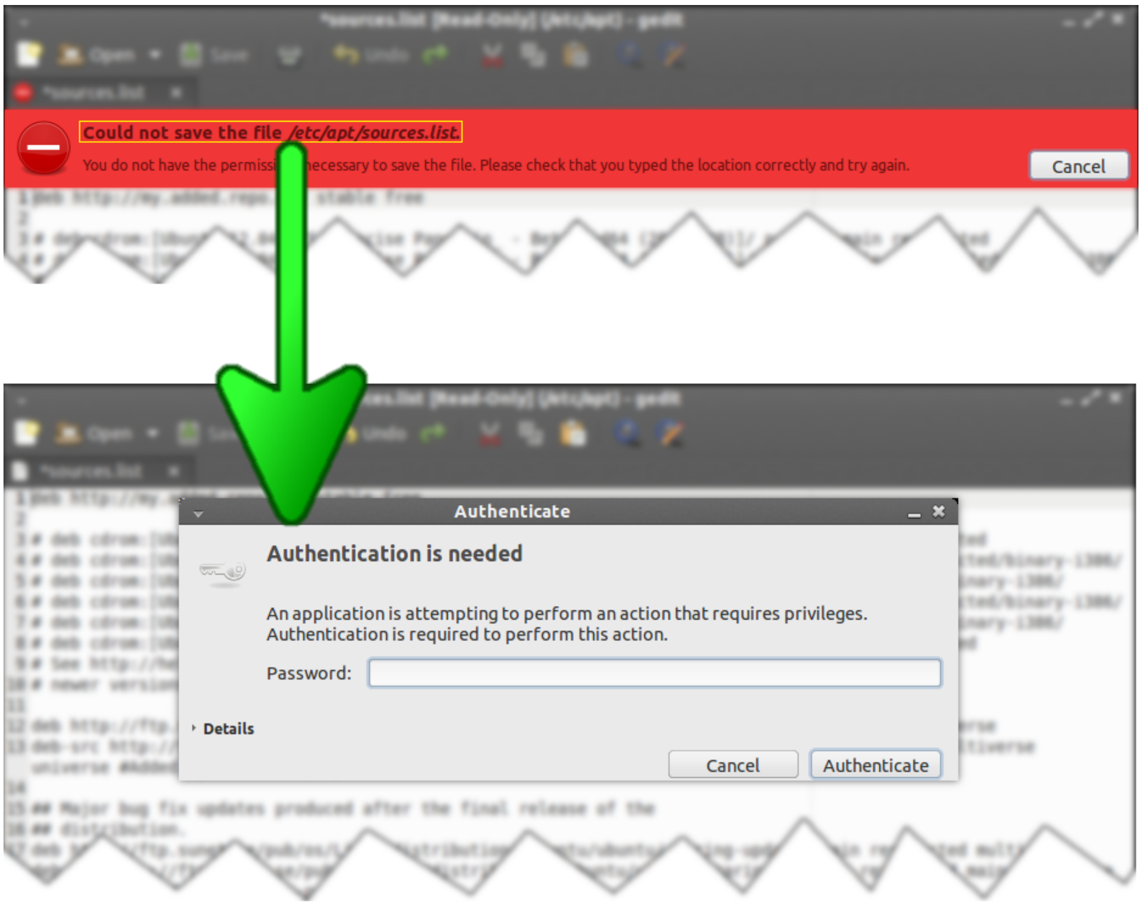

Could not save the file /etc/apt/sources.list.

You do not have the permissi necessary to save the file. Please check that you typed the location correctly and try again.

Authentication is needed

An application is attempting to perform an action that requires privileges.
Authentication is required to perform this action.

Password:

Implementing an authorization policy on I/O level in GNU/Linux

Bachelor's thesis in Computer Science and Engineering

JEAN-PHILIPPE GREEN
MATTIAS HOLMBERG
FILIP LEVENSTAM
TOBIAS TILLSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2013
Bachelor's thesis 2013:10

BACHELOR'S THESIS IN COMPUTER SCIENCE AND ENGINEERING

Implementing an authorization policy on I/O level in GNU/Linux

JEAN-PHILIPPE GREEN
MATTIAS HOLMBERG
FILIP LEVENSTAM
TOBIAS TILLSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2013

Implementing an authorization policy on I/O level
in GNU/Linux
JEAN-PHILIPPE GREEN
MATTIAS HOLMBERG
FILIP LEVENSTAM
TOBIAS TILLSTRÖM

Bachelor's thesis 2013:10
ISSN 1654-4676
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
Illustration of the problem with saving files and its solution

Chalmers Reproservice
Göteborg, Sweden 2013

Implementing an authorization policy on I/O level
in GNU/Linux
Bachelor's thesis in Computer Science and Engineering
JEAN-PHILIPPE GREEN
MATTIAS HOLMBERG
FILIP LEVENSTAM
TOBIAS TILLSTRÖM
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The purpose of this project has been to implement enhanced functionality for privileged file operations when using graphical programs in the GNU/Linux operating system. Today, administrative tasks are done by acquiring privileges before the program in question is started. One goal of this thesis is to show how to make administration easier, by instead requesting authentication when an operation is to be performed. When working with a text editor such as Gedit, it is often possible to open system files and make changes to the loaded text. Saving these changes will however be impossible, due to the user not having write permission on the file. The ideas presented in this report will give the user the possibility of having this action authorized, making it possible to save.

Implementations of these ideas can also enhance the security of the system by allowing less code to be run with elevated privileges. Instead of running Gedit with higher privileges, only the operation to save the changes will be done privileged. Less code running with the power to change system files means that if a vulnerability is found in some part of the system, there is less risk of an attacker using it for an intrusion.

The results of this project are twofold: (1) A mechanism has been created for changing a user's permissions on a file. This acts as a helper program for other programs to use when lacking permissions on a certain file. This helper program uses Polkit for authentication and, if the user is authorized, elevates the user's permissions on the affected file. The program can now continue to perform the requested file operation. The user's permissions on the files will be restored after a set amount of time. (2) The other result consists of guidelines on how to complete this task without changing any permissions on files. Instead this alternative solution can offer the same functionality in a more straightforward way. This by relaying the file operations to a custom made backend.

Keywords: Authorization, Graphical User Interface, Access Control List, Polkit

Nomenclature

ACL A list of permissions attached to a file. (Access Control List).

Backend Processing that occurs behind the scenes in software and is not seen by the user.

Client A program that as part of its execution sends requests to a server.

Daemon A process running in the background, usually without immediate contact with the user.

D-Bus A message bus for communication between processes. (Inter-process communication)

Distribution An operating system kernel, such as Linux, bundled with software utilities such as a graphical environment and a package manager.

GIO A library providing a virtual filesystem API. (Gnome Input/Output).

GLib A general purpose library providing data structures and utility functions. (Gnome Library).

Gnome A common desktop environment. (GNU Network Object Model Environment).

GNU A operating system based on Unix conventions. (GNU is Not Unix).

GVfs A virtual file system for the Gnome desktop. (Gnome virtual file system).

I/O The communication between a computer and the outside world. (Input/Output).

IPC A mechanism for a process to communicate with another process. (Inter-process communication)

Kernel The main component in an operating system acting as a bridge between applications and the hardware.

Mount Making a device available in the file system tree.

Library A collection of functions provided for other programs to use.

Linux The kernel replacing the Hurd kernel in GNU, forming the operating system GNU/Linux.

Permission The permissions flags define what different users are allowed to do with files.

Polkit(PolicyKit) An authorization framework for communication between unprivileged and privileged processes.

POSIX A family of standards for system calls in operating systems defined by IEEE. It consists of several parts including standards for core services, thread control and real-time services.

Process An instance of a program. it is possible to have several processes running one program.

Protocol A family of rules used for defining the communication between different end points.

Proxy The procedure of relaying an action to another process.

Root An abstract user that is the owner of all system files and also has the highest level of privileges.

Server A program that answers requests from client programs.

Setuid A flag that when set on a program will execute it with the privileges of the owner rather than the calling user. It can also be the system call for executing processes as another user. (Set user ID upon execution).

Shell script A script written for a command line interpreter of an operating system.

Stream A data flow, or more precise, a sequence of information that is transmitted.

Sudo A program for running a program or command as another user. Running as root is default. (Substitute user do).

Unix An operating system with conventions that are used in many other operating systems, such as GNU.

User A user is an agent using services on a computer. It may, but does not necessarily, represent an actual person.

Virtual Memory The memory visible to a process.

Contents

Figures

Listings

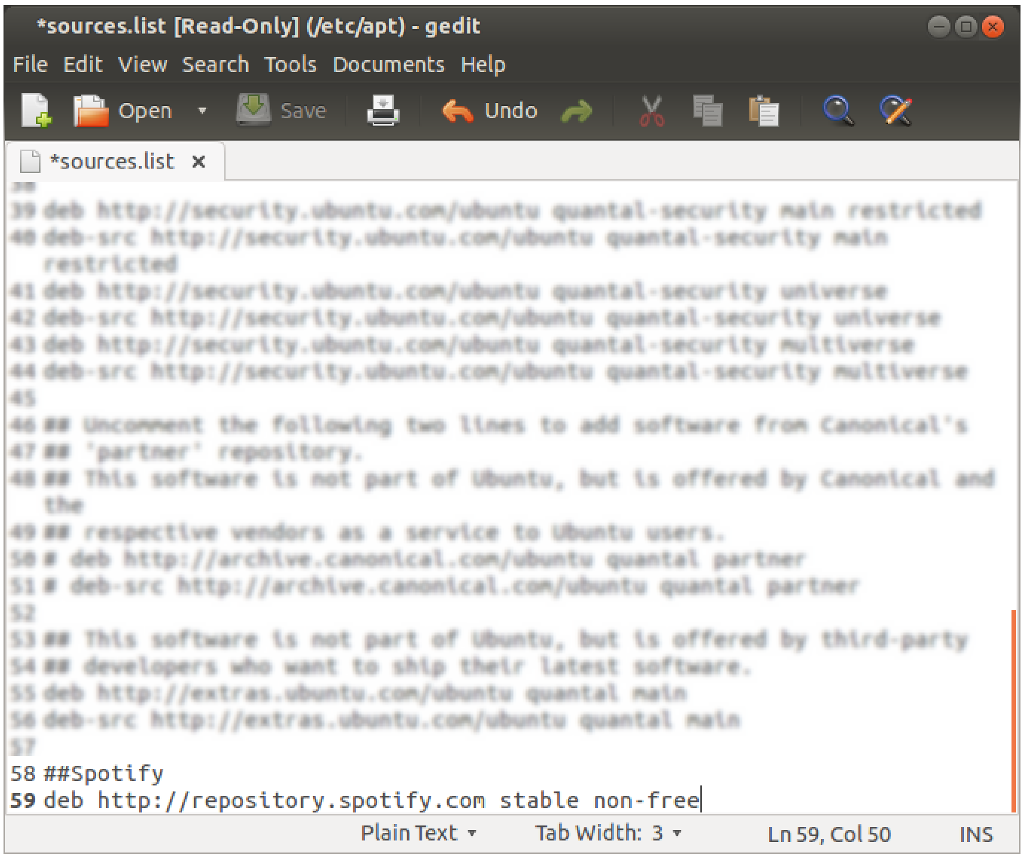
1 Introduction

GNU/Linux is an operating system based on Unix conventions [1]. It has previously been considered user-unfriendly which may have contributed to its low popularity, compared to other operating systems. During recent years several distributors have developed more user friendly distributions, such as Ubuntu, and its popularity has increased. In this project, user friendliness is taken even one step further by improving the authorization framework for administrative tasks.

In GNU/Linux users may have different permissions to, read, write or execute different files. A process started by a user also inherits the user's permissions. This is important for the system security since it means that a process started by a user only can modify files the user has correct permissions on (see Section 2.7). There is also a user called root that has full permissions on all files; in particular, root is the owner of most system files. Any other user needs to authenticate as an administrative user to get permission to modify the files owned by root. Thus malicious code can only modify system files if it runs with root privileges. This is an important security feature of GNU/Linux systems.

Root, as mentioned above, is the owner of the system files. Other users can read most of these files but will normally not have permission to modify them. To this day advanced users that need to modify these files using a graphical editor, such as Gedit, are required to run the program as root. Otherwise it would only be possible to open and read these files, but not to save any changes.

In Figure 1.1, two rows have been added to /etc/apt/sources.list, a file used to define software sources in Debian-based distributions such as Ubuntu. The file is owned by root, but Gedit was executed from an unprivileged user. Due to this, it is opened in read-only mode, a mode which enables reading but not writing. It is still possible to change the content of the loaded text, but saving is disabled. Any attempt to use the "save as" function to the same path is denied.


*sources.list [Read-Only] (/etc/apt) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
*sources.list
Plain Text
Tab Width: 3
INS

```
deb http://security.ubuntu.com/ubuntu quantal-security main restricted
deb-src http://security.ubuntu.com/ubuntu quantal-security main restricted
deb http://security.ubuntu.com/ubuntu quantal-security universe
deb-src http://security.ubuntu.com/ubuntu quantal-security universe
deb http://security.ubuntu.com/ubuntu quantal-security multiverse
deb-src http://security.ubuntu.com/ubuntu quantal-security multiverse

## Uncomment the following two lines to add software from Canonical's
## 'partner' repository.
## This software is not part of Ubuntu, but is offered by Canonical and the
## respective vendors as a service to Ubuntu users.
# deb http://archive.canonical.com/ubuntu quantal partner
# deb-src http://archive.canonical.com/ubuntu quantal partner

## This software is not part of Ubuntu, but is offered by third-party
## developers who want to ship their latest software.
deb http://extras.ubuntu.com/ubuntu quantal main
deb-src http://extras.ubuntu.com/ubuntu quantal main

##Spotify
deb http://repository.spotify.com stable non-free
```

Ln 59, Col 50

Figure 1.1: *Gnome's default text editor Gedit in read-only mode. Notice how the save button is disabled and the title includes a "[Read-Only]" tag.*

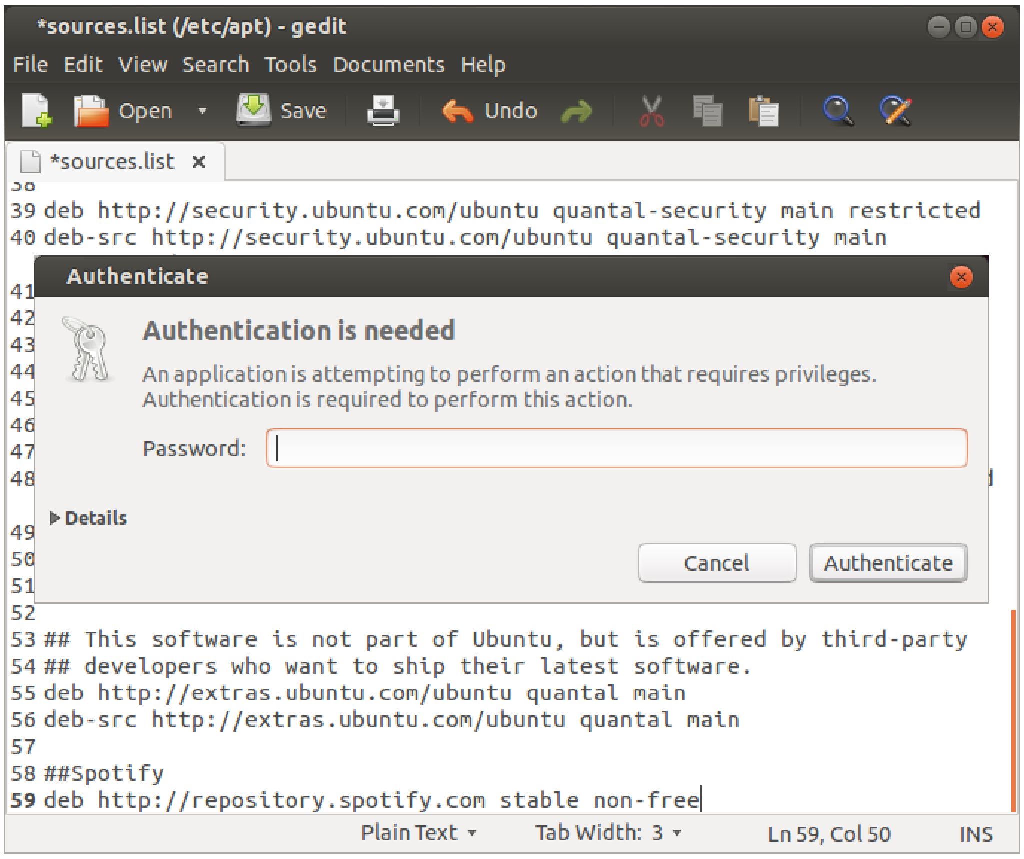
Another issue arises when a user wants to use a file manager to do administrative tasks. In order to do this, the entire program needs to be started with root privileges. This is a bad idea from several points of view. It makes the system more vulnerable; if the program has any bugs that can be exploitable, an attacker could gain control over the entire system. It also increases the risk of damaging the system by mistake, as the press of a button could delete important files.

Polkit, formerly PolicyKit, is an authorization framework created to let unprivileged programs request privileged actions. One such software is Ubuntu Software Center that requires extended privileges for installing software. Polkit uses rules that allow arbitrarily fine-grained control over which users are authorized to request certain actions (see Section 3.2).

1.1 Purpose

The purpose of this project has been to implement authorization for file operations in a user friendly and secure way. Implementation of this functionality on a general level means that existing software is able to utilize its functionality without modification. Another goal has been to reach as many users as possible. In order to accomplish this, several different approaches have been investigated from the viewpoints of usability, security, and their possibilities of being widely accepted by the Open Source community.

Today, if a user wishes to edit a file but lacks proper permissions, the entire program needs to run as another user with the required privileges for the file. The approach presented in this report will instead allow authorization once a privileged action is to be performed, as illustrated in Figure 1.2.



```
deb http://security.ubuntu.com/ubuntu quantal-security main restricted
deb-src http://security.ubuntu.com/ubuntu quantal-security main
```

Authentication is needed

An application is attempting to perform an action that requires privileges. Authentication is required to perform this action.

Password:

```

## This software is not part of Ubuntu, but is offered by third-party
## developers who want to ship their latest software.
deb http://extras.ubuntu.com/ubuntu quantal main
deb-src http://extras.ubuntu.com/ubuntu quantal main

##Spotify
deb http://repository.spotify.com stable non-free
```

Figure 1.2: *Gedit is not running as root but it is still possible to save after authorization.*

1.2 Requirements

It is important that an implementation meets certain criteria. In order for it to be stable, transparent, and secure it is crucial that the following is observed.

1.2.1 Function

The first requirement is functionality for acquiring permissions for single actions rather than entire processes. Processes should be able to increase their privileges for a specific action. Only when the process is to perform that action, authorization is requested. As soon as the action has been performed, the process is to relinquish the elevated permissions.

1.2.2 Interoperability

The method of acquiring privileges for an action was to be implemented at such a low level that no specific program has to be rewritten. The communication between the programs and the file system is instead handled in a different way. The program itself will not even know that the action required higher privileges. The result of the action will be returned to the program by an unprivileged process.

1.2.3 Security

Previously, if a user wished to edit a file while not having the proper permissions for it, the entire program needed to run with the permissions of the owner of the file. This contradicts a rule of thumb in security [2], namely letting as little code as possible run with higher privileges.

New software very often introduces new bugs and vulnerabilities so it is important that sound principles of secure coding are followed when implementing it.

The following principles should be followed in order to not compromise security:

Least privilege Every process runs with the lowest set of privileges required for its task(s).

Revocation of privileges As little code as possible runs with elevated privileges. If a privileged program is necessary in order to perform certain tasks, the program is to revoke its privileges and reinstate them only where needed.

1.2.4 Performance

As in most software, an implementation should not have a significant influence on the system performance. In other words, as few instructions as possible running for each dedicated task. This is especially important in time-critical stages.

1.3 Limitations

There was a higher priority for a comprehensive solution to one desktop environment over a limited solution to several. No liable statistics on the most popular desktop environment exist. However, as of January 29, 2013 the distributions most commonly searched for the last six month were Linux Mint, Mageia, Ubuntu, Fedora, and Open Suse [3]. All these except Mageia and Open Suse have desktop environments based on Gnome by default. Since an implementation was desired on as general a level as possible, the focus has been on modifying GIO, the software library used in Gnome for file management. GIO handles many types of file operations. Due to time constraints, this report does only concerns read and write operations.

1.4 Method

There are several approaches possible for achieving the desired functionality of this project. The approaches were evaluated considering how they would affect the stability, security, and performance of the system. In extent it was also important that the implementation was to be transparent for programs and that no code in other libraries or programs needed to be modified.

This report presents two approaches on how file operations can be handled. One is relaying the operation to another process that has the required privileges, the other is temporarily changing the permissions of the affected file. Modification of the kernel itself is another possibility, but was determined out of scope for this project.

1.5 Layout of the report

In Section 2, basic concepts of the GNU/Linux operating system that are necessary for understanding this report are described and explained. It focuses on some general knowledge of the GNU/Linux as an operating system and the capabilities and functions it offers. It is followed by Section 3 explaining concepts, libraries and tools that are essential for this report. The approaches and methods are further explained in Section 4. Results and discussion of advantages and disadvantages of the implementations, as well as general discussion of the workflow, are made in Section 5. Section 6 summarizes the project and aims to provide guidance for possible future work.

2 GNU/Linux basics

To understand the different approaches that have been considered, it is important to understand a few mechanisms and concepts in GNU/Linux. This section will provide the necessary background.

2.1 Users and groups

The term *user* has several meanings in computer science. In this report a user is a concept of an agent that is allowed to own files. A user may represent an actual person but that is not a necessity. The user root is, for example, an abstract user that owns all system files, and in general the only user with permission to modify these. However, in some distributions another user may receive these privileges if the user is a member of a group such as *sudoers*. Such membership generally allows the user to use a program called `sudo`. A user that is a member of the group wheel also achieve these additional privileges. A user can be a member of several groups and a group can have several members. Being a member of additional groups may add additional access to files owned by these groups (see Section 2.3).

2.2 Directories and files

In GNU/Linux, files, hard drives, and directories are handled in the same way. Directories as well as hard drives are files acting as containers of other files and directories. The top level in the filesystem hierarchy is denoted "`/`" and contains all folders and drives detected by the system. Most directories are owned by the user root. Common exceptions include the home directories of other users, that usually exist under `/home`, and files that user processes have created under `/tmp`.

2.3 Unix permission model

Unix-like operating systems traditionally use a permission model with nine bits per file that control file access. Of the nine bits, each group of three correspond to one of three different classes; owner, group, and others. As seen in Listing 2.1, the three classes each can have permission to read (`r`), write (`w`), and execute (`x`). A dash (`-`) in place of one of the permission bits represents a permission that the user does not have. For example, the string `-rw-r----- 1 root     wheel    0 May 18 12:15 rootfile` means that the file `rootfile` may be read and written by root, and read by any member of the wheel group.

The row in Listing 2.1 starting with a `d` represents a directory. For directories, the three characters `rwx` can be seen as representing permission to list files, to add or remove files, and to open the directory, respectively.

In addition to the nine permission bits, there are three bits that represent different attributes. These are the set user ID, set group ID and sticky bits. The setuid and setgid bits are explained in Section 2.7 and the meaning of the sticky bit depends on what kind of file it is set on. It was traditionally used on executable files to keep them in memory rather than being swapped out, but that functionality is obsoleted by virtual memory (see Section 2.6). The sticky bit can be used on directories with write permission for all users to disable the creation of symbolic links from them, this is a security feature of recent Linux kernels [4]. For ordinary, non-executable files, the sticky bit never had any function.

Listing 2.1: Files with different permissions and owners.

```
$ ls -l
drwxr-xr-x  2 mattias  wheel  512 May 18 12:15 directory
-rwxrwxrwx  1 mattias  wheel    0 May 18 12:15 executablefile
-rw-r-----  1 root     wheel    0 May 18 12:15 rootfile
```

2.4 Access Control Lists

The traditional UNIX permission model is simple but has many limitations. Therefore, Access Control Lists (ACLs) have been introduced as a more flexible way of controlling file permissions. Several, not quite compatible, implementations of ACLs exist for GNU/Linux but in this thesis only the POSIX ACL system is considered [5].

A minimal ACL contains three different entries - owner, owning group and others - corresponding to the three classes of the traditional permission bits. In contrast, extended ACLs also contain a mask entry and one or more entries of the types *named user* or *named group*.

In POSIX ACLs the additional entries are implemented by extending the group class with fields for named users and groups. In order to allow applications that are not aware of ACLs to function correctly, and in particular not grant them any permissions they should not have, the group class permission entry is redefined. That permission is used as a mask, so that the effective permissions of any entry in the group class, be it owning group, named user, or named group, is effectively the intersection of that entry's permission and the permission of the mask [5]. See Listing 2.2 for an example.

Listing 2.2: Access Control List of regular file

```
$ ls -l acltest.foo
-rw--w-r--+ 1 mattias mattias 0 Apr 22 13:35 acltest.foo
$ getfacl acltest.foo
# file: acltest.foo
# owner: mattias
# group: mattias
user::rw-
user:user1:-wx  #effective:-w-
group::rw-      #effective:-w-
mask::-w-
other::r--
```

The user interface to the POSIX ACL system are the commands `getfacl` (get file access control list) and `setfacl` (set file access control list). As seen in Section 2.2, `getfacl` corresponds to the `ls` command in that it gives information about all the permissions for a file. Similarly, `setfacl` is the ACL equivalent of `chmod` [6], and can set permissions for all the entries mentioned above.

2.5 Process management

In GNU/Linux, processes are managed by the kernel. A process may start child processes with system calls such as `clone()` or `fork()`. These functions create a child process that inherits most of the preferences from the parent. The commands are quite similar but there is a difference when it comes to how much of the information the process will inherit from the parent as well as where the child process will start to execute. A child process will not have the same id (pid) as the parent though, rather the parents id will be set as the ppid (parent pid). System calls such as `execve()` or `execvp()` can be used for a process to start executing as another program.

It is possible for a process to change what user it is running as. However, the kernel limits this functionality to the root user. This means that it is only possible to decrease the privileges of a process, given that root has full privileges to all files. This is crucial for the security since it otherwise would be possible for any program or script to gain full control of the system.

2.6 Memory management in Linux

A central concept when it comes to memory management in Linux is virtual memory. Instead of having all the processes of an operating system access the physical memory directly, each process has its own virtual memory space. This is then mapped to the physical memory using page tables [7].

There are several benefits of using virtual memory in an operating system, including security and CPU optimization [7]. For this project, the most relevant aspect is isolation.

Since processes are isolated from each other memory-wise; they normally cannot read or modify each other's memory. This is partly due to protecting programs from rogue programs. It is however possible to have multiple processes share a memory segment if needed, as seen in Figure 2.1. The system calls `shmget()` and `mmap()` are useful when writing such programs.

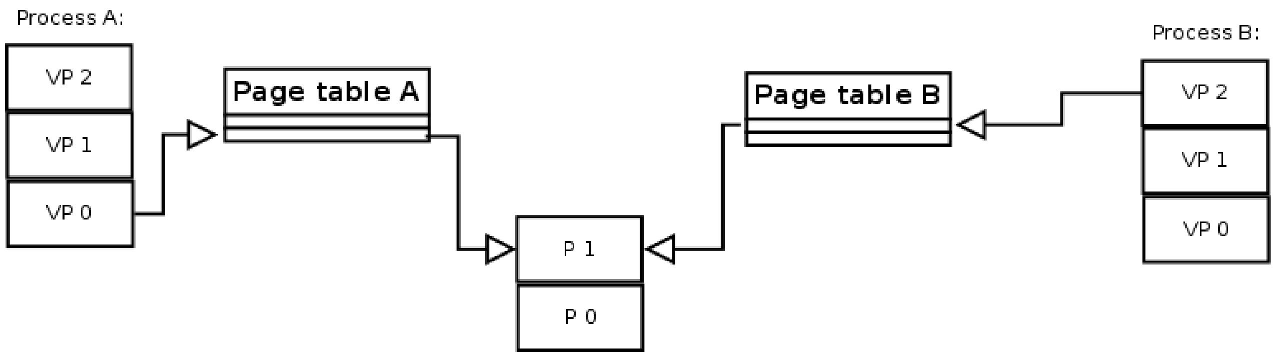


Figure 2.1: *Simplified depiction of two processes sharing the same physical memory. Virtual pages are denoted 'VP' and physical pages are denoted 'P'.*

2.7 The user ID model and setuid

In GNU/Linux and other Unix systems, each user has a unique user identifier - a UID. Beside the uniquely identifying UID, each user also has one or more group identifiers (GID). The group IDs will not play an important part in the present discussion but two things are worth noting. Firstly, group IDs can be used to determine file access. This can be useful for collaboration between users. Secondly, membership in an administrative group - commonly called "admin" or "wheel" - is normally what determines the right to run programs with elevated privileges, for users other than root.

2.7.1 UIDs of processes

Processes can have at least three different UIDs in modern GNU/Linux systems: real, effective and saved UID. The real UID (ruid) is the UID of the user that started the process, however, since UIDs can be changed using the `setuid` system call, the effective UID (euid) is the more interesting one. Together with the effective group ID (egid), and possibly supplemental GIDs, the euid is what determines the ownership of files that the process creates. Furthermore, it is used to determine whether the process can access existing files [8]. See Listings 2.3 and 2.4 for a demonstration on how the ruid and euid can differ.

Listing 2.3: Test program main.c that prints its own ruid and euid

```c
#include <stdio.h>
#include <unistd.h>
int main(){
    printf("Current process has: \nRUID: %u\nEUID: %u\n", getuid(),geteuid());
    return 0;
}
```

Listing 2.4: "ls -l" shows that the setuid bit is set on the "main" executable. The resulting change of UIDs is shown in the program output.

```
$ ls -l
total 12
-rwsrwxr-x 1 root jp 8625 maj 17 16:38 main

$ ./main
Current process has:
RUID: 1000
EUID: 0
```

All executable files can have the setuid bit set. When any user that has execute permission on the file runs it, the process will have as euid the UID of its owner. Furthermore, the saved UID (suid) will be set to the same value as the euid when the setuid process starts. There is also a setgid bit that sets the effective group ID in a similar fashion.

The saved UID is used whenever a process has temporarily changed its euid and needs to change it back. As mentioned in Section 1.2.3, it is good practice for privileged processes to drop their privileges and only reacquire them when needed, and this is where saved UID is useful.

Setuid programs are most often used to let ordinary users run programs with root privileges. One example where this is necessary is the passwd program which needs to modify password files that only root has access to [9].

2.7.2 Security aspects of setuid programs

There are inherent security risks to letting users run programs with root privileges. If not written properly, they can through exploits let a regular user perform root commands. A well-known exploit comes from setting the setuid bit on a shell script [10]. A shell script is first opened by the kernel. The kernel reads the first line, which tells it what interpreter to use (for example `#!/bin/bash`). Then, it opens the interpreter with the path to the script as an argument. If a legitimate setuid shell script exists, a symbolic link could be created to it, and right after it has been executed (but before the interpreter has been called) the link could be redirected towards another script. The other script would now run with root privileges. To cover this exploit, most systems ignore the setuid bit on shell scripts.

In order to secure a setuid program there are certain precautions to take. The program needs to clean up its environment before calling any external programs. If for example a user was able to retain the PATH variable that the user had previously defined and the program naively did not specify the full path to an executable, it would be easy for the user to obtain a root shell. For other, less obvious, ways of exploiting environment variables, see [11].

It is also necessary to check the return values of all functions that are used in the setuid program, instead of blindly assuming that they will succeed. If for example the seteuid system call failed to change euid from 0 to 1000 before a new program was started, the new program would have root privileges instead of the privileges of UID 1000.

System signals need to be handled appropriately. There are situations where overly complex signal handlers can be attacked by sending several signals and causing problems on the heap, which in turn can let the attacker gain root access [12].

3 Project-specific libraries, concepts and tools

In this section tools, mechanisms and libraries that are essential for this project are explained.

3.1 D-Bus

D-Bus is a system for IPC (inter-process communication). Many programs use this for communication with the operating system as well as with other programs. There are two different ways to communicate using D-Bus, one is to establish a communication channel between two processes and the other is to use a D-Bus-daemon to contact all listening processes.

D-Bus supplies two types of daemons, a system daemon which listens to the system bus, and a session daemon which only listens to a bus of the current session. The systemwide message bus is typically used for events such as newly added hardware, which may be interesting for any program in the system, while the session bus is typically used for communication between user applications[13].

The purpose with D-Bus is to create a more integrated experience for programs in the operating system. It is for example used when a hard drive is plugged in to send a signal to all programs that may be interested of such information. It is also used for contacting polkit (see Section 3.2).

3.2 Polkit

The system architecture of Polkit mainly consists of three parts; a subject, a mechanism, and an authentication agent. In this context, the subject is the unprivileged program that wants to perform a privileged action. This action can for instance be an attempt to read a file when the subject lacks the reading permission to do so. Through inter-process communication it tells the mechanism, which is a privileged helper program, that it wishes to perform this privileged action. In order to determine if the subject is to be granted this action, the mechanism speaks with the polkit daemon via the D-Bus system-message-bus (see Section 3.1). The polkit-daemon has rules dictating which actions each user is allowed to perform. It also communicates with an authentication agent that provides means for the user to type in the desired username (usually root) and password. Finally, the mechanism is informed whether or not the user is to be trusted, and performs the requested action.

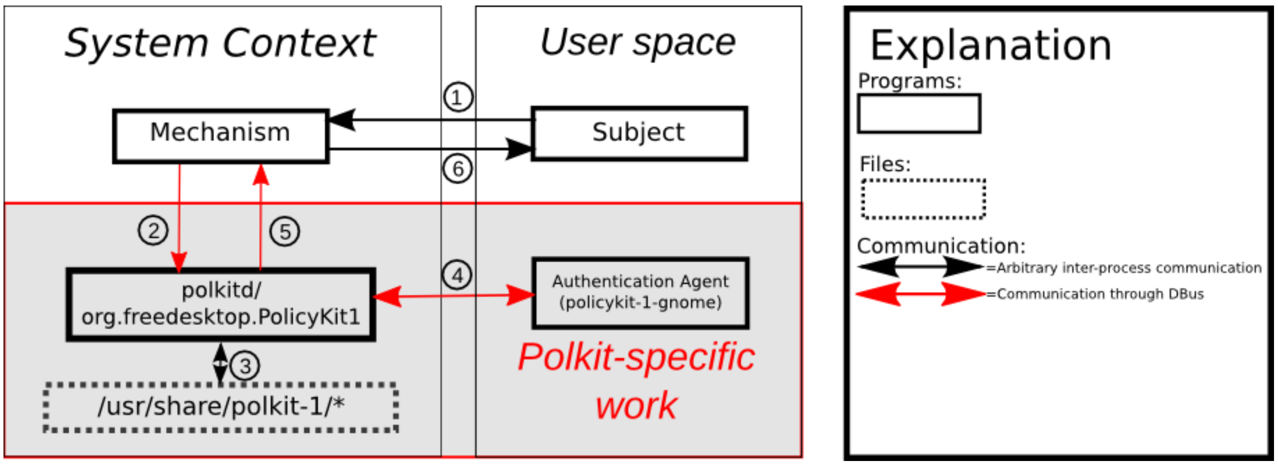


Figure 3.1: *Flowchart of general polkit usage.*

A detailed flowchart of this is shown in Figure 3.1 and described with the following steps.

1. The subject asks the mechanism to perform a privileged task.
2. The mechanism asks polkitd for permission.

3. Polkitd checks the rules for the specified task in related files.
4. If needed, polkitd asks the agent for authorization.
5. Polkitd tells the mechanism whether or not authorization was granted.
6. If authorized, the mechanism performs the task and returns the result to the subject.

Since the mechanism is a privileged program, there are certain precautions that must be taken in order to not compromise security. First of all, the subject is never to be trusted. Every request must be validated through an authorization process, in this case the polkit authority (polkitd). If the mechanism gains its privileges by being a setuid-program, there are of course more steps to be taken (see Section 2.7).

3.3 Gnome-core

Note: Because the Gnome-core libraries are distributed through a git-repository named glib, the bundle is often referred to as GLib. One of the smaller parts of this bundle is also called glib, which might lead to confusion. The bundle will therefore in this report be referred to as "Gnome-core" and the library as GLib, in order to avoid confusion.

Gnome-core [14] consists of three libraries: GLib, GObject and GIO. Together, these form a general-purpose library that provides data structures and utility functions for other parts of the GNU project, such as the text editor Gedit and the file manager Nautilus. The library is written in C using an object-oriented programming style in order to be stable and portable. Figure 3.2 shows an overview of how components are related to each other, and the following sections describe the main components.

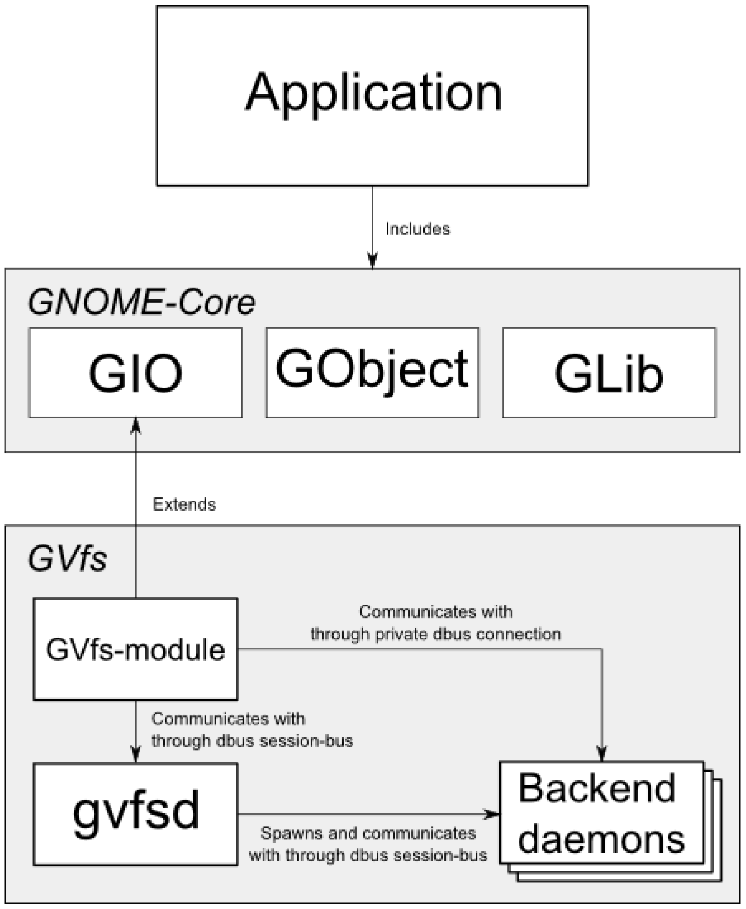

Application
Includes
GNOME-Core
GIO
GObject
GLib
Extends
GVfs
GVfs-module
Communicates with through private dbus connection
Communicates with through dbus session-bus
gvfsd
Spawns and communicates with through dbus session-bus
Backend daemons

Figure 3.2: *Overview of Gnome with focus on GIO/GVfs*

3.3.1 GLib

GLib includes tools and utilities that allows developers to create cross-platform programs. It provides data types, macros, threading, a plugin-system (GModule), type conversions, string utilities, file utilities, a mainloop abstraction and more [14]. Some of the data types it provides are [15, p. 159]:

- Linked lists
- Double-ended queues
- Self-balancing binary trees
- Unbalanced n-ary trees
- Hash tables
- C++ like strings
- Quarks
- Keyed data lists

GLib is widely used in the other parts of the Gnome-core as well as in some other libraries and programs.

3.3.2 GObject

Object oriented programming is a popular programming paradigm for desktop application development, with Java, C++, Objective-C, C#, PHP and Python making up more than 50% of the market [16]. Although C does not have native support for object oriented programming, it is possible to imitate its behavior. GObject [17] was created to make it easier for developers to use objects when developing in C. This is basically done by creating typed structures (struct in C) which includes class-variables and pointer to functions which you declare later, and then initializing the objects with GObject. GObject then takes care of memory allocation, type mechanisms and much more [17].

Generally, Gnome-libraries are written using GObject. The choice of creating an object system in C, instead of just using an object oriented language, has both traditional and technical reasons. The technical aspect is to make it stable, flexible and extensible in order to map it into other object-oriented languages such as Python, Java and Vala, see Appendix A.

3.3.3 GIO/GVfs

When performing I/O operations in GNU/Linux, it is common to use raw POSIX calls such as `read()` and `write()`. These work fine, but for efficient development of desktop applications, there is often a need of a higher level API. Aside from cutting down on development time, such an API should lead to easier integration between different applications. These needs are fulfilled by GIO, a library used by Gnome applications, that provides a virtual filesystem API. It also provides abstractions of network and D-Bus connections.

Files in GIO are identified by GFile [18]. GFile has 6 different access attributes defined by GFileInfo which determine what can be done: read, write, execute, delete, trash and rename [19]. Operations on files are made through GFileOutputStream and GFileInputStream.

In order to allow multiple protocols, such as ftp, http and ssh, there is an extension to GIO named GVfs. GVfs is a set of libraries and daemons that provides everything needed for accessing files on remote servers and devices attached to the computer [20].

GVfs is written using a client-server architecture. The client libraries, which are loaded by GIO-clients, communicate with the server side through D-Bus. The server side consists of one master GVfs-daemon and one backend for each supported protocol. The master daemon, gvfsd, controls spawning of backends, mounting and unmounting drives, and informs clients which backend to use for requested files, while the backends does the actual I/O work. Because the backends are running in separate processes, the system becomes very robust by treating crashed backends as unmounted without affecting the rest of the system [20].

th

4 Suggested implementation methods

Normally a function in GIO returns an error if an operation is unpermitted. With the suggested implementations described in this chapter, a helper program will instead be called, providing authorization for that certain operation. To execute the file operation the process needs to change its UID or have the ACLs of the files changed. The following approaches have therefore been considered.

4.1 Proxy file operations to another process with the required privileges

A helper program is called with the required permissions. After authorization it then acts as a proxy taking the unprivileged file operation, executing it privileged.

A detailed flowchart is shown in Figure 4.1, describing the following steps:

0 The application using GIO/GVfs does not have the necessary permissions for the task and is denied.

1 The operation is therefore proxied to the helper program.

2 – 5 Polkit-specific work as described in Figure 3.1.

6 If granted authorization, the helper program performs the file operation.

7 The helper program returns the result to the application.

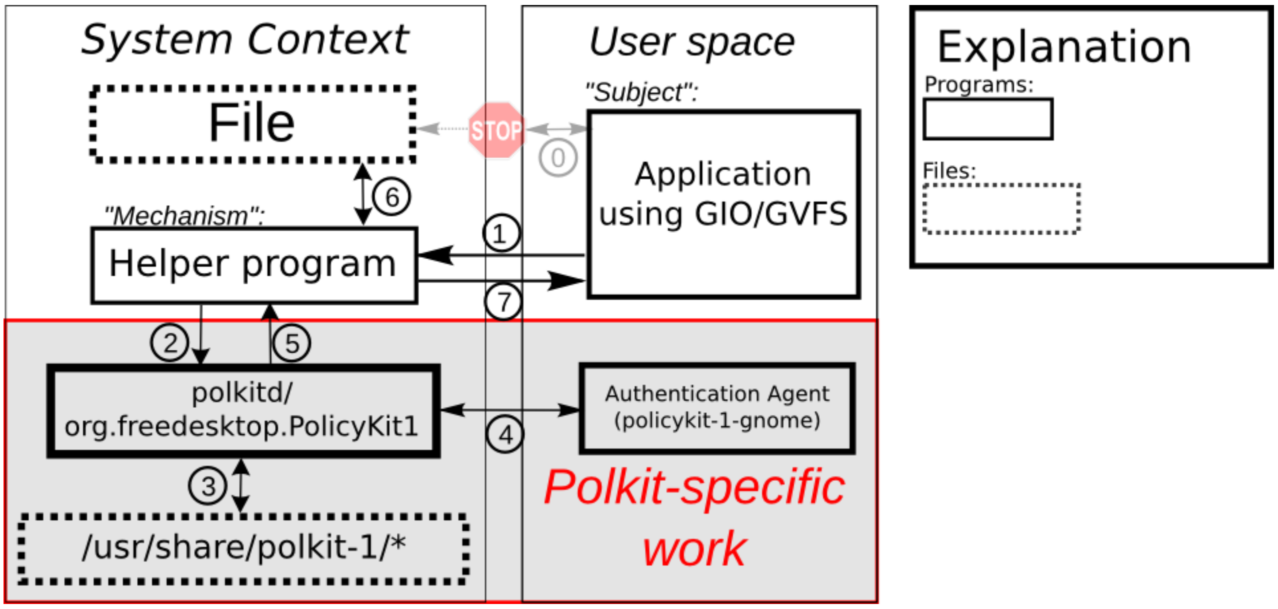


Figure 4.1: *Flowchart of the ideal proxy-implementation.*

4.2 Temporarily changing the ACL

Modifying ACLs can be used as a mechanism to grant or remove access rights a process has on files. Only root and the owner of the file is allowed to modify its ACL.

Therefore, if the calling program does not have the proper permissions to modify a file, it needs a setuid helper-program to change the ACL before attempting to do so. By receiving the path to the file as an argument, the helper program can create a backup of the affected ACL and then modify the permissions. It will need to send a signal to the calling program stating that it can continue with the operation. D-Bus can be used to both send the method call, and to handle the signaling. It is crucial that the ACL of the file is restored at some point, even in the event of a system crash.

The benefits of this approach is that the process does not gain any permissions on any other files than those affected. On the other hand this means that other processes running as the same user also gain permissions on these files.

A detailed flowchart of the ideal ACL-implementation is shown in Figure 4.2.

0 The application using GIO/GVfs does not have the necessary permissions for the task and gets denied.

1 The application asks the helper program to modify the ACL of the file.

2 – 5 Polkit-specific work as described in Figure 3.1.

6 If granted authorization, the helper program modifies the ACL of the file and starts the timer which will be used later to restore the ACL.

7 The helper program returns the result to the application.

8 The application performs the operation on the file.

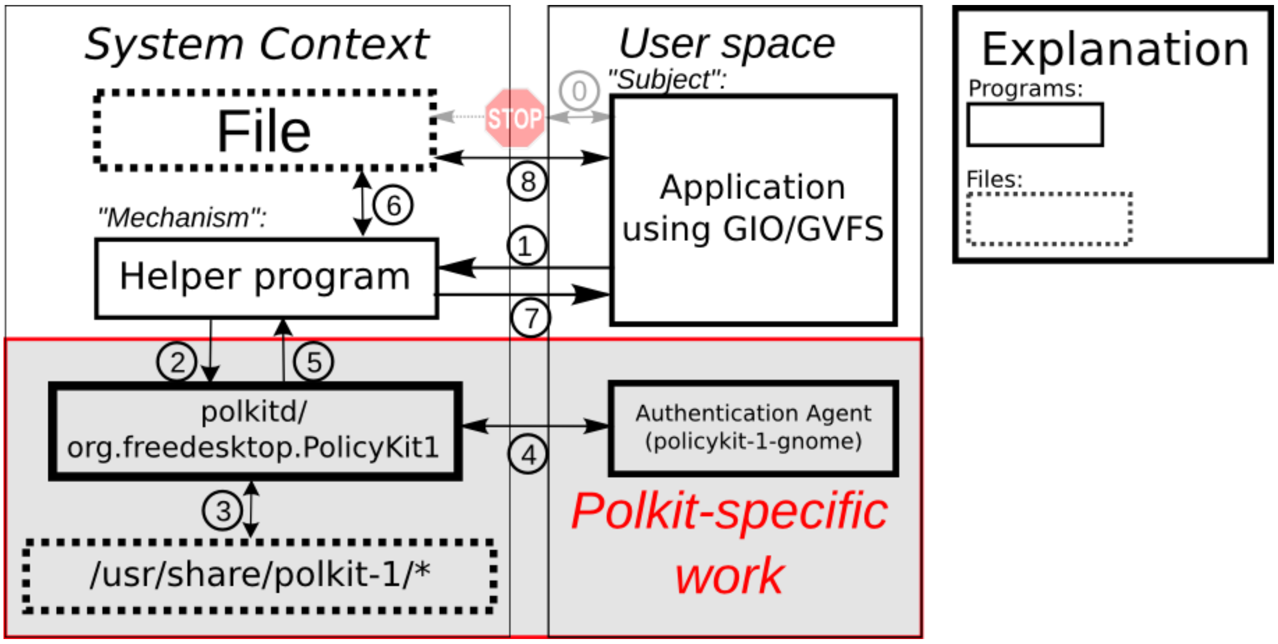


Figure 4.2: *Flowchart of the ideal ACL-implementation.*

5 Result & Discussion

This chapter presents the results of this project and aims to further discuss the different approaches considered, with regards to the requirements (see Section 1.2). Insights are also provided into how the approach of relaying operations could be completed, with regards to recently released documentation. Additionally, a general discussion of the workflow is provided.

5.1 ACL

5.1.1 Result

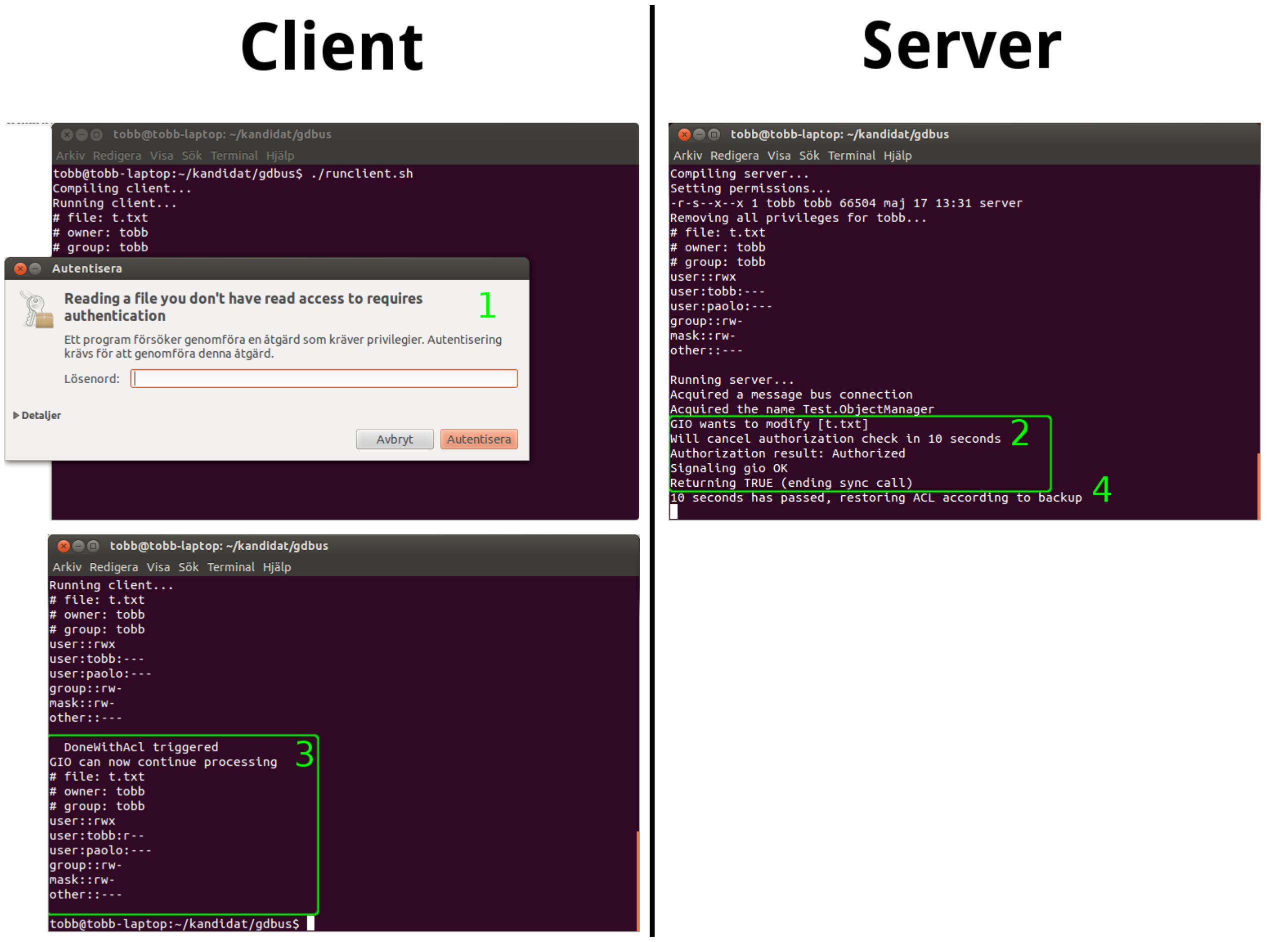
An alpha version of the ACL-approach was created and its code can be read in Appendix B. It is a client/server implementation where the client is to be seen as GIO, and server as the setuid-helper-program. An output of the program can be seen in Figure 5.1, where the procedure is as follows:

1 The client wishes to read a file and is prompted for the root password.

2 The server accepts the password, changes the ACL, and notifies the client.

3 The client has now been notified that it can continue.

4 After a set amount of time, the ACL is restored by the server.


Client
Server
tobb@tobb-laptop: ~/kandidat/gdbus
Arkiv Redigera Visa Sök Terminal Hjälp
Autentisera
Detaljer
Avbryt
Autentisera
1
2
3
4

```
tobb@tobb-laptop:~/kandidat/gdbus$ ./runclient.sh
Compiling client...
Running client...
# file: t.txt
# owner: tobb
# group: tobb
```

Reading a file you don't have read access to requires authentication

Ett program försöker genomföra en åtgärd som kräver privilegier. Autentisering krävs för att genomföra denna åtgärd.

Lösenord:

```
Running client...
# file: t.txt
# owner: tobb
# group: tobb
user::rwx
user:tobb:---
user:paolo:---
group::rw-
mask::rw-
other::---

  DoneWithAcl triggered
GIO can now continue processing
# file: t.txt
# owner: tobb
# group: tobb
user::rwx
user:tobb:r--
user:paolo:---
group::rw-
mask::rw-
other::---
tobb@tobb-laptop:~/kandidat/gdbus$
```

```
Compiling server...
Setting permissions...
-r-s--x--x 1 tobb tobb 66504 maj 17 13:31 server
Removing all privileges for tobb...
# file: t.txt
# owner: tobb
# group: tobb
user::rwx
user:tobb:---
user:paolo:---
group::rw-
mask::rw-
other::---

Running server...
Acquired a message bus connection
Acquired the name Test.ObjectManager
GIO wants to modify [t.txt]
Will cancel authorization check in 10 seconds
Authorization result: Authorized
Signaling gio OK
Returning TRUE (ending sync call)
10 seconds has passed, restoring ACL according to backup
```

Figure 5.1: *The output of our client/server program.*

Some functionality in the server has been implemented using a c-code generator called gdbus-codegen [21]. It processes an introspection XML-file [22] to create 1600+ lines of code and documentation for various functions one might be interested in.

In this implementation, the only part of the generated code needed was a way to call a procedure that was named `changeACL()`, and have a signal attached to it. Listing 5.1 shows the XML-code for this. The translated c-code is much harder to follow and is not included in this report. It can be generated by running the command shown in Listing 5.2. As the server uses Polkit, a very simple policy file has also been created called org.gnome.gio.policy, see Appendix B.

Listing 5.1: Full introspection XML-code:

```
<node>
  <interface name="Test.ObjectManager.gio">
    <property name="DoneWithACL" type="b" access="read"/>
    <method name="changeACL">
      <arg direction="in" type="s" name="pathname"/>
    </method>
  </interface>
</node>
```

Listing 5.2: Command to generate C-code from introspection XML using gdbus-codegen:

```
#!/bin/bash
gdbus-codegen --interface-prefix Test.ObjectManager.
              --generate-c-code generated-code
              --c-namespace Example
              --c-generate-object-manager
              --generate-docbook generated-docs
              auto.xml
```

As seen in Listing 5.1, the procedure `changeACL()` should receive the pathname as an argument from GIO and change its property `doneWithACL` when done.

So the whole procedure becomes as follows: The client makes a synchronous call to the function `changeACL()` located in the server. The argument included is the pathname of the file to be modified, derived from `gfile_get_path(file)`. In turn, the server calls Polkit to let the user try to authenticate. If successful, it makes a backup of the current ACL and then modifies it (using `getfacl` and `setfacl`). Lastly, it triggers its property `done_with_acl`, letting the client know it can finish processing the request. This property will of course always be triggered, even if the user fails to authenticate and the ACL remains unmodified.

An attempt to use the client/server code with GIO was made. When Gedit tries to read a file, one of many procedures that are called in GFile.c is `open_read_async_thread()`. If the effective UID lacks read permission, the procedure returns NULL instead of an input-stream. Thus, instead of returning NULL immediately, the server program was called to give the user a chance to provide the root-password in order to have the ACL changed. Unfortunately the system would freeze with these changes. Either due to bugs in the server program, or `open_read_async_thread()` being the wrong function to implement this in.

5.1.2 Discussion

While the approach of modifying ACLs of files offers the functionality the project was aiming for, it is clear that it is neither as efficient or secure as the approach of porting the operations. Modifying permissions on files is rather to be seen as a workaround of the problem than an actual solution. However, there are several benefits of this approach. It is always known in advance what code will be run as root. This is an advantage over the common situation where the entire editor used to deal with system files needs root privileges. Furthermore, the user process will not have increased access permissions on any other file. This is an advantage if vulnerabilities are found in the program in question, as it limits the damage that can be done by an attacker.

With regards to the requirements in Section 1.2, modifying ACLs is fundamentally connected to the following issues:

- While the processes performing the file operation indeed run without any elevated privileges, in practice all processes running as the same user will have these access rights as well. For example, if Nautilus is used to create a file in `/usr/bin`, any malware running as that user could remove or modify any file in that folder. This would let an attacker modify executables and redirect them to any other malware.

- A lot of code in the helper program is executed every time the user lacks permissions to a file. This inefficiency becomes an issue when dealing with a lot of files, by for instance using a file-manager such as Nautilus. When Nautilus opens a directory it uses GIO for checking what permissions the user has for each file in the directory. With the ACL-solution, it would mean that these lines of codes would be executed once for each file in the directory, which of course will affect the performance of the system. Beside affecting the performance, all that code running as root presents a large attack surface.
- Ensuring that the ACLs are restored correctly in the event of processes being interrupted requires that the backup files are checked. While this may not take a lot of time, it may happen at a time critical stage. If many additional processes were to run at this it would significantly increase the boot time.
- Storing backups requires caution. If the list of backups is exposed, other processes (including malware) will know which files that currently have modified ACLs.

5.2 Proxy

5.2.1 Result

An implementation of this approach was not completed. Some issues encountered are explained below.

5.2.2 Discussion

A natural approach to accomplish the task of this project is to relay operations to a privileged program, and this was indeed the original idea. More specifically, if GIO was to write to file, instead it would just relay its arguments to a helper program that would do the writing. The implementation of this approach failed partly due to improper memory handling, as illustrated by the following example using Gedit.

In order to write to a file, first a reference to a GFile is created. Through this reference a function is called, that either returns an output stream or, if permission is denied, an error. Even if a privileged helper program would open up this stream, two problems arise:

- There is no easy way to send this back to the original program. Shared memory space could be utilized, but as GIO handles files by references, having two programs share references to an object in different virtual memory spaces makes no sense (see Section 2.6).
- Even if the helper-program has the proper permissions to open a stream, sending it to the calling program would have no meaning. *This is due to streams having no privileges of their own, only processes do.*

Although an implementation has not been created, the approach of porting operations to a privileged program has been found to be possible.

The recently released GVfs documentation has made it easier to understand the basics regarding the communication between GIO and GVfs, as well as the their internal workings. Prior to the release of this documentation, the idea of this project was that the implementation was needed to be done in GIO. The GVfs repository was known, but the possibility of implementing a backend in it was not seen.

It is crucial to understand that the GIO source code only includes an abstract class for a virtual file system (in `gvfs.c` and `gvfs.h`), and an implementation of it for local files. When a program wants to access files over a certain protocol, the functionality of GIO is extended by the client-side of GVfs, which in turn communicates with the appropriate backend daemon.

The easiest way to implement this would probably be to create a proxy root backend in GVfs and an URI-scheme that maps to such a backend. From GIO's view, it would be just another network protocol although it is only meant for communication between local processes. If unauthorized, GLocalVfs needs to use the root backend in order to get the desired functionality. Because the GVfs-daemons communicates through the session-bus [20], the backend needs run in the same session as the application using it. The backend would then act as the subject according to the polkit model shown in Section 3.2, and implement the gfile functions by sending them over to a helper program.

There are multiple advantages with a proxy approach, as seen in the following list.

- Processes will always run with the minimum required privileges.
- Only the relayed operation will run with root privileges. By the requirements in Section 1.2.3, this is a large improvement over the current situation where an entire program is run as root.
- No need to do crash-controls on start-up. It is not a good practice to add unnecessary time-demanding procedures to the start-up process.

A possible disadvantage is that more data needs to be handled by the root process, and therefore more data can be used in the wrong way. This should not be a major issue if the program is well structured an the autorization is controlled in a good way through polkit or some other secure authorization mechanism.

5.3 Discussion of our method

It is clear that this project was not undertaken in the most efficient manner. It would have been beneficial if, during the planning process, the project had been divided into smaller pieces. Instead, we set out to solve the problem rather than clearly defining the problem and planning how to solve it. For one thing, this lead to duplicated work when different members tried - and got stuck at - similar solutions. A more structured approach would have let the project get going at an earlier time and might have yielded better results.

As stated in Section 1.3, we desired to bring the benefits of our solution to as many GNU/Linux users as possible. This lead to the choice of the desktop environment Gnome for our implementation. A lot of the relationship between Gnome and its input/output library GIO is explained in Section 3.3.2, but in short GIO is a huge library that is rather poorly documented. Many hours were spent trying to determine the complex relationship between the many parts of GIO and supporting libraries, and even more to figure out where our programs were to connect to the existing framework. Early in the project the desktop environment KDE and its I/O library KIO was up for discussion as a candidate for our implementation and it was considered that its API was better documented. Therefore it seems possible that some other environment could have let us focus more on the actual solution to our problem rather than trying to understand the underlying system.

Another good idea would have been to sign up for relevant mailing lists at an early stage. This could have let us have an active conversation with the developers of the libraries we were using rather than trying to understand the source code by ourselves. This was particularly problematic since the documentation for GVfs/GIO was released in the final stage of the project. It would also have been advantageous if one or more of our members had taken a course in Operating Systems, and thus could have provided some guidance regarding what to read among the wealth of related information that was found.

In summary, there are a few factors that could have made this project run more smoothly:

- Throughout the project, a more structured plan
- A simpler or better documented environment for our implementation
- More prerequisites for the participants

6 Conclusions

While the approach of modifying ACLs of files offers some of the functionality the project was aiming for, it is clear that it is neither as efficient or secure a solution as the approach of porting the operations. Modifying permissions on files is rather to be seen as a workaround of the problem than an actual solution.

While the primary ambition in the beginning indeed was to create a proxy solution, this changed due of the lack of documentation and our own knowledge concerning GIO and GVfs. Some documentation of these libraries was actually released during the final stage of this project [20]. Even though no implementation of the this approach has been made, the documentation promotes the possibilities of relaying operations.

This project has however increased the knowledge around the possibilities of creating a good implementation. For future work we would like to suggest the following guidelines:

Focus on the approach of porting operations As stated in Section 5, the ACL approach is fully viable but very ineffective. Porting operations has several advantages in comparison. It does not affect the security of the system (as long as the helper program does not have any security holes), and will only affect system performance during the time when the helper program executes.

Read the documentation on GVfs While the documentation surrounding GVfs was non-existent at the beginning of this project; developers have now begun to put one together [20]. The documentation suggests writing a custom root backend for GIO to use when performing privileged file operations. Or as stated in the documentation:

> “The right solution would be to provide proxy root backend through PolicyKit.”

A Mail conversation with Alexander Larsson

Hej Alexander!

Vi är 4 studenter på Chalmers som nu gör vårat kandidatarbete. Vi arbetar med att i GIO implementera funktionalitet för att kunna autensiera filoperationer som kräver root-privilegier genom polkit (tidigare policykit), något som du har skrivit om i din blogg<https://blogs.gnome.org/alexl/2007/11/23/file-operations-in-nautilus-gio-and-adventures-in-the-land-of-policykit/>. Då vi inte har lika bra koll på hur GIO, GLib och linux-programmering i allmänhet fungerar som du, så har vi några frågor som vi hoppas att du har tid att svara på:

1. *Har du tips på vilka filer vi bör titta närmre på? Du skriver exempelvis att implementationen bör ske i GFile eftersom alla filoperationer sker genom den. Vi undrar då om det inte skulle vara bättre att modifiera GOutputStream och GInputStream eftersom lösningen då skulle bli mer generell för alla typer av strömmar, inte bara filströmmar. Vad har du för tankar kring det?*

   GOutputStream och GInputStream är basklasser som aldrig behöver ändras, dom implementeras som subklasser av olika subsystem som man behöver strömma data till/från.

   GFile är gios version av ett "filnamn", och den har vissa operationer man kan göra på såna, t.ex att ta bort en fil med det namnet, döpa om den, få information om den eller att öppna den. Om man öppnar en fil får man en GInputStream subklass specifik för den typen av GFile. (i.e. en GLocalFileInputstream för en file:// uri, eller en GDaemonFileInputStream för en fil i gvfs.

   Det går inte att bara ta en stream vilken som helst (t.ex. en socket med tcp connection eller en GMemoryInputStream från en minnesbuffer) och "köra den som root".

2. *Har du mer djupgående tankar om hur detta ska lösas? Vi tänker använda oss av ett hjälpprogram med suid-flaggan tänd som anropar den funktion i GIO som tidigare försökts anropas tillsammans med de variabler som är relevanta, men tänker oss att det kan vara svårt att alltid veta vilka variabler som är relevanta. Vad tycker du?*

   Nä, det är inte svårt. Hela GFile API:et är designat för att kunna skicka operationer till en annan process, det är så GVfs integrationen i GIO fungerar. Jag skulle rekomendera att ni studerar både gvfs och gio först.

   Tyvärr är väl inte detta så väldigt bra dokumenterat. https://developer.gnome.org/gio/2.36/ch01.html har en introduktion till gio med en liten beskrivning av hur gvfs fungerar. Resten av gio API dokumentationen kan väl också vara bra att läsa innan man kollar på koden.

3. *Har du andra tips angående utveckling för GLib/GIO och open-source projekt i allmänhet? Vad är viktigt att tänka på om man vill få med en patch i senare utgåvor?*

   Jobba alltid med senaste versionen av koden. Hör av er till relevanta maintainers/mailing-listor tidigt för feedback (så behöver ni inte göra stora ändringar efter att ni gjort mycket jobb redan).

4. *Vi har märkt att alla GNOME-bibliotek (GLib, GIO och GObject) samt många GNOME-applikationer (exempelvis Gedit och Nautilus) är skrivna i C på ett objektorienterat vis genom att använda struct:ar som innehåller pekare till funktioner. På vilket sätt är detta att föredra gentemot att använda exempelvis C++ som mer eller mindre gör det jobbet åt en?*

   Det finns många olika orsaker, både tekniska och historisk. Generellt sett så är alla bibliotek i Gnome skrivna i C med GObject, vilket gör dem lätt att portera och lätt att binda till andra språk (speciellt med GObject Introspection). Applikationer däremot behöver inte vara i C, ofta är dom i javascript, python eller vala istället.

En annan sak är att börja planera ifrån användarsidan, inte den tekniska sidan. Dvs, utgå med att planera hur eran feature kommer se ut it nautilus (jag antar att det är huvudmålet) snarare än vilken klass som skall implementeras, sen fyller ni i behovet bakåt från UI till vad för API nautilus behöver till hur det APIt kan implementeras.

Detta påverkar mycket. T.ex. så avgör vilka features nautilus exponerar hur komplex lösningen blir. Man kan t.ex. tänka sig att vilja lösa problemet att man browsar filsystemet, hittar en fil som man vill göra en operation på (t.ex. byta namn, byta ägare, eller ta bort), men som man inte har rättigheter till. Då man får ett felmeddelande om detta skulle man lätt kunna tänka sig att det finns en knapp för ”authentisera” som bara låter en authentisera via polkit och göra om operationen. Detta är relativt lätt att implementera eftersom den höjda statusen är väldigt begränsad. Det har dock vissa svagheter, som att man inte kan se filer i kataloger man inte har läsrättigheter.

Ett annat alternativ är att man istället browsar ”som root” i ett fönster och ser allt som root ser och kan göra allt som root ser. Den typen av UI är mycket mer komplicerat eftersom att den höjda statusen nu flyter rakt in i nautilus interna strukturer. T.ex. har nautilus global cache för filinformation, som nu måste modifieras för att hantera de olika säkerhetsnivåerna som olika fönster kan vara i. Dessutom är det större säkerhetsrisk eftersom ”mer kod” har access till den förhöjda säkerhetsnivån (tom extensions kanske kommer åt den).

B Client and Server for ACL implementation

Listing B.1: Listings/org.gnome.gio.policy

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE policyconfig PUBLIC
 "-//freedesktop//DTD PolicyKit Policy Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/PolicyKit/1.0/policyconfig.dtd">
<policyconfig>

  <vendor>The GNOME Project</vendor>
  <vendor_url>http://www.gnome.org/</vendor_url>
  <icon_name>package-x-generic</icon_name>

  <action id="org.gnome.gio.write">
    <description>Write to a file (or directory)</description>
    <message>Writing to a file you don't have write access to requires authentication</message>
    <defaults>
      <allow_any>auth_admin</allow_any>
      <allow_inactive>auth_admin</allow_inactive>
      <allow_active>auth_admin_keep</allow_active>
    </defaults>
  </action>

  <action id="org.gnome.gio.read">
    <description>Read to a file (or directory)</description>
    <message>Reading a file you don't have read access to requires authentication</message>
    <defaults>
      <allow_any>auth_admin</allow_any>
      <allow_inactive>auth_admin</allow_inactive>
      <allow_active>auth_admin</allow_active>
    </defaults>
  </action>
</policyconfig>
```

Listing B.2: Listings/server.c

```
/* This file was originally based on a small code shell taken from http://www.freedesktop.org/
    software/polkit/docs/0.105/polkit-apps.html and goes under the same GNU Lesser General Public
    License */

#include "generated-code.h"
#include <gio/gdbusobject.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <polkit/polkit.h>

#define SLEEPTIME 10            //seconds until revoking privileges
#define ARGSIZE 256             //for getfacl / setfacl

/* server: [arg1] = file to change acl on; [arg2] = acl-backupfile
   (todo) compile: gcc -o server server2.c generated-code.c
   $(pkg-config --cflags --libs gio-unix-2.0 polkit-gobject-1)
*/
static gboolean do_cancel (GCancellable *cancellable);
static void check_authorization_cb (PolkitAuthority *authority,
                                    GAsyncResult    *res,
                                    GMainLoop       *loop);

static GDBusObjectManagerServer *manager = NULL;
static char* lockedFile;
static char* backupFile;
static uid_t euid, ruid;
static ExampleGio *gio;

```

```
/* changeACL is called from GIO via dbus  */
static gboolean changeACL(ExampleGio *gio2,
        GDBusMethodInvocation *invocation,
        gchar* pathname)
{

  euid = geteuid();
  ruid = getuid();
  seteuid(ruid);      //Revoke privileges, reinstate if polkit says ok
  pid_t parent_pid;
  const gchar *action_id;
  GMainLoop *loop;
  PolkitSubject *subject;
  PolkitAuthority *authority;
  GCancellable *cancellable;
  gboolean isRunning = FALSE;

  /* First check if arguments actually are files */
  lockedFile = pathname;
  backupFile = "/home/alles/backup.txt"; //can be changed to use argv
  gio = gio2;
  //access with F_OK returns true if the file doesn't exist
  if (access(lockedFile,F_OK) || access(backupFile,F_OK))
    {
      goto signalgio;
    }
  g_printf("GIO wants to modify [%s]\n",pathname);
  action_id = "org.gnome.gio.read"; //found in /usr/share/polkit-1/actions/
  loop = g_main_loop_new (NULL, isRunning);
  isRunning = TRUE;

  cancellable = g_cancellable_new ();

  authority = polkit_authority_get_sync(cancellable,NULL);


  parent_pid = getppid ();
  if (parent_pid == 1)
    {
      g_printerr ("Parent process was reaped by init(1)\n");
      return 1;
    }
  subject = polkit_unix_process_new (parent_pid);
  g_print ("Will cancel authorization check in 10 seconds\n");

  /* Set up a 10 second timer to cancel the check */
  g_timeout_add (10 * 1000,
                 (GSourceFunc) do_cancel,
                 cancellable);

  polkit_authority_check_authorization (authority,
                  subject,
                  action_id,
                NULL,
        POLKIT_CHECK_AUTHORIZATION_FLAGS_ALLOW_USER_INTERACTION,
        cancellable,
        (GAsyncReadyCallback) check_authorization_cb,
        loop);

  g_main_loop_run (loop);



 signalgio:
  g_printf("Signaling gio OK\n");
  example_gio_set_done_with_acl(gio, !example_gio_get_done_with_acl(gio));
  example_gio_complete_change_acl(gio,invocation);

 out:
  g_printf("Returning TRUE (ending sync call)\n");
  seteuid(ruid);
```

```c
  g_object_unref (authority);
  g_object_unref (subject);
  g_object_unref (cancellable);
  g_main_loop_unref (loop);

  return TRUE; //indicate that the method was handled
}

static void
on_bus_acquired (GDBusConnection *connection,
                 const gchar     *name,
                 gpointer         user_data)
{
  g_printf ("Acquired a message bus connection\n");
  GDBusObjectSkeleton *object;
  guint n;
  ExampleGio *gio;

  /* Create a new Test.ObjectManager rooted at /example/GIO */
  manager = g_dbus_object_manager_server_new ("/example/GIO");

  /* Create a new D-Bus object at the path /example/GIO/g */
  object = g_dbus_object_skeleton_new ("/example/GIO/g");

  /* Make the newly created object export the interface
   * org.gtk.GDBus.Example.ObjectManager.Gio (note
   * that @object takes its own reference to @gio).
   */

  gio = example_gio_skeleton_new ();
  example_gio_set_done_with_acl(gio,FALSE);
  g_dbus_object_skeleton_add_interface (object, G_DBUS_INTERFACE_SKELETON (gio));
  g_object_unref (gio);

  /* Handle changeACL() D-Bus method invocations on the .GIO interface */
  g_signal_connect (gio,
                    "handle-change-acl",
                    G_CALLBACK (changeACL),
                    NULL); /* user_data */

  /* Export the object (@manager takes its own reference to @object) */
  g_dbus_object_manager_server_export (manager, object);
  g_object_unref (object);

  /* Export all objects */
  g_dbus_object_manager_server_set_connection (manager, connection);

}

static void
on_name_acquired (GDBusConnection *connection,
                  const gchar     *name,
                  gpointer         user_data)
{
  g_print ("Acquired the name %s\n", name);
}

static void
on_name_lost (GDBusConnection *connection,
              const gchar     *name,
              gpointer         user_data)
{
  g_print ("Lost the name %s\n", name);
}

gint main (gint argc, gchar** argv)
{
  GMainLoop *mloop;
  guint id;

  mloop = g_main_loop_new (NULL, FALSE);
  id = g_bus_own_name (G_BUS_TYPE_SESSION,
                       "Test.ObjectManager",
                       G_BUS_NAME_OWNER_FLAGS_ALLOW_REPLACEMENT |
                       G_BUS_NAME_OWNER_FLAGS_REPLACE,
                       on_bus_acquired,
                       on_name_acquired,
                       on_name_lost,
                       mloop,
                       NULL);

  g_main_loop_run (mloop);

  g_bus_unown_name (id);
  g_main_loop_unref (mloop);

  return 0;

}
/* Authorization handling */
static void check_authorization_cb (PolkitAuthority *authority,
                                    GAsyncResult    *res,
                                    GMainLoop       *loop)
{
  GError *error;
  PolkitAuthorizationResult *result;
  error = NULL;
  result = polkit_authority_check_authorization_finish (authority, res, &error);
  if (error != NULL)
    {
      g_print ("Error checking authorization: %s\n", error->message);
      g_error_free (error);
    }
  else
    {
      const gchar *result_str;
      if (polkit_authorization_result_get_is_authorized (result))
        {
          result_str = "Authorized";
          g_printf("Adding permissions\n");
          pid_t child1,child2;
          char ruidAsString[16] = "";
          sprintf(ruidAsString, "%d", ruid);

          /* build the argument for getfacl >> backup-procedure */
          char* getfacArgs[] = {"getfacl","--absolute-names",lockedFile,NULL};
          char restoreString[ARGSIZE] = "--restore=";
          strcat(restoreString,backupFile);
          char* restoreArgs[] = {"setfacl", restoreString,NULL};

          /* Build the permission argument for setfacl: */
          char sargs[ARGSIZE] = "u:";
          strcat(sargs,ruidAsString);
          strcat(sargs,":r "); // todo:change to switch case rwx
          char* setfacArgs[] = {"setfacl","-m",sargs,lockedFile,NULL}; //should now be "setfacl -m
    u:####:r t.txt"

          //Fork-jungle begins here. #1
          seteuid(euid); //NOW ROOT

          child1 = vfork(); //vfork == parent waits for child to finish
          if(child1 > 0) //first parent #5 (original process)
            {
              //I should restore the acl according to backup
              int lastFork = fork();
              if(lastFork == 0) //executes simultaneusly as the last piece of code (ending
    procedure)
                {
                  sleep(SLEEPTIME);
                  g_printf("%d seconds has passed, restoring ACL according to backup\n",SLEEPTIME);
                  execv("/usr/bin/setfacl",restoreArgs);
                }
            }
          else //first child = #2
            {
              child2 = vfork();

              if(child2 > 0) //Second parent = #4
                {
                  //I should modify the acl to add permissions
                  execv("/usr/bin/setfacl",setfacArgs);
                  //setfacl-process now killed, goto #5
                }
              else //second child = #3
                {
                  //I should redirect stdout and make a backup of the acl
                  int newOut;
                  //      int bak; //unnecessary in this case to backup stdout
                  fflush(stdout);
                  //      bak = dup(1); // -||-
                  newOut = open("/home/alles/backup.txt",O_WRONLY|O_APPEND);
                  dup2(newOut,1);
                  close(newOut);
                  execv("/usr/bin/getfacl",getfacArgs); //outout >> backup
                  /*can restore stdout here if necessary with bak:
                    fflush(stdout);
                    dup2(bak, 1);
                    close(bak);
                  */
                }
              _exit(2); //should never occur(both processes die with execv)
            }
        } //end of authorized code

      else if (polkit_authorization_result_get_is_challenge (result))
        {
          result_str = "challenge";
        }
      else
        {
          result_str = "not authorized";
        }

      g_print ("Authorization result: %s\n", result_str);
    } //end of authorization check

  g_main_loop_quit (loop);
}

static gboolean do_cancel (GCancellable *cancellable)
{
  //  g_print ("Timer has expired; cancelling authorization check\n");
  //  g_cancellable_cancel (cancellable);
  return FALSE;
}
```

Listing B.3: Listings/client.c

```
#include "generated-code.h"

/* This function is run when server's change_ACL() has been executed.
 */

static void
on_interface_proxy_properties_changed (GDBusObjectManagerClient *manager,
                                       GDBusObjectProxy         *object_proxy,
                                       GDBusProxy               *interface_proxy,
                                       GVariant                 *changed_properties,
                                       const gchar *const       *invalidated_properties,
                                       gpointer                  user_data)
{
  GVariantIter iter;
  const gchar *key;
  GVariant *value;
  gchar *s;

  //  g_print ("Properties Changed on %s\n", g_dbus_object_get_object_path (G_DBUS_OBJECT (
    object_proxy)));
  g_variant_iter_init (&iter, changed_properties);
  while (g_variant_iter_next (&iter, "{&sv}", &key, &value))
    {
      s = g_variant_print (value, TRUE);
      g_print ("  %s triggered\nGIO can now continue processing\n", key);
      g_variant_unref (value);
      g_free (s);
    }
  _exit(2);
}

gint main (gint argc, gchar *argv[])
{
  GDBusObjectManager *manager;
  GMainLoop *loop;
  GError *error;
  ExampleGio *proxy;
  if(argc < 2) {
    g_printf("Usage: %s file\n",argv[0]);
    return 1;
  }
  char* filepath = argv[1];
  manager = NULL;
  loop = NULL;
  //  g_type_init (); //obsolete in 2.36
  loop = g_main_loop_new (NULL, FALSE);
  error = NULL;
  manager = example_object_manager_client_new_for_bus_sync (G_BUS_TYPE_SESSION,
                    G_DBUS_OBJECT_MANAGER_CLIENT_FLAGS_NONE,
                                                                      "Test.ObjectManager",
                                                                      "/example/GIO",
                                                                      NULL, /* GCancellable */
                                                                      &error);

  if (manager == NULL)
    {
      g_printerr ("Error getting object manager client: %s", error->message);
      g_error_free (error);
      goto out;
    }

  proxy = example_gio_proxy_new_for_bus_sync(G_BUS_TYPE_SESSION,
                                             G_DBUS_PROXY_FLAGS_NONE,
                                             "Test.ObjectManager", //busname
                                             "/example/GIO/g",     //object
                                             NULL,                 //cancel.
                                             &error);

  if (proxy == NULL)
    {
      g_printerr ("Error creating proxy: %s\n", error->message);
      g_error_free (error);
      goto out;
    }

  g_signal_connect (manager,
                    "interface-proxy-properties-changed",
                    G_CALLBACK (on_interface_proxy_properties_changed),
                    NULL);

  /* methodcall to helper */
  example_gio_call_change_acl_sync(proxy,filepath,NULL,&error);
  g_main_loop_run (loop);

 out:
```

```c
  if (manager != NULL)
    g_object_unref (manager);
  if (proxy != NULL)
    g_object_unref (proxy);
  if (loop != NULL)
    g_main_loop_unref (loop);

  return 0;
}
```

References

[1] B. Still, *Handbook of research on Open Source Software.* 2007.

[2] M. Bishop. (2002). How attackers break programs, [Online]. Available: http://nob.cs.ucdavis.edu/bishop/secprog/sans2002.pdf (visited on 05/14/2013).

[3] 2013. [Online]. Available: http://distrowatch.com/dwres.php?resource=popularity.

[4] K. Cook. (2013). Linux kernel source tree - git, [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=800179c9b8a1e796e441674776d11cd4c05d61d7.

[5] A. Gruenbacher, "Posix access control lists on linux", SuSE Labs, SuSE Linux AG Nuremberg, Germany, Tech. Rep., 2003. [Online]. Available: http://users.suse.com/~agruen/acl/linux-acls/online/.

[6] D. MacKenzie and J. Meyering. (2010). Chmod(1) - linux man page, [Online]. Available: http://linux.die.net/man/1/chmod (visited on 05/14/2013).

[7] D. Rusling, *The Linux Kernel.* 1999, ch. 3. [Online]. Available: http://www.tldp.org/LDP/tlk/mm/memory.html (visited on 04/29/2013).

[8] *The gnu c library*, 2013, ch. 29. [Online]. Available: http://www.gnu.org/software/libc/manual/html_mono/libc.html#Users-and-Groups.

[9] H. Chen, D. Wagner, and D. Dean, "Setuid demystified", *Proc. 11th USENIX Security Symp.*, pp. 171–190, Aug. 2002. [Online]. Available: http://www.cs.ucdavis.edu/~hchen/paper/usenix02.pdf.

[10] J. Mellander, "Unix filesystem security", *Inform. Security Tech. Report*, vol. 7, no. 1, pp. 11–25, Mar. 2002. [Online]. Available: http://hdl.handle.net/2060/19900067423.

[11] M. Bishop, "How to write a setuid program", Research Institute for Advanced Comput. Science, NASA Ames Research Center, Moffett Field, CA 94035, Tech. Rep., 1985. [Online]. Available: http://hdl.handle.net/2060/19900067423.

[12] M. Zalewski, "Delivering signals for fun and profit", BindView Corporation, Tech. Rep., May 2001. [Online]. Available: http://lcamtuf.coredump.cx/signals.txt.

[13] *Dbus*, 2012. [Online]. Available: http://www.freedesktop.org/wiki/Software/dbus (visited on 04/29/2013).

[14] *Glib reference manual*, 2013. [Online]. Available: https://developer.gnome.org/glib/.

[15] A. Krause, *Found. of GTK+ development.* 2007.

[16] (2013). Tiobe programming community index for may 2013, [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (visited on 04/23/2013).

[17] T. G. Project, *Gobject reference manual*, 2012. [Online]. Available: https://developer.gnome.org/gobject/stable/pr01.html (visited on 05/13/2013).

[18] *Gio reference manual*, 2012, ch. GFile. [Online]. Available: https://developer.gnome.org/gio/2.32/GFile.html (visited on 04/23/2013).

[19] *Gio reference manual*, 2012, ch. GFileInfo. [Online]. Available: https://developer.gnome.org/gio/2.32/GFileInfo.html (visited on 04/23/2013).

[20] T. Bzatek and A. Larsson. (May 6, 2013). Gvfs documentation, [Online]. Available: https://live.gnome.org/gvfs/doc (visited on 05/14/2013).

[21] *Gio reference manual*, 2012, ch. gdbus-codegen. [Online]. Available: https://developer.gnome.org/gio/2.36/gdbus-codegen.html (visited on 05/17/2013).

[22] H. Pennington, A. Carlsson, A. Larsson, S. Herzberg, S. McVittie, and D. Zeuthen, *D-bus specification*, 2013. [Online]. Available: http://dbus.freedesktop.org/doc/dbus-specification.html#introspection-format (visited on 05/17/2013).