.2 FUSE_USE_VERSION

```
#define FUSE_USE_VERSION 30
```

Definition at line 1 of file tcfs.c.

9.32.1.3 HAVE_SETXATTR

```
#define HAVE_SETXATTR
```

Definition at line 2 of file tcfs.c.

9.32.2 Function Documentation

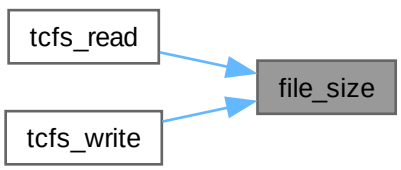
9.32.2.1 file_size()

```
static int file_size (
            FILE * file ) [inline], [static]
```

Definition at line 346 of file tcfs.c.

Referenced by tcfs_read(), and tcfs_write().

Here is the caller graph for this function:



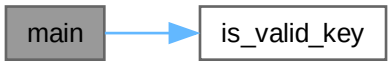
9.32.2.2 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 768 of file tcfs.c.

References arguments::destination, is_valid_key(), password, arguments::password, root_path, arguments::source, and tcfs_oper.

Here is the call graph for this function:



9.32.2.3 parse_opt()

```
static error_t parse_opt (
            int key,
            char * arg,
            struct argp_state * state )  [static]
```

Definition at line 741 of file tcfs.c.

References arguments::destination, arguments::password, and arguments::source.

9.32.2.4 tcfs_access()

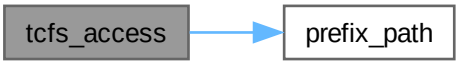
```
static int tcfs_access (
            const char * fuse_path,
            int mask )  [static]
```

Definition at line 89 of file tcfs.c.

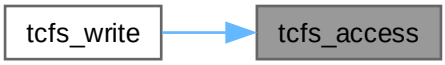
References prefix_path(), and root_path.

Referenced by tcfs_write().

Here is the call graph for this function:



Here is the caller graph for this function:



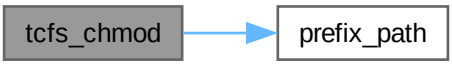
9.32.2.5 tcfs_chmod()

```
static int tcfs_chmod (
            const char * fuse_path,
            mode_t mode )  [static]
```

Definition at line 263 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



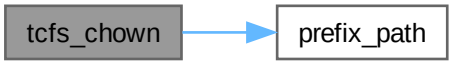
9.32.2.6 tcfs_chown()

```
static int tcfs_chown (
            const char * fuse_path,
            uid_t uid,
            gid_t gid )  [static]
```

Definition at line 278 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



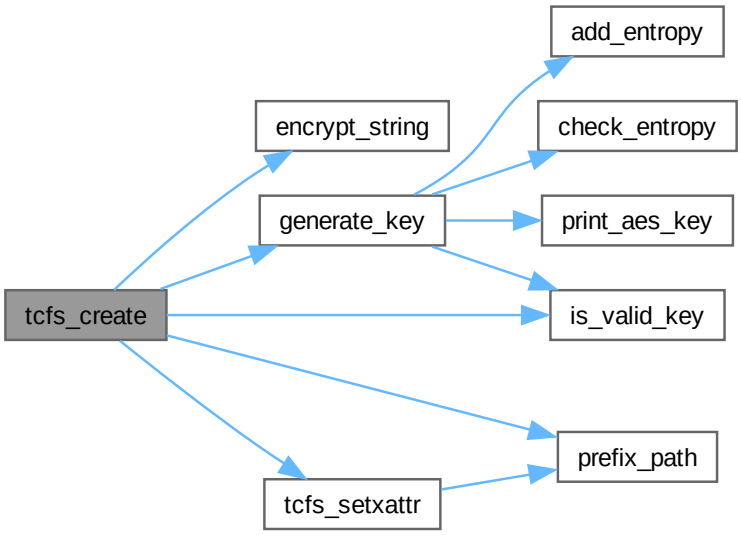
9.32.2.7 tcfs_create()

```
static int tcfs_create (
            const char * fuse_path,
            mode_t mode,
            struct fuse_file_info * fi )  [static]
```

Definition at line 553 of file tcfs.c.

References encrypt_string(), generate_key(), is_valid_key(), password, prefix_path(), root_path, and tcfs_setxattr().

Here is the call graph for this function:



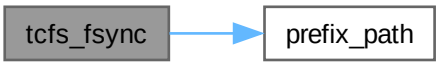
9.32.2.8 tcfs_fsync()

```
static int tcfs_fsync (
            const char * fuse_path,
            int isdatasync,
            struct fuse_file_info * fi )  [static]
```

Definition at line 632 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



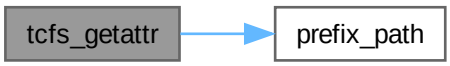
9.32.2.9 tcfs_getattr()

```
static int tcfs_getattr (
            const char * fuse_path,
            struct stat * stbuf )  [static]
```

Definition at line 74 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.10 tcfs_getxattr()

```
static int tcfs_getxattr (
            const char * fuse_path,
            const char * name,
            char * value,
            size_t size )  [static]
```

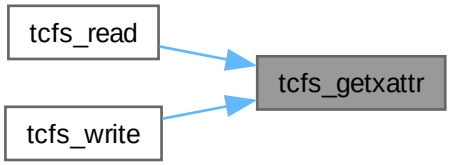
Definition at line 645 of file tcfs.c.

References prefix_path(), and root_path.

Referenced by tcfs_read(), and tcfs_write().

Here is the call graph for this function:

Here is the caller graph for this function:



9.32.2.11 tcfs_link()

```
static int tcfs_link (
            const char * from,
            const char * to )  [static]
```

Definition at line 250 of file tcfs.c.

9.32.2.12 tcfs_listxattr()

```
static int tcfs_listxattr (
            const char * fuse_path,
            char * list,
            size_t size )  [static]
```

Definition at line 666 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:

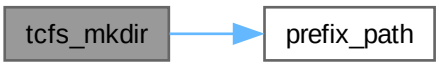
9.32.2.13 tcfs_mkdir()

```
static int tcfs_mkdir (
            const char * fuse_path,
            mode_t mode )  [static]
```

Definition at line 179 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



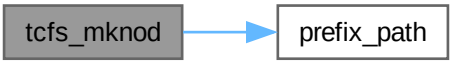
9.32.2.14 tcfs_mknod()

```
static int tcfs_mknod (
            const char * fuse_path,
            mode_t mode,
            dev_t rdev )  [static]
```

Definition at line 153 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



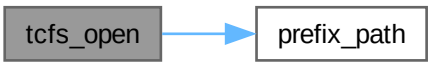
9.32.2.15 tcfs_open()

```
static int tcfs_open (
            const char * fuse_path,
            struct fuse_file_info * fi )  [static]
```

Definition at line 331 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.16 tcfs_opendir()

```
static int tcfs_opendir (
            const char * fuse_path,
            struct fuse_file_info * fi )  [static]
```

Todo Implement the opendir function

Definition at line 52 of file tcfs.c.

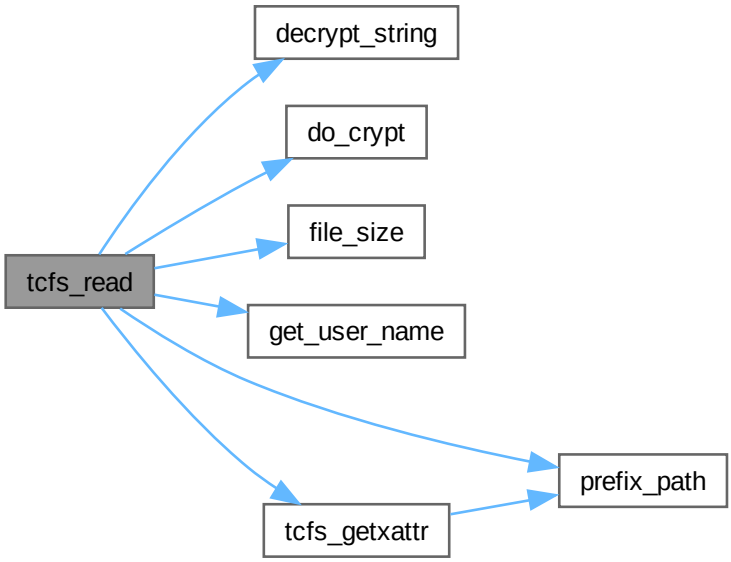
9.32.2.17 tcfs_read()

```
static int tcfs_read (
            const char * fuse_path,
            char * buf,
            size_t size,
            off_t offset,
            struct fuse_file_info * fi )  [static]
```

Definition at line 357 of file tcfs.c.

References DECRYPT, decrypt_string(), do_crypt(), file_size(), get_user_name(), password, prefix_path(), root_path, and tcfs_getxattr().

Here is the call graph for this function:



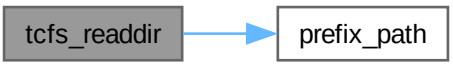
9.32.2.18 tcfs_readdir()

```
static int tcfs_readdir (
            const char * fuse_path,
            void * buf,
            fuse_fill_dir_t filler,
            off_t offset,
            struct fuse_file_info * fi )  [static]
```

Definition at line 119 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.19 tcfs_readlink()

```
static int tcfs_readlink (
            const char * fuse_path,
            char * buf,
            size_t size )  [static]
```

Definition at line 104 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:

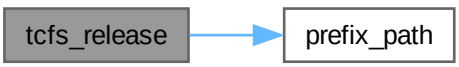
9.32.2.20 tcfs_release()

```
static int tcfs_release (
            const char * fuse_path,
            struct fuse_file_info * fi )  [static]
```

Definition at line 620 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.21 tcfs_removexattr()

```
static int tcfs_removexattr (
            const char * fuse_path,
            const char * name )  [static]
```

Definition at line 678 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:

9.32.2.22 tcfs_rename()

```
static int tcfs_rename (
            const char * from,
            const char * to )  [static]
```

Definition at line 237 of file tcfs.c.

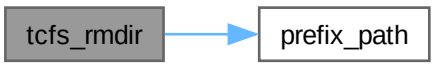
9.32.2.23 tcfs_rmdir()

```
static int tcfs_rmdir (
            const char * fuse_path )  [static]
```

Definition at line 209 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.24 tcfs_setxattr()

```
static int tcfs_setxattr (
            const char * fuse_path,
            const char * name,
            const char * value,
            size_t size,
            int flags )  [static]
```

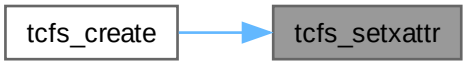
Definition at line 540 of file tcfs.c.

References prefix_path(), and root_path.

Referenced by tcfs_create().

Here is the call graph for this function:

Here is the caller graph for this function:



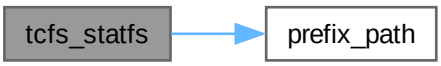
9.32.2.25 tcfs_statfs()

```
static int tcfs_statfs (
            const char * fuse_path,
            struct statvfs * stbuf )  [static]
```

Definition at line 525 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.26 tcfs_symlink()

```
static int tcfs_symlink (
            const char * from,
            const char * to )  [static]
```

Definition at line 224 of file tcfs.c.

9.32.2.27 tcfs_truncate()

```
static int tcfs_truncate (
            const char * fuse_path,
            off_t size )  [static]
```

Definition at line 293 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:

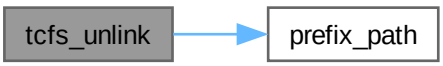
9.32.2.28 tcfs_unlink()

```
static int tcfs_unlink (
            const char * fuse_path )  [static]
```

Definition at line 194 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:



9.32.2.29 tcfs_utimens()

```
static int tcfs_utimens (
            const char * fuse_path,
            const struct timespec ts[2] )  [static]
```

Definition at line 309 of file tcfs.c.

References prefix_path(), and root_path.

Here is the call graph for this function:

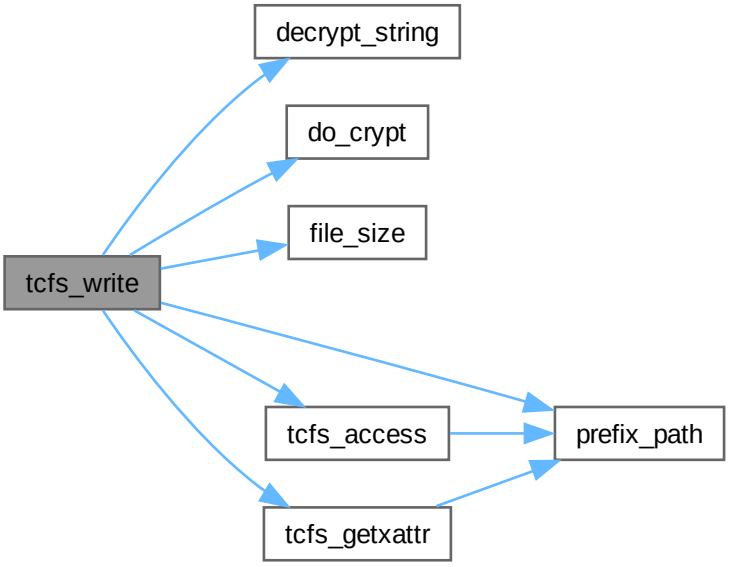
9.32.2.30 tcfs_write()

```
static int tcfs_write (
            const char * fuse_path,
            const char * buf,
            size_t size,
            off_t offset,
            struct fuse_file_info * fi )  [static]
```

Definition at line 439 of file tcfs.c.

References DECRYPT, decrypt_string(), do_crypt(), ENCRYPT, file_size(), password, prefix_path(), root_path, tcfs_access(), and tcfs_getxattr().

Here is the call graph for this function:



9.32.3 Variable Documentation

9.32.3.1 argp

```
struct argp argp = { options, parse_opt, args_doc, doc, 0, NULL, NULL }  [static]
```

Definition at line 765 of file tcfs.c.

9.32.3.2 argp_program_bug_address

```
const char* argp_program_bug_address = "carloalbertogiordano@duck.com"
```

Definition at line 720 of file tcfs.c.

9.32.3.3 argp_program_version

```
const char* argp_program_version = "TCFS Alpha"
```

Definition at line 719 of file tcfs.c.

9.32.3.4 args_doc

```
char args_doc[] = ""  [static]
```

Definition at line 725 of file tcfs.c.

9.32.3.5 doc

```
char doc[]  [static]
```

Initial value:
```
= "This is an implementation on TCFS\ntcfs -s <source_path> "
                    "-d <dest_path> -p <password> [fuse arguments]"
```

Definition at line 722 of file tcfs.c.

9.32.3.6 options

```
struct argp_option options[]  [static]
```

Initial value:
```
= { { "source", 's', "SOURCE", 0, "Source file path", -1 },
        { "destination", 'd', "DESTINATION", 0, "Destination file path", -1 },
        { "password", 'p', "PASSWORD", 0, "Password", -1 },
        { NULL } }
```

Definition at line 727 of file tcfs.c.

9.32.3.7 password

```
password
```

Contains the password passed to TCFS when started.

Definition at line 43 of file tcfs.c.

Referenced by main(), tcfs_create(), tcfs_read(), and tcfs_write().

9.32.3.8 root_path

```
root_path
```

Contains the fullpath to the mounted directory.

Definition at line 38 of file tcfs.c.

Referenced by main(), tcfs_access(), tcfs_chmod(), tcfs_chown(), tcfs_create(), tcfs_fsync(), tcfs_getattr(), tcfs_getxattr(), tcfs_listxattr(), tcfs_mkdir(), tcfs_mknod(), tcfs_open(), tcfs_read(), tcfs_readdir(), tcfs_readlink(), tcfs_release(), tcfs_removexattr(), tcfs_rmdir(), tcfs_setxattr(), tcfs_statfs(), tcfs_truncate(), tcfs_unlink(), tcfs_utimens(), and tcfs_write().

9.32.3.9 tcfs_oper

```
struct fuse_operations tcfs_oper  [static]
```

Initial value:

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

Definition at line 689 of file tcfs.c.

Referenced by main().

9.33 tcfs.c

Go to the documentation of this file.

```
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     {
```

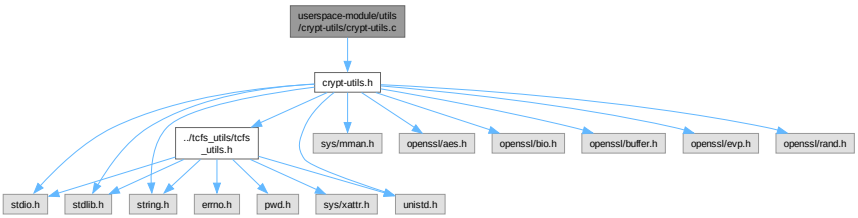
```
00568       fclose (res);
00569       return -errno;
00570     }
00571
00572   // Generate and set a new encrypted key for the file
00573   unsigned char *key = malloc (sizeof (unsigned char) * 33);
00574   key[32] = '\0';
00575   generate_key (key);
00576
00577   if (key == NULL)
00578     {
00579       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 }
```

9.34 userspace-module/utils/crypt-utils/crypt-utils.c File Reference

`#include "crypt-utils.h"`
Include dependency graph for crypt-utils.c:



Macros

- #define BLOCKSIZE 1024

  This defines the max size of a block that can be cyphered. This definition is marked as internal and should not be used directly by the user.
- #define IV_SIZE 32

  The fixed size of the initialization vector IV . This definition is marked as internal and should not be used directly by the user.
- #define KEY_SIZE 32

  The fixed size of the key. This definition is marked as internal and should not be used directly by the user.

Functions

- int do_crypt (FILE ∗in, FILE ∗out, int action, unsigned char ∗key_str)

  High level function interface for performing AES encryption on FILE pointers Uses OpenSSL libcrypto EVP API .
- int check_entropy (void)

  Verify if there is enough entropy in the system to generate a key.
- void add_entropy (void)

  Force new entropy in /dev/urandom, This function is marked as internal and should not be used by the user.
- void generate_key (unsigned char ∗destination)

  Generate a new AES 256 key for a file.
- unsigned char ∗ encrypt_string (unsigned char ∗plaintext, const char ∗key, int ∗encrypted_key_len)

  Encrypt the ∗plaintext string using a AES 256 key.
- unsigned char ∗ decrypt_string (unsigned char ∗ciphertext, const char ∗key)

  Decrypt the ∗ciphertext string using a AES 256 key.
- int is_valid_key (const unsigned char ∗key)

  Check if a given key is valid.

9.34.1 Macro Definition Documentation

9.34.1.1 BLOCKSIZE

`#define BLOCKSIZE 1024`

This defines the max size of a block that can be cyphered. This definition is marked as internal and should not be used directly by the user.

Definition at line 12 of file crypt-utils.c.

9.34.1.2 IV_SIZE

```
#define IV_SIZE 32
```

The fixed size of the initialization vector IV . This definition is marked as internal and should not be used directly by the user.

Definition at line 19 of file crypt-utils.c.

9.34.1.3 KEY_SIZE

```
#define KEY_SIZE 32
```

The fixed size of the key. This definition is marked as internal and should not be used directly by the user.

Definition at line 25 of file crypt-utils.c.

9.34.2 Function Documentation

9.34.2.1 add_entropy()

```
void add_entropy (
            void  )
```

Force new entropy in /dev/urandom, This function is marked as internal and should not be used by the user.

Parameters

void	

Returns

void

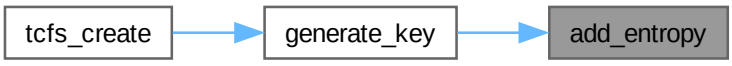
Note

Very dangerous, if this fails an error will be printed and the program will exit with EXIT_FAILURE

Definition at line 199 of file crypt-utils.c.

Referenced by generate_key().

Here is the caller graph for this function:



9.34.2.2 check_entropy()

```
int check_entropy (
            void  )
```

Verify if there is enough entropy in the system to generate a key.

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

Parameters

void	

Returns

A value greater than 0 corresponding to the entropy level, if an error occurs -1 is returned

Note

This function evaluates the entropy by checking the /proc/sys/kernel/random/entropy_avail file.

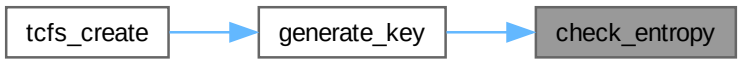
See also

man page 4 for random

Definition at line 169 of file crypt-utils.c.

Referenced by generate_key().

Here is the caller graph for this function:



9.34.2.3 decrypt_string()

```
unsigned char * decrypt_string (
            unsigned char * ciphertext,
            const char * key )
```

Decrypt the *ciphertext string using a AES 256 key.

Parameters

ciphertext	This is the string to decrypt
key	The AES 256 KEY

Returns

unsigned char * The plaintext string will be allocated and then returned

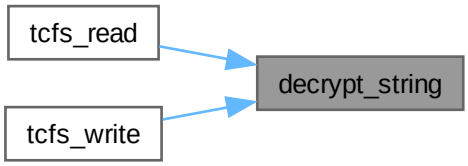
Note

After the use remember to free the result

Definition at line 325 of file crypt-utils.c.

Referenced by tcfs_read(), and tcfs_write().

Here is the caller graph for this function:



9.34.2.4 do_crypt()

```
int do_crypt (
            FILE * in,
            FILE * out,
            int action,
            unsigned char * key_str )  [extern]
```

High level function interface for performing AES encryption on FILE pointers Uses OpenSSL libcrypto EVP API
.

Author

By Andy Sayler (www.andysayler.com)
Created 04/17/12

Modified 18/10/23 by [Carlo Alberto Giordano]

Derived from OpenSSL.org EVP_Encrypt_* Manpage Examples
http://www.openssl.org/docs/crypto/EVP_EncryptInit.html#EXAMPLES
With additional information from Saju Pillai's OpenSSL AES Example
http://saju.net.in/blog/?p=36
http://saju.net.in/code/misc/openssl_aes.c.txt

Parameters

in	The input file
out	The output file
action	Defines if the action to do on the input file should be of encryption or decryption.

See also

ENCRYPT

DECRYPT

Parameters

key_str	The key that must be AES 256

Returns

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

See also

print_err

Note

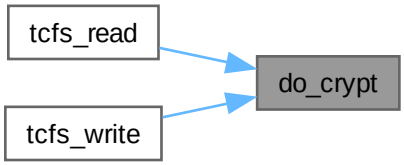
This function cyphers using AES 256 CBC

Definition at line 54 of file crypt-utils.c.

References BLOCKSIZE, IV_SIZE, and KEY_SIZE.

Referenced by tcfs_read(), and tcfs_write().

Here is the caller graph for this function:



9.34.2.5 encrypt_string()

```
unsigned char * encrypt_string (
            unsigned char * plaintext,
            const char * key,
            int * encrypted_key_len )
```

Encrypt the ∗plaintext string using a AES 256 key.

Parameters

plaintext	This is the string to encrypt
key	The AES 256 KEY
encrypted_len	This will be set to the encrypted string length

Returns

unsigned char * The encrypted string will be allocated and then returned

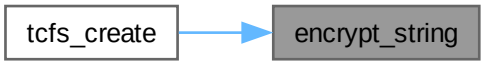
Note

After the use remember to free the result

Definition at line 275 of file crypt-utils.c.

Referenced by tcfs_create().

Here is the caller graph for this function:



9.34.2.6 generate_key()

```
void generate_key (
            unsigned char * destination )
```

Generate a new AES 256 key for a file.

Parameters

destination	Pointer to the string in which the generated key will be saved. If an error occurs it will be set to NULL

Returns

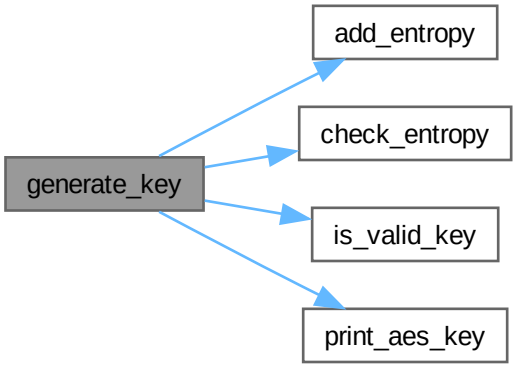
void

Definition at line 232 of file crypt-utils.c.

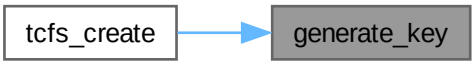
References add_entropy(), check_entropy(), is_valid_key(), and print_aes_key().

Referenced by tcfs_create().

Here is the call graph for this function:



Here is the caller graph for this function:



9.34.2.7 is_valid_key()

```
int is_valid_key (
            const unsigned char * key )
```

Check if a given key is valid.

Parameters

key	The key to validate

Returns

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

See also

print_err

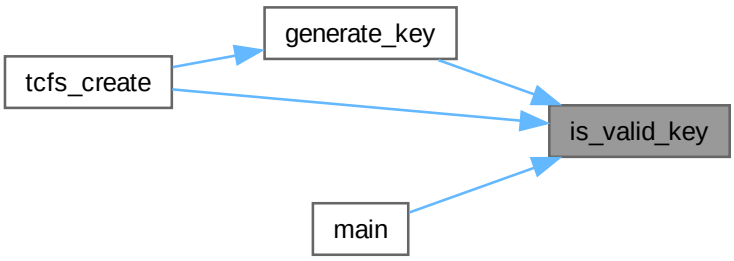
Note

This function only checks for key length

Definition at line 360 of file crypt-utils.c.

Referenced by generate_key(), main(), and tcfs_create().

Here is the caller graph for this function:



9.35 crypt-utils.c

Go to the documentation of this file.

```
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
```

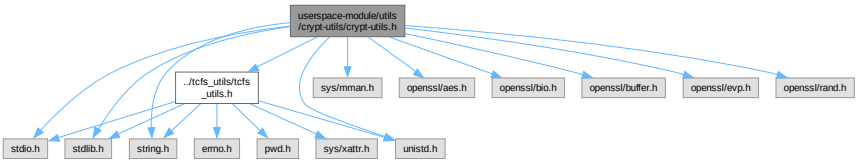
```
00289   EVP_EncryptInit_ex (ctx, cipher, NULL, (const unsigned char *)key, iv);
00290
00291   size_t plaintext_len = strlen ((const char *)plaintext);
00292   unsigned char ciphertext[plaintext_len + AES_BLOCK_SIZE];
00293   memset (ciphertext, 0, sizeof (ciphertext));
00294
00295   int len;
00296   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 }*/
```

9.36 userspace-module/utils/crypt-utils/crypt-utils.h File Reference

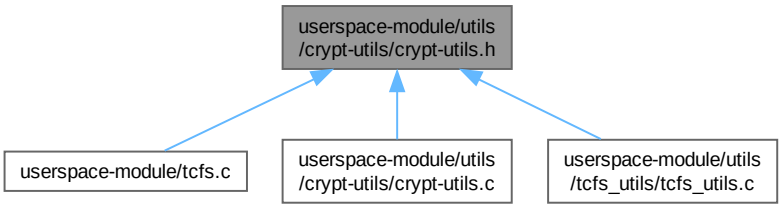
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <openssl/aes.h>
```

```
#include <openssl/bio.h>
#include <openssl/buffer.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include "../tcfs_utils/tcfs_utils.h"
```

Include dependency graph for crypt-utils.h:



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



Macros

- #define BLOCKSIZE 1024
- #define ENCRYPT 1

  Signifies that the selected action is encryption.
- #define DECRYPT 0

  Signifies that the selected action is decryption.

Functions

- int do_crypt (FILE ∗in, FILE ∗out, int action, unsigned char ∗key_str)

  High level function interface for performing AES encryption on FILE pointers Uses OpenSSL libcrypto EVP API .
- void generate_key (unsigned char ∗destination)

  Generate a new AES 256 key for a file.
- unsigned char ∗ encrypt_string (unsigned char ∗plaintext, const char ∗key, int ∗encrypted_len)

  Encrypt the ∗plaintext string using a AES 256 key.
- unsigned char ∗ decrypt_string (unsigned char ∗base64_ciphertext, const char ∗key)

  Decrypt the ∗ciphertext string using a AES 256 key.
- int is_valid_key (const unsigned char ∗key)

  Check if a given key is valid.

9.36.1 Macro Definition Documentation

9.36.1.1 BLOCKSIZE

```
#define BLOCKSIZE 1024
```

Definition at line 15 of file crypt-utils.h.

9.36.1.2 DECRYPT

```
#define DECRYPT 0
```

Signifies that the selected action is decryption.

Definition at line 25 of file crypt-utils.h.

9.36.1.3 ENCRYPT

```
#define ENCRYPT 1
```

Signifies that the selected action is encryption.

Definition at line 20 of file crypt-utils.h.

9.36.2 Function Documentation

9.36.2.1 decrypt_string()

```
unsigned char * decrypt_string (
            unsigned char * ciphertext,
            const char * key )
```

Decrypt the ∗ciphertext string using a AES 256 key.

Parameters

ciphertext	This is the string to decrypt
key	The AES 256 KEY

Returns

unsigned char ∗ The plaintext string will be allocated and then returned

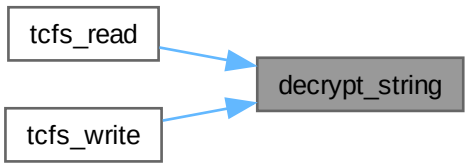
Note

After the use remember to free the result

Definition at line 325 of file crypt-utils.c.

Referenced by tcfs_read(), and tcfs_write().

Here is the caller graph for this function:



9.36.2.2 do_crypt()

```
int do_crypt (
              FILE * in,
              FILE * out,
              int action,
              unsigned char * key_str )  [extern]
```

High level function interface for performing AES encryption on FILE pointers Uses OpenSSL libcrypto EVP API
.

Author

By Andy Sayler (www.andysayler.com)
Created 04/17/12

Modified 18/10/23 by [Carlo Alberto Giordano]

Derived from OpenSSL.org EVP_Encrypt_∗ Manpage Examples
http://www.openssl.org/docs/crypto/EVP_EncryptInit.html#EXAMPLES
With additional information from Saju Pillai's OpenSSL AES Example
http://saju.net.in/blog/?p=36
http://saju.net.in/code/misc/openssl_aes.c.txt

Parameters

in	The input file
out	The output file
action	Defines if the action to do on the input file should be of encryption or decryption.

See also

ENCRYPT

DECRYPT

Parameters

key_str	The key that must be AES 256

Returns

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

See also

print_err

Note

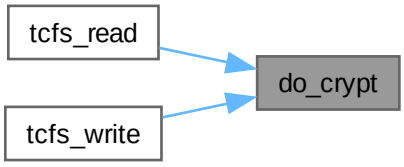
This function cyphers using AES 256 CBC

Definition at line 54 of file crypt-utils.c.

References BLOCKSIZE, IV_SIZE, and KEY_SIZE.

Referenced by tcfs_read(), and tcfs_write().

Here is the caller graph for this function:



9.36.2.3 encrypt_string()

```
unsigned char * encrypt_string (
            unsigned char * plaintext,
            const char * key,
            int * encrypted_key_len )
```

Encrypt the ∗plaintext string using a AES 256 key.

Parameters

plaintext	This is the string to encrypt
key	The AES 256 KEY
encrypted_len	This will be set to the encrypted string length

Returns

unsigned char * The encrypted string will be allocated and then returned

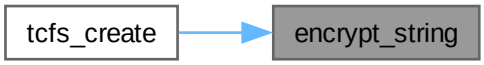
Note

After the use remember to free the result

Definition at line 275 of file crypt-utils.c.

Referenced by tcfs_create().

Here is the caller graph for this function:



9.36.2.4 generate_key()

```
void generate_key (
            unsigned char * destination )
```

Generate a new AES 256 key for a file.

Parameters

destination	Pointer to the string in which the generated key will be saved. If an error occurs it will be set to NULL

Returns

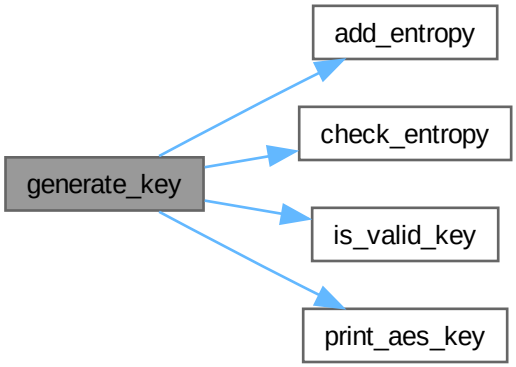
void

Definition at line 232 of file crypt-utils.c.

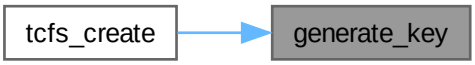
References add_entropy(), check_entropy(), is_valid_key(), and print_aes_key().

Referenced by tcfs_create().

Here is the call graph for this function:



Here is the caller graph for this function:



9.36.2.5 is_valid_key()

```
int is_valid_key (
            const unsigned char * key )
```

Check if a given key is valid.

Parameters

key	The key to validate

Returns

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

See also

print_err

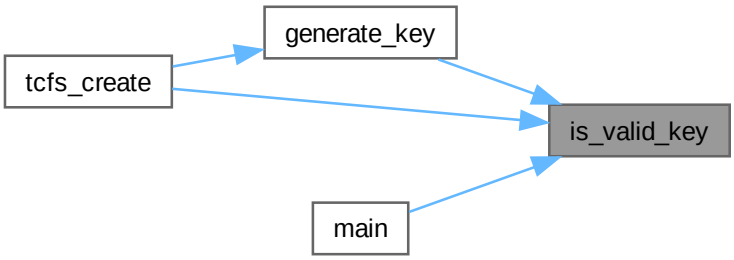
Note

This function only checks for key length

Definition at line 360 of file crypt-utils.c.

Referenced by generate_key(), main(), and tcfs_create().

Here is the caller graph for this function:



9.37 crypt-utils.h

Go to the documentation of this file.

```
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 */
```

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

This file will handle key exchanges with the kernel module.

9.38.1 Detailed Description

This file will handle key exchanges with the kernel module.

This is not being currently developed

Definition in file password_manager.c.

9.39 password_manager.c

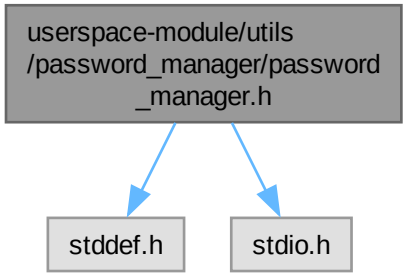
Go to the documentation of this file.

```
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 }*/
```

9.40 userspace-module/utils/password_manager/password_manager.h File Reference

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

Include dependency graph for password_manager.h:



Macros

- #define WITH_SYS_CALL 1

  the system aims to be independent from the kernel module. The kernel module is not beeing developed so this is useless. This definition is marked as internal and should not be used directly by the user
- #define WITHOUT_SYS_CALL 0

  the system aims to be independent from the kernel module. The kernel module is not beeing developed so this is useless. This definition is marked as internal and should not be used directly by the user

9.40.1 Macro Definition Documentation

9.40.1.1 WITH_SYS_CALL

```
#define WITH_SYS_CALL 1
```

the system aims to be independent from the kernel module. The kernel module is not beeing developed so this is useless. This definition is marked as internal and should not be used directly by the user

Definition at line 10 of file password_manager.h.

9.40.1.2 WITHOUT_SYS_CALL

```
#define WITHOUT_SYS_CALL 0
```

the system aims to be independent from the kernel module. The kernel module is not beeing developed so this is useless. This definition is marked as internal and should not be used directly by the user

Definition at line 17 of file password_manager.h.

9.41 password_manager.h

Go to the documentation of this file.

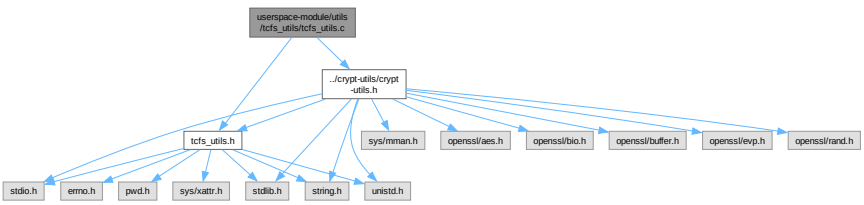
```
00001 #include <stddef.h>
00002 #include <stdio.h>
00003
00010 #define WITH_SYS_CALL 1
00017 #define WITHOUT_SYS_CALL 0
00018 /*
00019 int insert_key(char* key, char* cert, int is_sys_call);
00020 char *request_key(int is_sys_call);
00021 int delete_key(int is_sys_call);*/
```

9.42 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.

9.42.1 Detailed Description

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

See also

tcfs.c

Definition in file tcfs_utils.c.

9.42.2 Function Documentation

9.42.2.1 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:

9.42.2.2 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

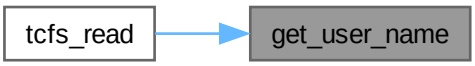
Note

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

Definition at line 17 of file tcfs_utils.c.

Referenced by tcfs_read().

Here is the caller graph for this function:



9.42.2.3 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:

9.42.2.4 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

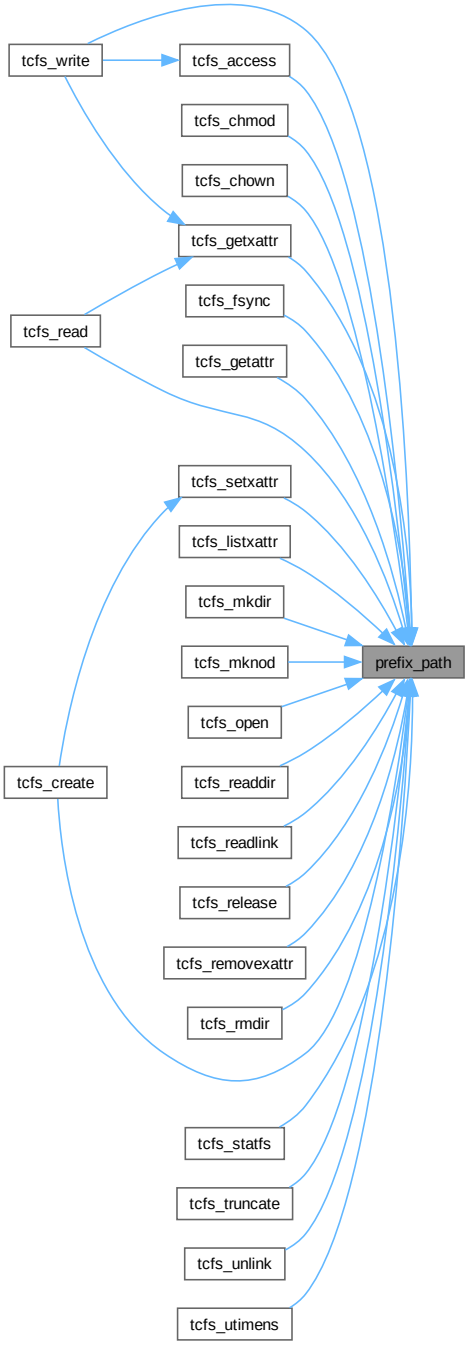
Note

Please free the result after use

Definition at line 57 of file tcfs_utils.c.

Referenced by tcfs_access(), tcfs_chmod(), tcfs_chown(), tcfs_create(), tcfs_fsync(), tcfs_getattr(), tcfs_getxattr(), tcfs_listxattr(), tcfs_mkdir(), tcfs_mknod(), tcfs_open(), tcfs_read(), tcfs_readdir(), tcfs_readlink(), tcfs_release(), tcfs_removexattr(), tcfs_rmdir(), tcfs_setxattr(), tcfs_statfs(), tcfs_truncate(), tcfs_unlink(), tcfs_utimens(), and tcfs_write().

Here is the caller graph for this function:



9.42.2.5 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

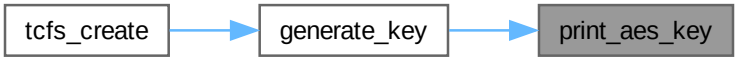
Returns

void

Definition at line 170 of file tcfs_utils.c.

Referenced by generate_key().

Here is the caller graph for this function:



9.42.2.6 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.

9.43 tcfs_utils.c

Go to the documentation of this file.

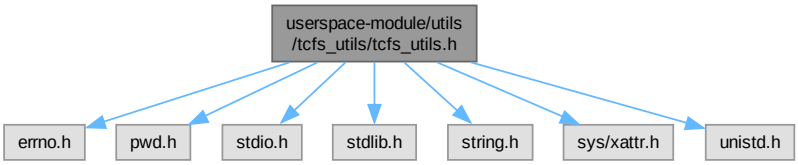
```
00001 #include "tcfs_utils.h"
00002 #include "../crypt-utils/crypt-utils.h"
00003
00016 void
00017 get_user_name (char *buf, size_t size)
00018 {
00019   uid_t uid = geteuid ();
00020   struct passwd *pw = getpwuid (uid);
00021   if (pw)
00022     snprintf (buf, size, "%s", pw->pw_name);
00023   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 }
```

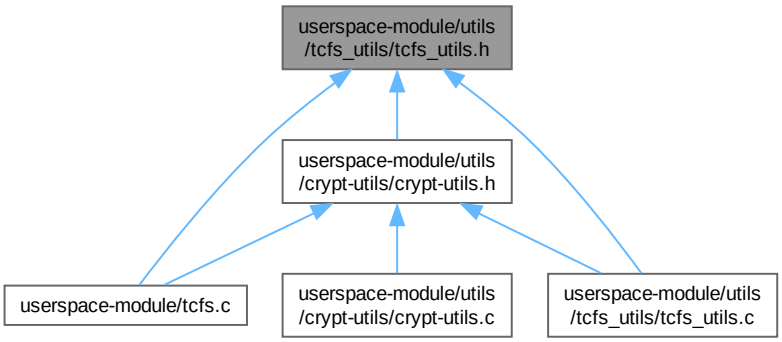
9.44 userspace-module/utils/tcfs_utils/tcfs_utils.h File Reference

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

Include dependency graph for tcfs_utils.h:



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



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.

9.44.1 Function Documentation

9.44.1.1 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:

9.44.1.2 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

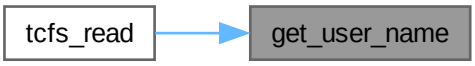
Note

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

Definition at line 17 of file tcfs_utils.c.

Referenced by tcfs_read().

Here is the caller graph for this function:



9.44.1.3 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:

9.44.1.4 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

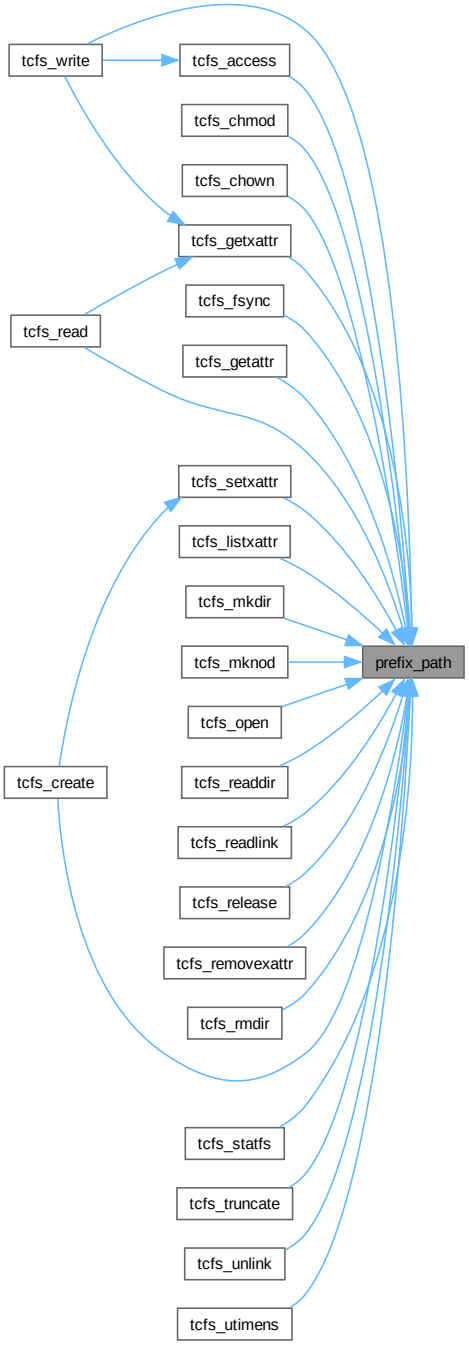
Note

Please free the result after use

Definition at line 57 of file tcfs_utils.c.

Referenced by tcfs_access(), tcfs_chmod(), tcfs_chown(), tcfs_create(), tcfs_fsync(), tcfs_getattr(), tcfs_getxattr(), tcfs_listxattr(), tcfs_mkdir(), tcfs_mknod(), tcfs_open(), tcfs_read(), tcfs_readdir(), tcfs_readlink(), tcfs_release(), tcfs_removexattr(), tcfs_rmdir(), tcfs_setxattr(), tcfs_statfs(), tcfs_truncate(), tcfs_unlink(), tcfs_utimens(), and tcfs_write().

Here is the caller graph for this function:



9.44.1.5 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

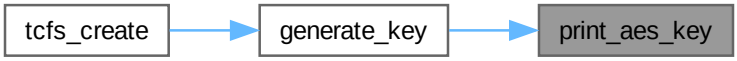
Returns

void

Definition at line 170 of file tcfs_utils.c.

Referenced by generate_key().

Here is the caller graph for this function:



9.44.1.6 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.

9.45 tcfs_utils.h

Go to the documentation of this file.

```
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