# Role-Based Access Control Filesystem (RBACFS)

Scott Linder Ryan DePrekel Justin Lanyon

November 18, 2016

**Abstract**

This document describes the purpose, implementation, and maintenance of the Role-Based Access Control Filesystem (RBACFS) software. Any assumed technical knowledge is stated, including links to relevant resources. Information on obtaining source code, installing prerequisites, and building/testing the software is provided. Finally the structure of the codebase, including descriptions of all modules and their inter-dependencies, is explained.

## 1 Technologies

Familiarity with all technologies employed in the development and deployment of RBACFS is required to get the most out of this document.

### 1.1 Git

All source code and documentation is maintained in the Git version control system. Downloads for all major platorms are available from the official homepage at `https://git-scm.com/`.

The current canonical repository for the software is publically hosted at `https://github.com/scott-linder/rbacfs`.

### 1.2 Language and Compiler

RBACFS is developed exclusively in the C programming language, using the GNU Compiler Collection (GCC) compiler. The C standard targetted is `c99` with GNU extensions, known as `gnu99`. The version 6.2.1 of `gcc` has been used during development and testing.

### 1.3 Make

GNU Make is used to compile both program and documentation source code into usable programs and documents. Most directories in the source tree, save those containing only static documents, contain one `Makefile`. The version 4.2.1 has been used during development, but any modern version of GNU Make should suffice.

### 1.4 FUSE

Developing a filesystem typically involves modifying kernel code via loadable kernel modules. Security vulnerabilities in this code effect the security and stability of the entire system, and all running programs. To avoid this considerable risk, and to ease development and modification of RBACFS, it is written against the Filesystem in Userspace (FUSE) kernel interface. This allows the filesystem to execute in underprivileged userspace, while still providing the same interface as a filesystem written as a kernel module.

At least a basic understanding of the FUSE API is required to make changes to FUSE-specific code in RBACFS, although every attempt has was made during design and implementation to isolate this code within one module. Resources describing the use of FUSE are available online, such as `https://lastlog.de/misc/fuse-doc/doc/html/`.

Version 26 of the FUSE API is required to run RBACFS. This is available in version 2.9.7 of libfuse.

### 1.5 Operating System

Any operating system which supports a gnu99 compatible compiler, and version 26 of the FUSE kernel interface should be capable of compiling and running RBACFS. However, the system has be tested exclusively on 32-bit versions of Kali Linux 2016.2 and Ubuntu Linux 16.04. When migrating to a new environment, it is imperative that the full test suite be run to ensure basic functionality.

## 2 Obtaining the Software

The current central git repository is hosted at `https://github.com/scott-linder/rbacfs` and contains all source code and documentation.

## 3 Building the Software

### 3.1 Prerequisites

All development prerequisites have been mentioned above. These include Linux, `gcc`, FUSE, and `make`. The full list includes:

- gcc 6.2.1
- make 4.2.1
- libfuse 2.9.7
- virtualbox 4.3.36
- Kali Linux 2016.2 *or* Ubuntu Linux 16.04

### 3.2 Configure

There are no separate configuration steps to build the software. As only a limited set of operating systems are directly supported, the software is already configured to build and function properly.

### 3.3 Make

The software can be built from the `src/` directory with the following command:

```
make
```

which will create a single binary named `rbacfs`.

### 3.4 Install

The software can optionally be installed from the `src/` directory after it has been built with the following command:

```
sudo make install
```

which will simply copy the `rbacfs` binary to `/usr/local/bin`.

The installation prefix may be overridden by setting the `PREFIX` environment variable (the default being `/usr/local`.) For example, if the desired installation directory is `/usr/bin`, the following command may be used:

```
sudo env PREFIX=/usr make install
```

## 4 Security Considerations

RBACFS cannot provide true mandatory access control at the VFS level. This means it cannot actually enforce access control on top of existing filesystems (e.g. ext4). In order to emulate this behavior, RBACFS "shadows" existing directory hierarchies, applying mandatory control first and then passing requests through to an existing filesystem.

Requests are thus also subject to discretionary access control checks based on the effective `uid` and `gid` of the `rbacfs` process.

One way to emulate true mandatory controls is to create a dedicated `rbacfs` user and group to run the process, change the owner and group of all files to be protected to `rbacfs`, and set all permissions to `0700`.

Thus, only the `rbacfs` user (and the `root` user) are able to access these files, requiring all other users to go through the shadowed version which implements RBAC.

## 5 System Overview

### 5.1 Modules

The software is divided into internal libraries called "modules". The dependencies of these modules are as shown in figure 5.1.

#### 5.1.1 fuse

The `fuse` module communicates with the FUSE library to actually mount the filesystem and implement the RBAC policy in the respective filesystem system calls.

#### 5.1.2 policy

The `policy` module converts the parse tree from the `parse` module into and searchable datastructure to efficiently service queries made by the `fuse` module.

The two primary interfaces provided are the `user_role` and `obj_role_perms` hashmaps. Given a user, the `user_role` hashmap returns a list of roles assigned to that user. Given an object and a role, the `obj_role_perms` hashmap returns a list of permissions the given role has to the given object.

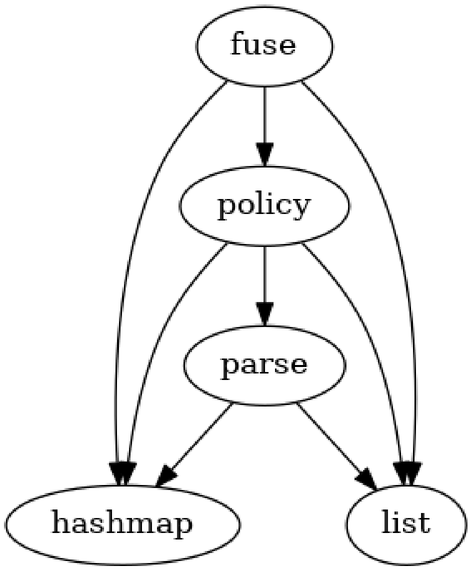

Figure 1: Module dependencies

### 5.1.3 parse

The `parse` module reads the RBAC definition file provided by the user and lexes and parses it into a list of definition structs. The libraries used for lexing and parsing are Flex and Bison, respectively.

The grammar of the definition language, in EBNF, is

```
<input> ::= <def-list>
<def-list> ::= <def> | <def-list> <def>
<def> ::= "user: " <id-list> <id-list>
            | "object: " <id-list> <perm-list> <opt-recursive> <path>
<id-list> ::= <id> | <id-list> "," <id>
<perm-list> ::= <perm> | <perm-list> "," <perm>
<opt-recursive> ::= "-r" | ""
<path> ::= /[a-zA-Z0-9_.\-/]*
<id> ::= [a-zA-Z0-9_.\-/]+
<perm> ::= "r" | "w" | "x"
```

### 5.1.4 list and hashmap

The final two modules implement simple, generic versions of a linked list and hashmap, both of which are used repeatedly throughout the other modules.

### 5.2 Module Structure

Each module contains a `Makefile`, `lib.c`, `lib.h`, and any other source files required to build the library. Each `Makefile` contains a rule for building `lib.a`, a static library which exports at least those functions found in `lib.h`. Thus, modules may depend on each other's `lib.a` and discover function prototypes in each other's `lib.h`.

### 5.3 Control Flow

The program entry point is defined in `rbacfs.c` in the root of `src/`. The last argument is used as a filename for the `parse` module to generate a list of definitions, and then definitions are passed to the `policy` module. Finally, the remaining commandline arguments, along with the policy structure, are sent to the `fuse` module, which mounts the filesystem and begins listening for filesystem accesses.

## 6 Updates

Updates can be applied simply by going through the same set of steps found in sections 2 and 3 above. Re-installation is equivalent to updating.