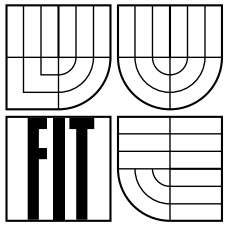
Brno University of Technology

Faculty of Information Technology



Thesis

Callback Framework for VFS layer

2005 **František Hrbata**

Statement

I hereby state that I have created this thesis on my own and under the supervision of Ing. Tomas Kasparek. I have listed all infomation sources which I used.

Brno November 14, 2005 .....

Acknowledgements

I would like to thank GRISOFT, s.r.o. and especially Ing. Obluk Karel, Ph.D. for giving me the opportunity to create this thesis. I would also like to thank my supervisor Ing. Kasparek Tomas and all of the following people for their help and support.
Ing. Suchanek Jaroslav, Ing. Hemzal Jiri and Mgr. Bracek Jiri, Harper-Dolejsek Michael.

Abstract

This thesis focuses on filesystem access control in the Linux kernel. It describes all existing methods of filesystem access control and points out their problems and limitations. Further it describes in detail Virtual Filesystem Switch (VFS) implementation in the Linux kernel and introduces a new framework called Redirfs. It is based on the redirection of the VFS objects operations and allows for third-party Linux kernel modules, which are called filters, to be notified about all events in the VFS layer.

Keywords

VFS, framework, filesystem, LSM, RSBAC, Dazuko, Linux, kernel, LKM, filter, avgflt, inode, dentry, super block, vfsmount, cache, FiST, overlay filesystem, buffer cache, dentry cache, inode cache, Redirfs, process, Linux kernel linked lists, fs_struct, syscall table, interface, user space, kernel space.

Contents

List of Figures

Chapter 1

Prologue

Filesystem access control is an important part of operating system security. The Linux kernel provides default filesystem access control based on rights (read, write, execute) which are divided for owner, group and others. In some cases this is not enough and access control needs to be extended. As an example, we can consider on–access scanning by an antivirus application. The antivirus application needs to be notified when other processes are trying to open, execute or close some files. This is not possible with the standard filesystem access control mechanism. Another example can be problems with root's rights. By default there are no limitations for root. In some cases it is desirable to limit the root's rights. For example when a process need to be executed with root privileges. This is also not possible with the standard Linux access control mechanism. Several projects like RSBAC, Medusa or LSM provide enhanced access control frameworks for Linux kernels. This thesis focuses on filesystem access control, describes the existing methods of filesystem access control, and points out their problems and limitations. Further it describes in detail the Linux implementation of the Virtual Filesystem Switch (VFS) and introduces a new Redirfs framework based on it. Redirfs aims to be a general framework which allows third party modules to be notified about VFS events. Redirfs tries to overcome the problems and limitations of existing solutions. This thesis uses filesystem access control analysis from semestral project.

Chapter 2 describes in detail the Virtual Filesystem Switch and its implementation in the Linux kernel. It also provides information about all objects which form the VFS, and describes their data fields and operations. Further it focus on VFS objects interaction and interconnection.

Chapter 3 presents the Linux Security Module framework which was included in the Linux kernel 2.6. It describes its interface, modules stacking, and points out its flawed design.

Chapter 4 describes overlay filesystems and the way they are implemented for the Linux kernels. It presents the FiST project which provides an overlay filesystem generator.

Chapter 5 presents Dazuko which is aimed to be a cross-platform device driver that allows applications to control file access on a system. It describes several ways in which Dazuko interacts with the Linux kernel.

Chapter 6 introduces the Redirfs framework which creates a new layer between the VFS layer and the native filesystems. It replaces the VFS object's operations and allows third-party modules to be notified about VFS events. It describes its design, architecture, implementation, and interface.

Chapter 7 contains sample code of the dummyflt which uses the Redirfs framework.

Chapter 8 describes implementation of the avgflt. This is a filter using the Redirfs framework, which communicates with the AVG7 Anti-Virus daemon and provides on-access scanning.

Chapter 2

The Virtual Filesystem Switch

The Virtual Filesystem Switch, or VFS was first introduced by Sun Microsystems in 1986 and implemented in the SunOS. Since then all other Unix based operating systems have adopted the VFS concept. The VFS provides an abstract software layer that allows the operating system kernel to call functions of different filesystems without having to know anything about the specific filesystem type. The following text will describe VFS implementation in the Linux kernel, with a view to the last Linux kernel version 2.6.11.3 available at the time of writing this thesis.

In the beginning the Linux kernel only supported the Minix filesystem. However its design restrictions were too limiting, so it was necessary to implement a new filesystem, which was then called Extended File System (ExtFS). The new ExtFS had to coexist with Minix and that was why the first VFS implementation was included in the Linux kernel. Initially, implementation of the Linux VFS was written by Chris Provenzano. It was later rewritten by Linus Torvalds and included along with the ExtFS into the Linux kernel 0.96c in April 1992. Thanks to the VFS a lot of other filesystems were written or ported and included in the Linux kernel, which now, in comparison with other Unix type systems, supports most filesystems. Filesystems supported by the Linux kernel can be divided into three groups.

Disk-based Filesystems

These are the base filesystems, which manage data stored on hard drives.

- Filesystems written primary for Linux such as Second Extended Filesystem (Ext2), Third Extended File System (Ext3) and the Reiser Filesystem (ReiserFS).
- Filesystems for other Unix type systems such as SYSV filesystem (System V, Coherent, XENIX), UFS (BSD, Solaris, Next) and VERITAS VxFS (SCO UnixWare).
- ISO9660 CD-ROM Filesystem and Universal Disk Format (UDF) DVD file system.
- Other proprietary Filesystems such as HPFS (IBM's OS/2), HFS (Apple's Macintosh), Amiga's Fast Filesystem (AFFS) and Acorn Disk Filing System (ADFS).
- Other journaling filesystems originating in systems other than Linux such as JFS (IBM) and XFS (SGI).

Network Filesystems

These filesystems allows access to other filesystems located on other computer's hard drives via the network. The most known are NFS, Coda, Andrew Filesystem (AFS), Server Message Block (SMB) and Novell's NetWare Core Protocol (NCP).

Special Filesystems

These filesystems are not stored on hard drives, but are created dynamically. Typical examples are /proc filesystem or /sys filesystem.

2.1 VFS Role

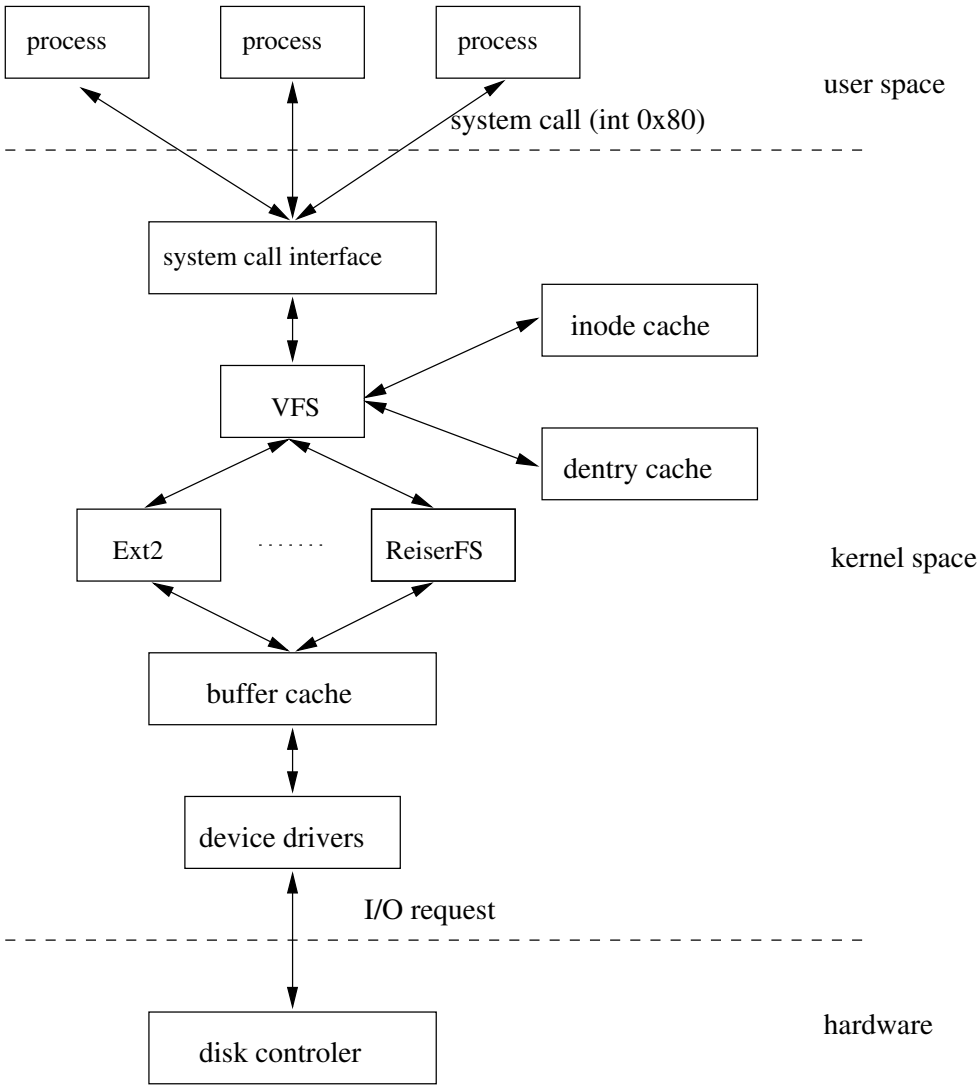


Figure 2.1: The VFS role during system calls

The VFS provides a universal method for accessing different filesystems. It consist of several objects, which will be described in detail later in this chapter. The file system driver has to be able to convert physical representation of the filesystem stored on disk into VFS objects and vice versa. This concept warrants a uniform interface for all filesystems as is shown in figure 2.1.

All system calls relating to the filesystems are directed to the VFS. All processes in the user space use the same interface for all filesystems as is shown in figure 2.1. The differences between specific filesystems are handled by the VFS. In fact the VFS hides all specific filesystems and presents them to the user space processes as one.

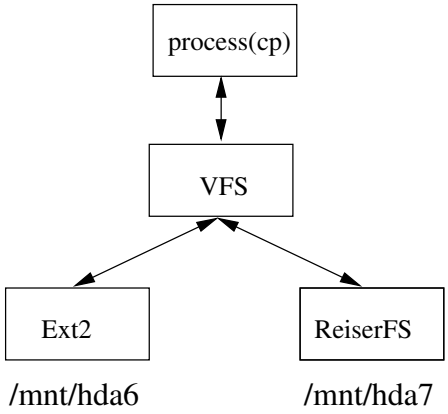


Figure 2.2: The VFS role in a simple copy file operation

Figure 2.2 shows a simple copy file operation. We will consider the following command, which copies the `test` file from `/mnt/hda6` to the `/mnt/hda7` directory.

```
$ cp /mnt/hda6/test /mnt/hda7/test
```

It is obvious, that the `cp` command doesn't know about the filesystem mounted on `/mnt/hda6` or `/mnt/hda7` directory. Read, write and other operations used by cp command are handled by the VFS and directed to the specific file system.

2.2 VFS Data Structures

The previous section has shown the VFS role. This section will describe in detail all objects forming the Linux VFS and their main data fields with focus on the interaction between the VFS objects.

The VFS design is object oriented, even if it is written in pure C code. The main objects are super block, inode, dentry and file. All these objects are represented by structure, which aside from data fields contains a pointer to the structure with pointers to the functions. This structure of pointers is filled by the filesystem driver, so that every object created contains pointers to the functions of the specific filesystem. That is how the VFS knows, which specific filesystem functions to call in the simple copy example in figure 2.2.

All file paths mentioned in the following text are relative to the root directory of the Linux source tree.

2.2.1 Linux Kernel Linked Lists

For the following text, which will describe the VFS data structures, it is important to understand how Linux kernel linked lists work.

The Linux kernel uses a lot of linked lists (hundreds), so it would be a big waste to create new structures for each list and implement new operations (add, delete, initialization etc.). Linux kernel defines two generic linked lists to prevent this waste. Both lists are double linked and defined in `include/linux/list.h`. The first is defined by the structure `list_head` and the second by the structures `hlist_head` and `hlist_node`.

The list based on the `list_head` structure is circular. The second list isn't circular, has separate list head from list nodes and is primarily used for hash tables. We will focus only on lists based on the `list_head` structure because the principles are the same for both. The `list_head` structure contains only two pointers `next` and `prev`. Both pointers point again to the `list_head` structure. In figure 2.3 you can see an example of how the aux structure is linked. The usage is very simple, all that has to be done is to add a `list_head` field to the structure and then use functions and macros for list manipulation, which are also defined in `include/linux/list.h`

```
struct list_head {
    struct list_head *next, *prev;
};
```

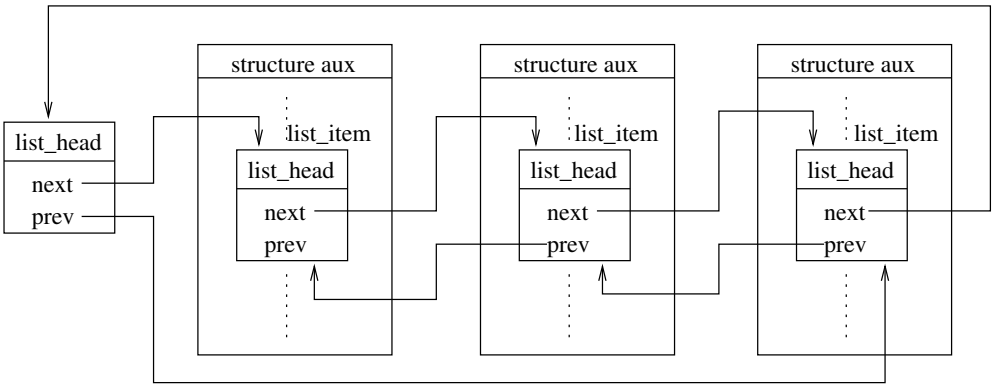


Figure 2.3: List created by list_head structures

The list is linked through the `list_head` structure placed in the `aux` structure. Now it is time to explain how the pointer to the aux structure is acquired. The best way will be to describe the `list_entry(ptr, type, member)` macro, which does this. The `ptr` argument is a pointer to the `list_head` structure inside the `aux` structure, (which in our example is a pointer to the `list_item` variable), `type` is the name of the structure, (which in our example is `struct aux`), and `member` is a name of the `list_head` structure, (which in our example is `list_item`). Macro `list_entry` counts the address of the `aux` structure and returns its pointer.

The base command (expression) in `list_entry` is `&((type*)0)->member` defined in `include/linux/stddef.h`, which returns the offset from `member` address to the requested structure start address. Now, when the offset is known, it is very easy to count the address of `aux` structure.

2.2.2 Superblock Object

The superblock structure `struct super_blok` is defined in `include/linux/fs.h` and is created for every mounted filesystem. It presents an entry point to the filesystem. All superblock objects are linked through `struct list_head s_list` field. List header `struct list_head super_blocks` is defined in `fs/super.c`. The superblock contains some metadata information like block size, maximum size of file, mount flags and others. It is partially filled by VFS and the rest of the fields have to be filled by a specific file system driver. The superblock contains the `struct dentry *s_root` field, which represents the root dentry object for the filesystem. The dentry object is described in section 2.2.6. The field `struct file_system_type *s_type` is a pointer to the object, which contains information about the filesystem (for example filesystem name, which is used as a argument to the mount command). The filesystem type object is described in section 2.2.9. The field `struct list_head s_dirty` is the list header of modified inodes. It is used when a filesystem is unmounted and all dirty inodes have to be flushed to the disk. The field `struct list_head s_files` is the list header of all file objects opened on the filesystem. The file object is described in section 2.2.8. The pointer to superblock operations is stored in `struct super_operations *s_op` field and has to be filled by the file system driver. Operations mostly convert VFS's inode to the disk inode and vice versa (`read_inode, write_inode`).

2.2.3 Inode Object

All information needed by the VFS about a file are stored in a data structure called `struct inode`, defined in `include/linux/fs.h`. The VFS's inode is different to the inode stored on disk. The inode in VFS is created by VFS and duplicates some information from the inode stored on disk to reduce slow access to the hard disk. In some operating systems (*BSD) the VFS's inodes are called vnodes (virtual inode) to avoid confusion with inodes stored on disk. In the following text all inode information relates to the inode in the VFS.

The inode structure can represent a regular file, directory, character device, block device, fifo, symlink, or unix sockets. The field `umode_t i_mode` is used to distinguish the file type and also contains access rights to the inode. Every inode is identified by an inode number stored in the field `unsigned long i_ino` which is unique within the scope of one filesystem. The inode structure doesn't contain the file name. That is because it can have several file names (hardlinks). The file name is stored in the dentry object, which will be described in section 2.2.6. The inode contains a list of all file names (dentries) assigned to it and the head of the list is stored in `struct list_head i_dentry`. The number of hardlinks pointing to an inode is in the field `unsigned int i_nlink`. The inode can be in several states. The state is stored in the `unsigned long i_state` field and can contain the following inode state bits. `I_DIRTY_SYNC, I_DIRTY_DATASYNC, I_DIRTY_PAGES` or `I_DIRTY`, which defines all three previous and indicates that the inode

is dirty and the corresponding disk inode has to be updated. `I_FREE` indicates that the inode is being freed, `I_CLEAR` means that the inode content is no longer meaningful, `I_NEW` means that inode is newly created and its data has to be filled by the specific filesystem and finally `I_LOCK` means that the inode is involved in I/O transfer. Every inode has a usage counter `atomic_t i_count`, which indicates that the inode is used by some process. Each inode always appears in one of the following lists. List `struct list_head inode_unused` of valid unused inodes with `i_count` set to zero, defined in `fs/inode.c`. List `struct list_head inode_in_use` of valid in-use inodes with positive `i_count` and `i_nlink` fields, defined in `fs/inode.c` and list of dirty inodes with positive `i_count` and `i_nlink` fields, but also with dirty flag set in `i_state` field. The field `struct list_head i_list` is used to link inode in one of the mentioned lists. Inodes are placed in an inode cache, which reduces access to the disk. VFS uses a hash table to speed up inode searching. The field `struct list_head i_hash` is used to link inodes with the same hash value. More about the inode cache can be found in section 2.3.2.

The inode contains inode and file operations. The pointer to inode operations is stored in the `struct inode_operations *i_op` field and the pointer to file operations is stored in the `struct file_operations *i_fop`. Both operations are defined in `include/linux/fs.h`. Note that Linux has seven types of file and every type has its own operations. It means, that inode operations for regular files will be different to inode operations for the directory. Also not all operations have to be set. For example, only a few file operations will be set for the directory. The rest will be set to NULL, which means that these operations are not supported. VFS implements some generic operations. For example if the filesystem doesn't have permission operation, VFS will call a generic permission function.

The following text will describe the main inode and file functions.

2.2.4 Inode Operations

```
int create(struct inode *dir, struct dentry *dentry, int mode,
struct nameidata *nd)
```

Creates a new disk inode for a regular file. Argument `dir` is a pointer to the parent inode, which is an inode representing the directory, `dentry` is a new dentry object to which the new inode will be attached, `mode` contains the inode type and its access rights (S_IFREG, S_IRWXU, etc.), `nd` is the result from path lookup.

```
struct dentry *lookup(struct inode *dir,struct dentry *dentry,
struct nameidata *nd)
```

This function is called when the searched dentry (name) object was not found in the dentry cache. Argument `dir` is the inode object of parent, `dentry` is a newly created dentry object, that has to be filled by a specific filesystem and `nd` contains the dentry object which was last found in the dentry cache.

```
int link(struct dentry *old,struct inode *dir, struct dentry *new)
```

Creates new hard link that refers to the inode connected to the `old` dentry. The name of the new link is in the `new` dentry and `dir` is the directory inode. The hard link can refer

only to the regular file. Hard link and its regular file have to be on the same filesystem. This is because a new dentry object is only created for the hard link. The inode object is the same for both, only is incremented the `i_nlink` counter.

```
int unlink(struct inode *inode, struct dentry *dentry)
```

Removes the hard link of the file specified by `dentry`. The argument `inode` is the parent inode.

```
int symlink(struct inode *dir, struct dentry *dentry, const char *path)
```

Creates a new symbolic link. The argument `dir` is the parent inode, `dentry` is the newly created dentry object for the symlink and `path` is the path where the symlink will point to. Symlink can be created for every type of file and can point across file systems. Filesystem driver creates a new disk inode for symlink and saves the path into it. When the Linux kernel does a path lookup and finds that an actual inode is the symlink, it reads the path stored in the disk inode and redirects the following path lookup to the new path.

```
int mkdir(struct inode *dir, struct dentry *dentry, int mode)
```

Creates a new directory inode. The argument `dir` is the inode of the parent, `dentry` is a newly created dentry object with the name for the new directory. Specific filesystem will create a new directory disk inode and bind it to the dentry object. The argument `mode` contains access rights to the new directory.

```
int rmdir(struct inode *,struct dentry *)
```

Removes the directory specified by `dentry` object. Argument `dir` is the parent inode.

```
int mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t rdev)
```

Creates a new disk inode for a special file. The argument `dir` is the parent inode, `dentry` is a newly created dentry object to which the new inode for the special file will be bound, `mode` contains access rights and file type for the new inode and `rdev` is a device's major number.

```
int rename(struct inode *old_dir, struct dentry *old_dentry,
struct inode *new_dir, struct dentry *new_dentry)
```

Argument `old_dir` is the directory inode from which the file represented by `old_dentry` will be removed, `new_dir` is the new directory inode and `new_dentry` contains a new file name.

```
int readlink(struct dentry *dentry, char __user *buffer, int buflen)
```

Copies into the memory area specified by `buffer` the file pathname corresponding to the symbolic link specified by the `dentry`.

`int follow_link(struct dentry *dentry, struct nameidata *nd)`

Translates a symbolic link specified by a `dentry` object to the `nd` structure which is then used for the following path lookup.

`void truncate(struct inode *inode)`

Modifies the size of the file associated with the `inode`.

`int permission(struct inode *inode, int mask, struct nameidata *nd)`

Checks whether the specified access mode defined by `mask` is allowed for the inode. The `mask` argument contains access type (MAY_EXEC, MAY_WRITE, etc.)

2.2.5 File Operations

`loff_t llseek(struct file *file, loff_t offset, int origin)`

Updates the file position. The `file` argument is the object that represents the opened file. The file object is described in section 2.2.8. The argument `offset` is the new file position in the file and `origin` designates from which part of the file the `offset` will be set (0 – the new position is set to the `offset` value from file start position, 1 – the new file position is set to its current location plus `offset`, 2 – the new file position is set to the size of the file plus `offset`).

`ssize_t read(struct file *file, char __user *buf, size_t count, loff_t*offset)`

Reads `count` bytes from the `file` starting at position `offset` to the `buffer`.

`ssize_t write(struct file *, const char __user *, size_t, loff_t *)`

Writes `count` bytes from the `buffer` to the `file` on the `offset` position.

`int readdir(struct file *dir, void *dirent, filldir_t filldir)`

Returns the next directory entry of the `dir` in `dirent`. The argument `filldir` contains the address of an auxiliary function that extracts the fields in a directory entry.

`unsigned int poll(struct file *file, struct poll_table_struct *table)`

Checks whether there is activity on the `file` and goes to sleep until something happens to it. The argument `table` contains the type of activity that happened to the `file`.

`int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long args)`

Sends the command `cmd` with `args` to an underlying hardware device specified by `inode` and `file`.

`int mmap(struct file *file, struct vm_area_struct *vma)`

Performs a memory mapping of the `file` into a process address space specified by `vma`.

`int open(struct inode *, struct file *file)`

VFS calls this function after the `file` with `inode` is opened or created. Most filesystems use the `generic_file_open` function provided by the VFS. It disallows the opening of large files on 32bit systems the `O_LARGEFILE` wasn't specified.

`int flush(struct file *file)`

Called when a reference to the `file` is closed. That is when the `f_count` field of the file object is decremented.

`int release(struct inode *inode, struct file *file)`

Called when the last reference to the `file` object with the `inode` is closed. That is when `f_count` field is zero.

`int fsync(struct file *file, struct dentry *dentry, int datasync)`

Writes all cached data of the `file` with `dentry` to the disk. If argument `datasync` is set the cached data will be flushed only if the inode has set the I_DIRTY_DATASYNC flag.

2.2.6 Dentry Object

The dentry object is represented by the `struct dentry` structure, which is defined in `include/linux/dcache.h`. Dentries are used for file path lookup and are common for all specific filesystems. It means that VFS creates a common layer over all filesystems for file path lookup. Dentries are only part of the VFS and specific filesystems have no corresponding image of the dentry on disk. The dentry object is created for every component of a path name that a process looks up. For example, we will consider `/mnt/hda8/file` file path. The dentry object is created for `/`, `mnt`, `hda8` and `file` component. It is obvious, that dentry objects aren't created only for directories, but also for files (every file type). All dentry objects are linked in an n-ary tree as is shown in figure 2.4. Field `struct list_head d_subdirs` contains a header of list, which links all entries (subdirectories, files, char devices, etc.) in the directory. The `d_subdirs` is set only if the dentry object represents the directory because no other file type can have subdirectories or contain other files. Note that directories in the VFS are considered as files that contain other subdirectories and files. Entries of directory dentry are linked through `struct list_head d_child` field. The `d_child` name for this field can be a little confusing, because dentries are linked through `d_child` on the same level. Perhaps sibling would be a better name. Field `struct dentry* d_parent` contains pointer to the dentry parent object. The dentry name is stored in the `struct qstr d_name` field. The structure `qstr` contains, in addition the dentry name, also the name length and hash value of the dentry name. The hash value is used to speed up comparison of dentry names. Dentry also contains a pointer `struct inode *d_inode` to the inode object. As was written in section 2.2.3 the inode

doesn't contains the file name because it can have several filenames. The name is stored in a dentry object and several dentry objects can point to the same inode object. Dentry objects pointing to the same inode are linked through the `struct list_head alias` field and list header is stored in the corresponding inode object. Dentry objects are stored in the dentry cache. The field `struct hlist_node d_hash` links the dentry object with the same hash value (dentry cache as well as inode cache is not collision free). Dentry cache is described in section 2.3.1. Field `d_count` is the dentry usage counter.

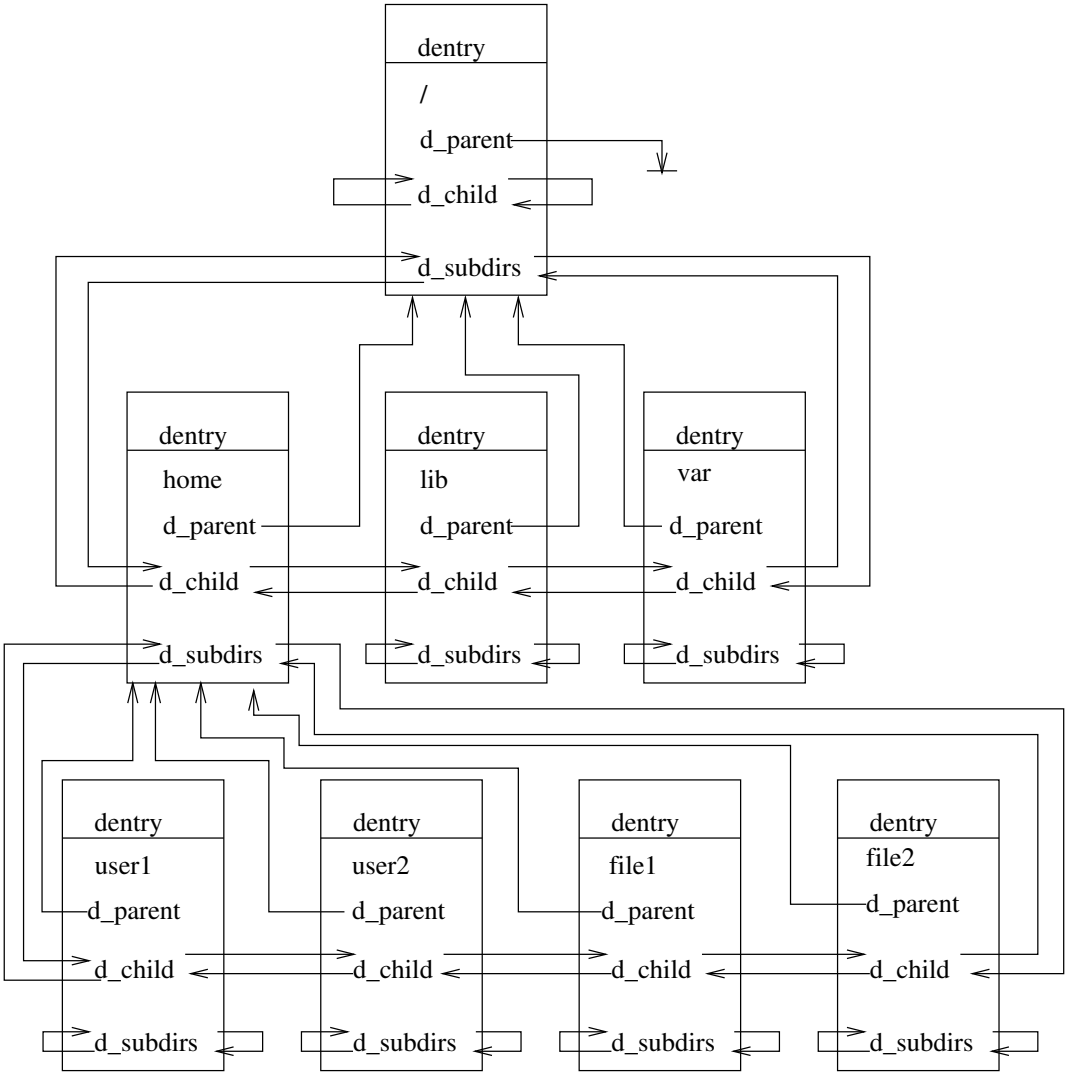


Figure 2.4: N-ary dentry objects tree

The dentry object may be in one of four states.

free – The dentry object contains no valid information and is not used by the VFS. The corresponding memory area is handled by the slab allocator (see section 2.3).

unused – The dentry object is not currently used by the kernel. The `d_count` usage counter of the object is zero, but the `d_inode` field still points to the associated inode. The dentry object contains valid information, but should be discarded if necessary in order to reclaim memory.

in-use – The dentry object is currently used by the kernel. The `d_count` usage counter is positive and the `d_inode` field points to the associated inode object. The dentry object contains valid information and cannot be discarded.

negative – The inode associated with the dentry object doesn't exist and the `d_inode` field is set to NULL. This could happened if the disk inode has been deleted or a non-existing path has been resolved.

All unused dentry objects are linked in the LRU (Last Recently Used) list and can be freed in order to reclaim memory. The header of the LRU list `struct list_head dentry_unused` is in `fs/dcache.c`. The `int d_mounted` flag signalizing that the dentry object is mount point of other filesystem. It is used by the path look up routine to switch path look up to the other filesystem. The dentry object contains a pointer `struct dentry_operations d_op` to the dentry operations, which are described in section 2.2.7. Most filesystems don't fill these operations and keep them empty.

2.2.7 Dentry Operations

```
int d_revalidate(struct dentry *dentry, struct nameidata *nd)
```

Determines whether the `dentry` object is still valid. The VFS default function does nothing, although network filesystems (NFS) may specify their own function.

```
int d_hash(struct dentry *dentry, struct qstr *name)
```

Creates a hash value from `name` and saves it to the `dentry` object. This is used by filesystems that implement their own hash function for the dentry cache.

```
int d_compare(struct dentry *dentry, struct qstr *name1,
struct qstr *name2)
```

Compares `name1` and `name2`. The `dentry` argument is the parent dentry. Each filesystem can use its own comparison function. For example Window's file systems don't distinguish capital from lowercase letters.

```
int d_delete(struct dentry *dentry)
```

Called when the last reference to the `dentry` object is deleted. It means that `d_count` is zero.

`void d_release(struct dentry *dentry)`

Called when the `dentry` object is going to be freed.

`void d_iput(struct dentry *dentry, struct inode *inode)`

Called before the `dentry` object is freed. This function is used to notify the filesystem that the `inode` object associated with this `dentry` object should be freed. If the filesystem doesn't implement this function then the VFS's `iput` function is called. It decrements the inode `i_count` counter and if the `i_nlink` and `i_count` field is zero then the `inode` is freed.

2.2.8 File Object

A file object describes how a process interacts with a file it has opened. The file object is created when the file is opened and consists of a `struct file` structure defined in `include/linux/fs.h`. All opened files within one file system are linked through `struct list_head f_list` field. The header of this list is stored in the corresponding super block object. The file object contains a pointer `struct dentry *f_dentry` to the dentry object and a pointer `struct vfsmount *f_vfsmount` to the vfsmount object, which is described in section 2.2.11. The main information stored in the file object is the position in the file, which is kept in the `loff_t f_pos` field. Because inode can be used by several dentries, which means several files, it is not possible to keep this information in the inode object. The `f_pos` represents the current position in the file from which the next operation will take place. From other data fields we can mention the `unsigned int f_uid, f_gid` containing the uid and gid of the user which opened the file. Pointer to the file operations is stored in the `struct file_operations *f_op` field and it is a copy of the `struct file_operations *i_fop` pointer from corresponding inode object. Figure 2.5 shows how processes interact with the VFS through the file objects. We can see three processes and two of them have the same file opened. Every process in the Linux kernel is represented by the structure `struct task_struct` defined in `include/linux/sched.h`. The pointer for the currently running task in the kernel is stored in the `current` variable.

Every process keeps all its opened files in structure `struct files_struct *files` defined in `include/linux/file.h`. This structure contains the `struct file **fd` field, which points to an array of pointers to file objects. The size of the array is stored in the `int maxfds` field. Usually, `fd` points to the `struct file *fd_array[NR_OPEN_DEFAULT]` field. The `NR_OPEN_DEFAULT` defines the default size for the `fd_array`. It expands to the `BITS_PER_LONG`, which is defined differently for every architecture. For example, for i386 architecture its value is 32. If the file process opens more than `NR_OPEN_DEFAULT` files, the kernel allocates a new, larger array of file pointers and stores its address in the `fd` field and updates the `maxfds` field to the new array size. So when the user process opens a file the VFS creates all required objects and stores a pointer to the file object in the `fd` array. As a result the user process receives an index (file descriptor) to the `fd` field, where the pointer to the corresponding file is stored. Usually, the first element (index 0) of the array is associated with the standard input of the process, the second with standard output and the third with the standard error. Unix processes use the file descriptor as the main file identifier. When the user process wants to process some operation on a file, it forwards the file descriptor to the kernel. The kernel then gets a pointer to the corresponding file

object from `current->files->fd[fd]`. Notice that, thanks to the `dup(), dup2()` and `fnclt` system calls, two file descriptors may refer to the same opened file. That is, two elements of the array could point to the same file object. As an example we can mention the shell construct `2>&1`, which redirects the standard error to the standard output.

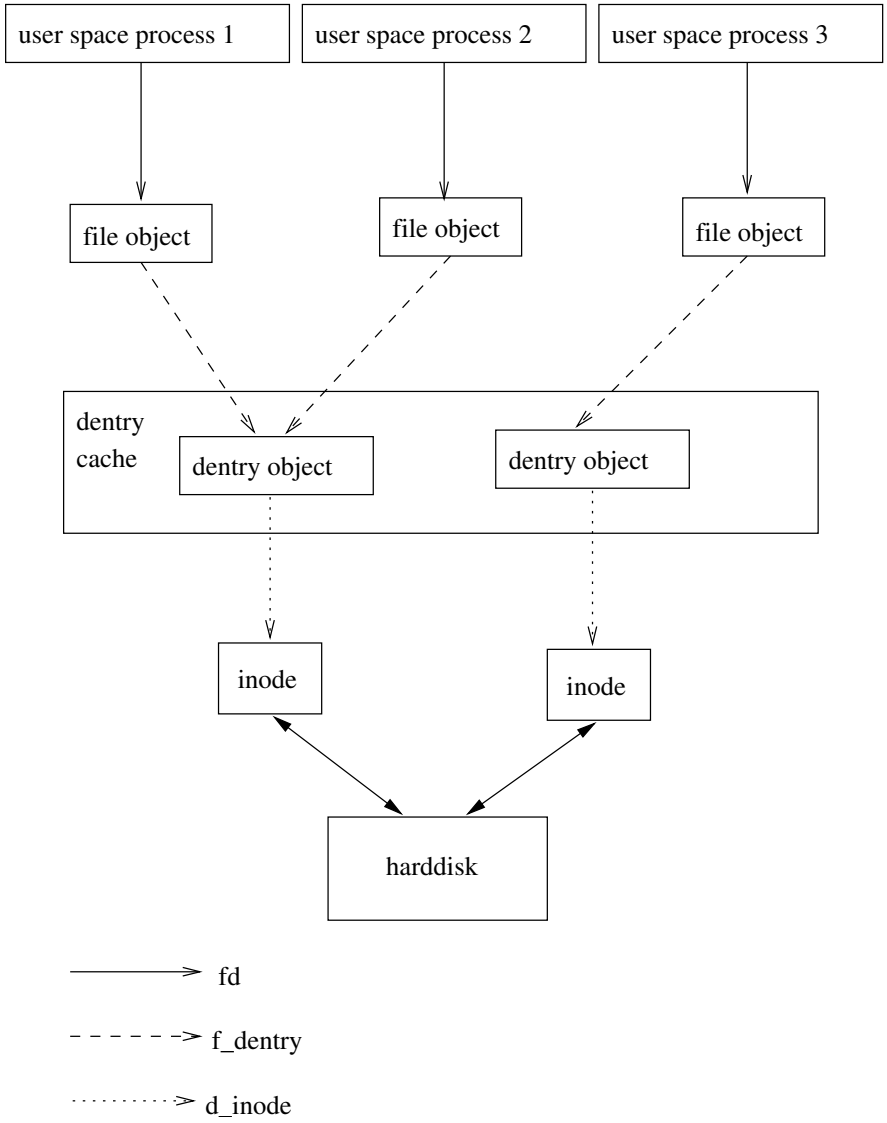


Figure 2.5: Interaction between processes and VFS objects via the file object

2.2.9 Filesystem Type Object

Each registered filesystem is represented in the Linux kernel as a filesystem type object. The filesystem type object is created for each filesystem and keeps its base information. It is represented by the `struct file_system_type` structure defined in `include/linux/fs.h`.

Thanks to this structure the Linux kernel knows about the filesystem. Filesystems can be included directly in the Linux kernel or can be registered on-the-fly as a Linux kernel module (LKM). All file type objects are inserted into a simply linked list. The `file_systems` variable defined in `fs/filesystems.c` points to the first item. Field `struct file_system_type *next` points to the next item in the list. Filesystem type contains the name of the filesystem stored in the `const char *name` field, filesystem flags are stored in the `int fs_flags` field (for example `FS_REQUIRES_DEV`, which says that the filesystem has to be stored on a physical disk device). In the `struct list_head fs_supers` field is stored the head of a list of superblock objects corresponding to mounted filesystems of the given type. The `struct super_block *(*get_sb)` field keeps a pointer to the specific filesystem function, which returns the filled super block object.

We will assume, that we have written a new filesystem. Now we have to somehow tell the kernel about it. This is done through the `register_filesystem()` function defined in `fs/filesystems.c`. Firstly a new filesystem type object has to be created and properly filled. Then the `register_filesystem` function is called with the newly created filesystem type object. The kernel adds our new filesystem to the `file_systems` list. Now when the new filesystem is mounted, the kernel searches for the filesystem type object corresponding to our new filesystem in the `file_systems` list. The search is based on the filesystem name, which is forwarded to the kernel as a `-t` mount option (for example ext2).

2.2.10 Fs_struct Object

In Linux, each process has its own current working directory and its own root directory. This information is stored in a fs_struct object represented by the `struct fs_struct` structure defined in `include/linux/fs_struct.h`. As mentioned in section 2.2.8 each process in the Linux kernel is represented by the `struct task_struct` structure. This structure contains a `fs` pointer to the fs_struct object for the current task. So the kernel knows the root and current working directory of each process. In fs_struct object are among others stored pointers to the root dentry (`struct dentry *root`), the current working directory dentry (`struct dentry *pwd`), a pointer to the vfsmount structure for the root directory (`struct vfsmount *rootmnt`) and a pointer to the vfsmount structure for the current working directory (`struct vfsmount *pwdmnt`). The vfsmount object is described in section 2.2.11.

2.2.11 Vfsmount Object

In the Linux kernel it is possible to mount a filesystem over another filesystem's directory tree. One filesystem can be mounted several times and can be even mounted on its own directory tree. It is also possible to stack multiple mounts on a single mount point. In this case only the last mounted filesystem is visible and hides the filesystem, which was mounted before it. After the top most filesystem is unmounted, then the file system below it is visible again. The vfsmount object is used to keep information about mounted filesystem and its relationship with other mounted filesystems. It is defined by the structure `struct vfsmount` defined in `include/linux/mount.h`. The vfsmount object contains the `struct vfsmount *parent` pointer to the parent vfsmount object on which it is mounted and a list of vfsmount objects (children) mounted on its directory tree. The head of this

list is stored in the `struct list_head mnt_mounts` field and vfsmount objects in this list are linked through the `struct list_head mnt_child` field.

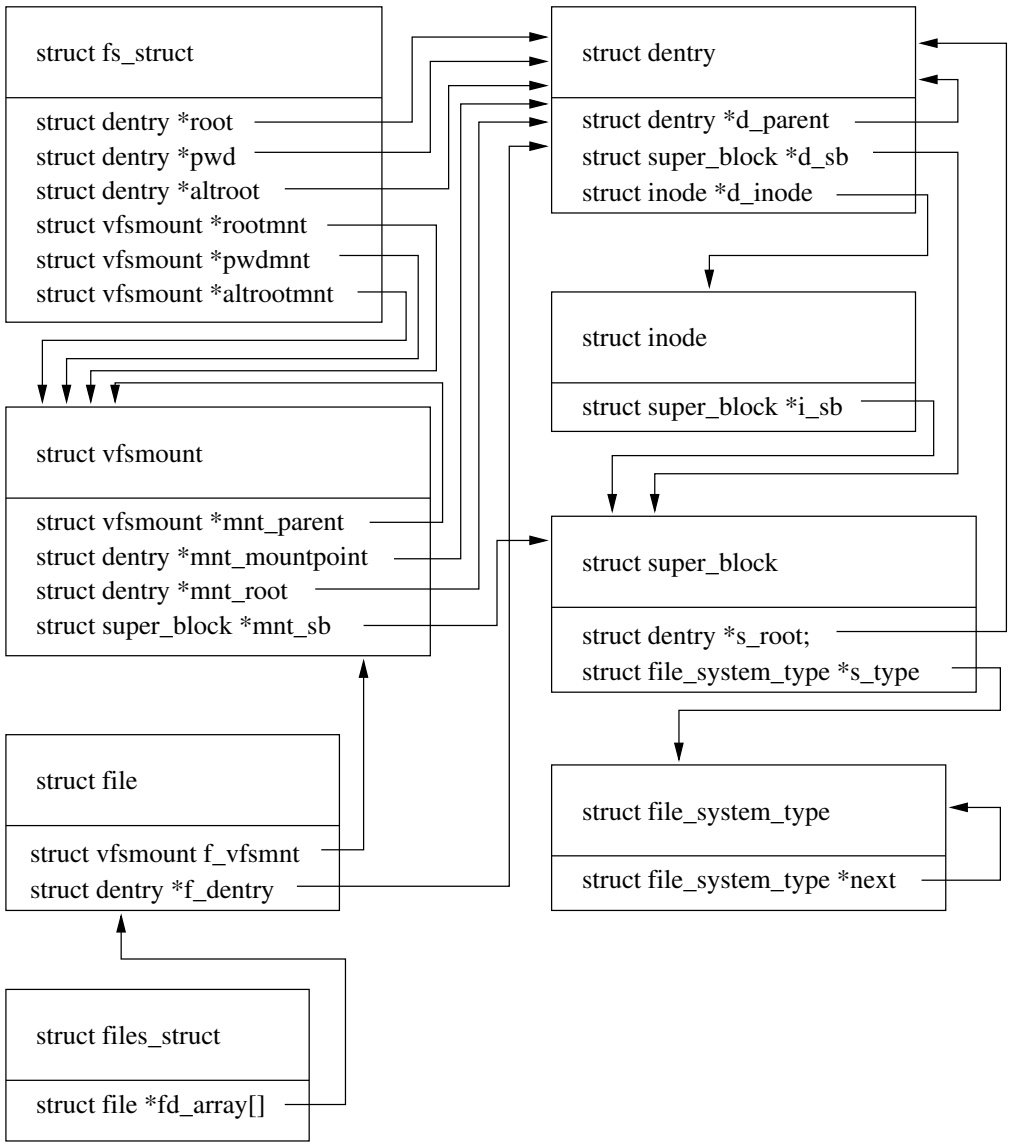


Figure 2.6: VFS objects connection

Figure 2.6 shows the base connection between VFS objects, which were described in the text above.

2.3 VFS Caches

VFS uses the dentry and inode cache to eliminate accesses to the hard disk. Because dentry and inode object are allocated and deallocated very often, VFS uses for these

objects the slab allocator. The slab allocator is based on the Linux buddy system for allocating continuous pages of dynamic memory. The idea behind the slab allocator is to preallocate several objects of given type and keep them in slabs. A slab is a series of contiguous physical pages in which objects are kept. The slab allocator then provides these preallocated objects. Objects are not deallocated but are returned back to the slab allocator. So there is no overhead for allocating and deallocating objects of given type because they are already allocated. Slab allocator is used for small objects which are used very often.

2.3.1 Dentry Cache

The dentry cache consists of n-ary dentry tree and dentry hash table. Both are protected against concurent acceess by one `dcache_lock` spinlock which is defined in `fs/dcache.c`. Dentry hash table `dentry_unused` is used for lookup operation which converts filename path to the corresponding dentry object. The address of the parent dentry object and the hash value of the dentry name are used as a key to the hash table. Filesystems can invalidate dentry objects in the dentry hash table. Which means that further lookup operations fail and real lookup operation of specific filesystem is called. Note that dentry is removed only from the hash table, but it still exists in the dentry tree and has the `DCACHE_UNHASHED` flag set. In Linux one process may delete file, but there can be other procceses using it. So a proper dentry object has to exist until the last reference to the dentry object is removed, but no other process can open it again.

2.3.2 Inode Cache

The inode hash table `inode_hashtable` is defined in `fs/inode.c`. The address of the supper block and inode number are used as a key to the hash table. Concurent access to the hash table is protected by the `inode_lock` spinlock.

2.4 Buffer Cache

The buffer cache is another cache whose purpose is to reduce slow disk operations. It is common for all native filesystems. When the buffer is read from disk its data is stored in the `buffer_head` structure. This structure contains, aside from the buffer data, some meta information about the buffer. For example if the buffer is dirty and etc. When the native filesystem tries to read or write data from or to the disk, the Linux kernel first looks for the corresponding buffer in the buffer cache.

Chapter 3

Linux Security Module – LSM

The Linux kernel provides only some base access control mechanisms. For example access rights to files are divided only for user, group and others. Some filesystems allow to use Access Control Lists which provide more flexible file access control. Other problem are root rights because root user can do anything in the system. Some programs need to run with root rights. This can be dangerous if such a program has a bug which allows non privileged process to obtain root rights. In this case it is desirable to reduce root rights or introduce control access based on processes. Some projects like RSBAC, Medusa, SELinux, LIDS or VXE provide a way to improve access control mechanisms. Each project provides its own set of patches for the Linux kernel to improve kernel security. The problem is that these patches have to be maintained for several Linux kernel versions and they cannot be used at the same time. The Linux Security Module or LSM tries to solve these problems and provides framework which allows third-party modules to register several security callback functions for different events in the kernel. It is intended as a common way for all security modules to obtain specific events from the kernel. The big advantage of the LSM framework is that it is integrated into the Linux kernel from version 2.5.

3.1 LSM Structure

The LSM provides the `struct security_operations` structure. It contains pointers for all operations which can be registered by the security module. The security module fills this structure with its callback functions and registers itself by the `register_security` function. This function takes as a argument pointer to the structure with callback operations. The security module can unregister itself with the `unregister_security` function.

3.2 LSM Integration in Linux Kernel

The LSM adds to the Linux kernel structures (task_struct, inode, file etc.) new data field. It is a void pointer which can be used by the security module to attach its private data to the kernel object. The LSM also adds a calling of registered callback functions to the specific functions in the kernel.

3.3 LSM and Modules Stacking

The LSM provides a very primitive stacking of security modules. The LSM allows only one security module to be directly registered in the LSM framework. This module is called master. The LSM framework contains the pointer `security_operations *security_ops` which is defined in `security/security.c`. This pointer always points to the master's `security_operations` structure. The LSM contains `dummy_security_ops` which are used when there is no security module (master) registered in the framework. When the first security module (master) is registered the LSM changes the `security_ops` pointer to the new security module operations. All security functions in the kernel are called through the `security_ops` pointer. This is why only one security module can be directly registered in the LSM framework. The structure `security_ops` contains pointers for registration and unregistration functions. The master module has to implement these operations to allow other security modules to use the LSM. When a second security module registers itself to the LSM then registration function of the master module is called. When a third security module registers itself then the registration function of the master module is called and this function has to call the registration function of the second module. And so on. This is how security modules are stacked in the LSM framework. It is obvious that each security module is responsible for calling the security module which was registered after it. This approach is very bad. There is no way to affect the order in which the security modules will be called. The security modules can not be safely removed. Only the master module can attach private data to the kernel objects. Moreover the LSM doesn't provide enough callback functions for all security modules.

At this moment only a few security modules are using the LSM framework.

Chapter 4

Overlay Filesystems

The overlay filesystem creates new a layer over the existing native filesystem. The idea is to create a whole filesystem which will be linked with the native filesystem. VFS doesn't provide any special support for overlay filesystems. The overlay filesystem is mounted over the native filesystem and it works with the VFS objects which are created by native filesystem. Overlay filesystems can be stacked on each other. So for example there can be native filesystem on which can be linked several overlay filesystems. The problem is that overlay filesystem duplicates all VFS objects (inode, file, dentry) of the filesystem under it (it can be native filesystem or other overlay filesystem). So if three overlay filesystems are used over one native filesystem, every VFS object (file, dentry and inode) has four copies in the VFS layer. One copy for native filesystem and three copies for overlay filesystems. Another problem is that some applications need to be notified about only a few VFS events. For example antivirus on-access scanners just need to know when the file is opened, executed or closed. To write a whole filesystem only to get these events is really a lot of work. There is no way to affect the order in which overlay filesystem will be called and of course overlay filesystem can not be safely unmounted because it can be used by other overlay filesystema.

4.1 FiST

FiST is a project which tries to make the creating of overlay filesystems easier. Actually it is an overlay filesystem generator. If you want to create a new overlay filesystem you only need to implement functions in which you are interested. FiST will generate a whole filesystem for you. The advantage is that FiST describes the overlay filesystem in its special language and provides generators for Linux, FreeBSD and Solaris. It means that the overlay filesystem is defined only once and will work on all three operating systems.

Chapter 5

Dazuko

Dazuko aims to be a cross-platform device driver that allows applications to control file access on a system. The current release of Dazuko supports Linux 2.2-2.6, Linux/RSBAC, and FreeBSD 4/5 kernels. It is mainly used by anti-virus companies for on-access scanning. The first version was developed by H+BEDV Datentechnik GmbH (anti-virus company from Germany) which was later released as an open source code under BSD and GPL licence. Dazuko is now maintained by John Ogness. In the following text will be described Dazuko for Linux kernels.

Dazuko consists of the Linux kernel module called dazuko and a user space library called dazukoio. Communication between the module and the library is done through the `/dev/dazuko` char device with major number 33.

5.1 Dazuko Interface

Dazuko provides a well defined interface which allows user space processes to be notified about some filesystem events. Currently Dazuko generates the following events: on_open, on_close, on_close_modified, on_exec, on_unlink and on_rmdir. Dazuko provides interface for several programing languages like C, Python, Perl or Java. Here will be briefly described the interface for the C language.

All functions return zero on success.

Registration

```
int dazukoRegister(const char* groupName, const char *mode)
```

Registers an application to the Dazuko module. The argument `grouName` specifies the group name to which the application will be added. This is useful when an application runs in several instances. Dazuko then delivers event which occurred to the first available application in the group. The second parameter `mode` specifies the mode in which the application will interact with Dazuko. There are currently two modes available, read-only "r" and read-write "r+". The read-only mode allows an application to receive all accesses but does not give the application an opportunity to decide if the access should be allowed or not.

Configuration

`int dazukoSetAccessMask(unsigned long accessMask)`

Sets bit mask of events which will be delivered to the application. Possibilities are `ON_OPEN`, `ON_CLOSE`, `ON_CLOSE_MODIFIED`, `ON_EXEC`, `ON_UNLINK` and `ON_RMDIR`.

`int dazukoAddIncludePath(const char *path)`

Adds a new directory path (including all subdirectories) for which events will be delivered to the applications.

`int dazukoAddExcludePath(const char *path)`

Removes the directory path.

`int dazukoRemoveAllPaths(void)`

Removes all guarded paths.

Access Control

`int dazukoGetAccess(struct dazuko_access **acc)`

This is a blocking function which waits until the event is delivered. The argument `acc` then contains all information about this event.

`int dazukoReturnAccess(struct dazuko_access **acc)`

The application which is registered with "r+" mode has to call this function because the dazuko module waits for an answer.

Unregistration

`int dazukoUnregister(void)`

Unregisters the application from the dazuko module.

5.2 Interaction with Linux Kernel

5.2.1 Replacement of Syscall Table

For Linux kernels 2.2.x and 2.4.x Dazuko replaces addresses of selected functions in the syscall table. The advantage of this approach is that Dazuko doesn't need to modify the Linux kernel source code and it is relatively easy to replace functions in the syscall table. Generally it is not a good idea to modify the syscall table. It is work for a rootkit, not for the security module. Imagine that the Linux kernel will contain some rootkit detector. In this case Dazuko will not be able to work. Another problem is that the syscall functions are to high. This means that Dazuko is not able to catch accesses to the filesystems, for example, from NFS kernel daemon. Dazuko has to replace `sys_open` syscall to be able to

generate on_open event. This means that it is called every time the file is opened and it has to check if the file is in some directory paths which were selected by applications.

5.2.2 LSM Interface

For Linux kernels 2.6.x Dazuko uses LSM framework. Is is better than replacing the syscall table, but maintaining the security module for the LSM is a lot of work. The problem is in the LSM's approach to modules stacking. The LSM framework also doesn't provide a way to catch on_close events.

5.2.3 RSBAC Interface

Dazuko uses for the Linux kernels 2.6.x in addition to the LSM framework also the RSBAC framework which solves the problem with on_close events.

5.2.4 Overlay Filesystem

Currently Dazuko uses three ways to interact with the Linux kernel. Overlay filesystem should replace all of them. It is being implemented in these days and uses the FiST generator.

Chapter 6

RedirFS Framework

RedirFS stands for **REDIR**ecting **F**ile**S**ystem. In chapters 3, 4 and 5 were described the existing possibilities of file access control in the Linux kernel and their limitations. Among these limitation belong replacing the syscall table, duplication of VFS objects, bad support for security modules, absent safety unloading of security modules, no possibility how to specify order in which the security modules will be called and so on. To overcome these limitations, I propose a new solution - the RedirFS Framework.

6.1 RedirFS Goals

- Provide general, fast, flexible and open source framework allowing to redirect native file system calls in VFS objects.
- Should not modify Linux kernel source code.
- Implemented as a Linux kernel module.
- Independent of the Linux kernel version.
- Be able to register, unregister and manage several third-party filters (other Linux kernel modules).
- Provide a well defined interface allowing filters to set callback functions for selected file system operations.
- Allow filters to set pre and post native file system callbacks.
- Allow filters to select directory subtrees for which their callback functions will be called.
- Provide a way to allow filters to exclude selected directory subtrees.
- Be able on-the-fly to include and exclude directory subtrees and add, remove or modify filter's callback function.
- Modify only VFS object operations which have to be modified.
- Be able to call filters in a specific order.

- React on return values from filters and be able to interrupt the filter call chain.
- Allow the filter to forward data from pre to post callback function.
- Allow filters to attach their private data to selected VFS's objects.
- Return VFS objects operations to the native file system calls as soon as possible.
- Avoid as much as possible VFS objects duplication.

6.2 RedirFS Design

As you can see in figure 6.1 RedirFS Framework creates a new layer between the VFS and native filesystems and has two main functions.

1. Interact with the VFS layer and redirect operations selected by filters over specified directory subtrees to the Framework.
2. Interact with filters and call all of them which are interested in the VFS event which occurs.

The RedirFS Framework provides a well defined interface to filters which is described in detail in section 6.6. Filter can ask the RedirFS Framework for the following actions.

- Register itself to the RedirFS Framework. It is the first thing that the filter has to do. As a result the filter receives from the RedirFS Framework its handler. The filter has to use this handler in every following interaction with the RedirFS Framework to identify itself. During registration the filter has to set its name and priority. From the RedirFS point of view the filter's name is not very important and is used only for some statistic or debug information. Several filters can have the same name. More important is the filter's priority. It is a unique number and the RedirFS Framework calls filters in order defined by theirs priorities. They are called upwardly from lower to higher numbers. Lower number means higher priority.
- Set or modify pre and post callback filter's functions for selected VFS operations.
- Remove previously set callback functions.
- Set directory subtrees over which the filter's callback functions will be called.
- Remove previously set directory subtrees. When the filter removes the directory subtree which is part of other subtree previously set by the same filter then the directory subtree which it about to be removed is excluded from the previously set subtree.
- Activate itself. The filter is not active by default. It means that the filter can set callbacks and subtrees, but it doesn't receive any events until it is activated.
- Deactivate itself. The filter is not removed from the RedirFS Framework, but it is just not notified about registered VFS events until it is activated again.

- Remove itself. This completely removes the whole filter (Its callbacks and all subtrees).

The whole idea behind the RedirFS Framework is to replace selected VFS object operations by RedirFS operations so that VFS calls RedirFS framework instead of native filesystem. RedirFS is then able to notify all interested filters that the VFS tries to call native filesystem. Figure 6.1 shows that only operations selected by filters are redirected to the RedirFS Framework. All other operations go directly to the native file system and the RedirFS Framework doesn't affect them in any way.

6.3 RedirFS Architecture

The RedirFS Framework is formed by several objects which are in detail described in section 6.4. Here their purpose will be described.

6.3.1 Root Object

Filters can select for directory subtrees their callback functions will be called for. Directory subtrees are identified by full directory path. When a filter asks the RedirFS Framework for a new subtree, it creates a root object representing the requested subtree. Then the filter is attached to this new root object. It is possible that the root object for the requested subtree already exists. It can happen when another filter already selected the same subtree. In this case the RedirFS Framework only attach filter to the existing root object.

Root objects are connected in n-ary trees like dentry objects in the VFS. The difference is that in the VFS there is only one main dentry object (dentry object for root directory) and in the RedirFS Framework can be several main root objects. These main root objects are linked in a list. The whole RedirFS Framework is based on root objects and it is important to understand how the root objects are connected and how the root objects tree is modified while new filters and directory subtrees are added. In figures 6.2, 6.3 and 6.4 is shown what happen with root objects trees when three different filters are added.

In figure 6.2 only one filter called cryptfs is registered in the RedirFS Framework and it has selected two directory subtrees. So two new root objects are created and the cryptfs filter is attached to them.

In figure 6.3 is registered second filter called avgflt. This filter has selected only one directory subtree, but it contains both subtrees which were set by cryptflt filter. Root objects are reorganized and the avgflt filter is attached to the new root object and to all root objects which are under it. In figure 6.4 is attached one more filter called the compressflt. It has selected the `/home/john/data/compress` subtree which belongs under the `/home/john/data` root object. And again the compressflt is attached to the new root object and to all root objects which are under it. In this case there are no root objects under it.

Root objects connection in figures 6.2, 6.3 and 6.4 are simplified and also only filter pointers are attached to the root objects. Detailed information is in section 6.4. Note that filters are attached to the root objects in a specific order. It is because some filters have to be called sooner or later then others. The order is specified by the filter's priority.

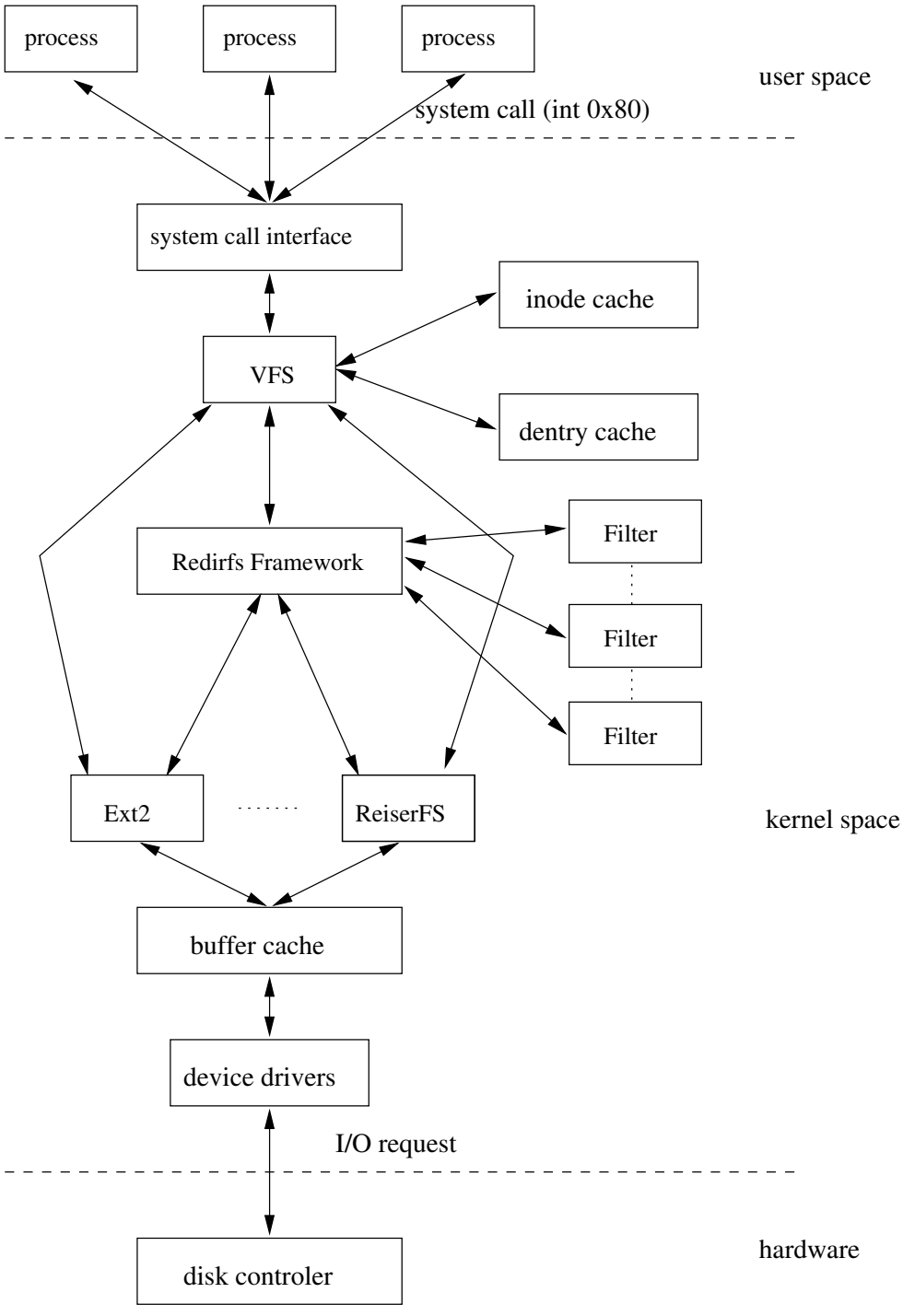


Figure 6.1: The RedirFS role during system calls

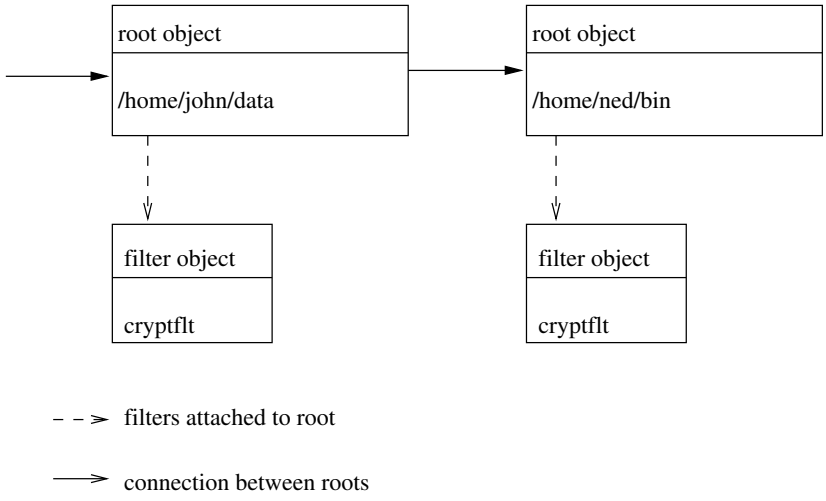


Figure 6.2: Example: RedirFS root objects for one filter

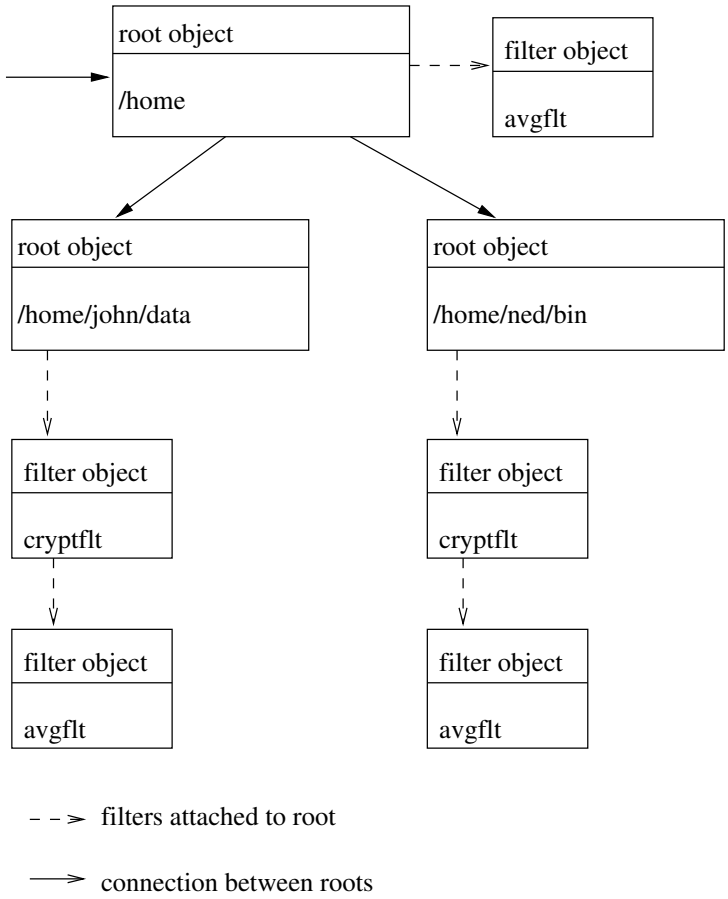


Figure 6.3: Example: RedirFS root objects for two filters

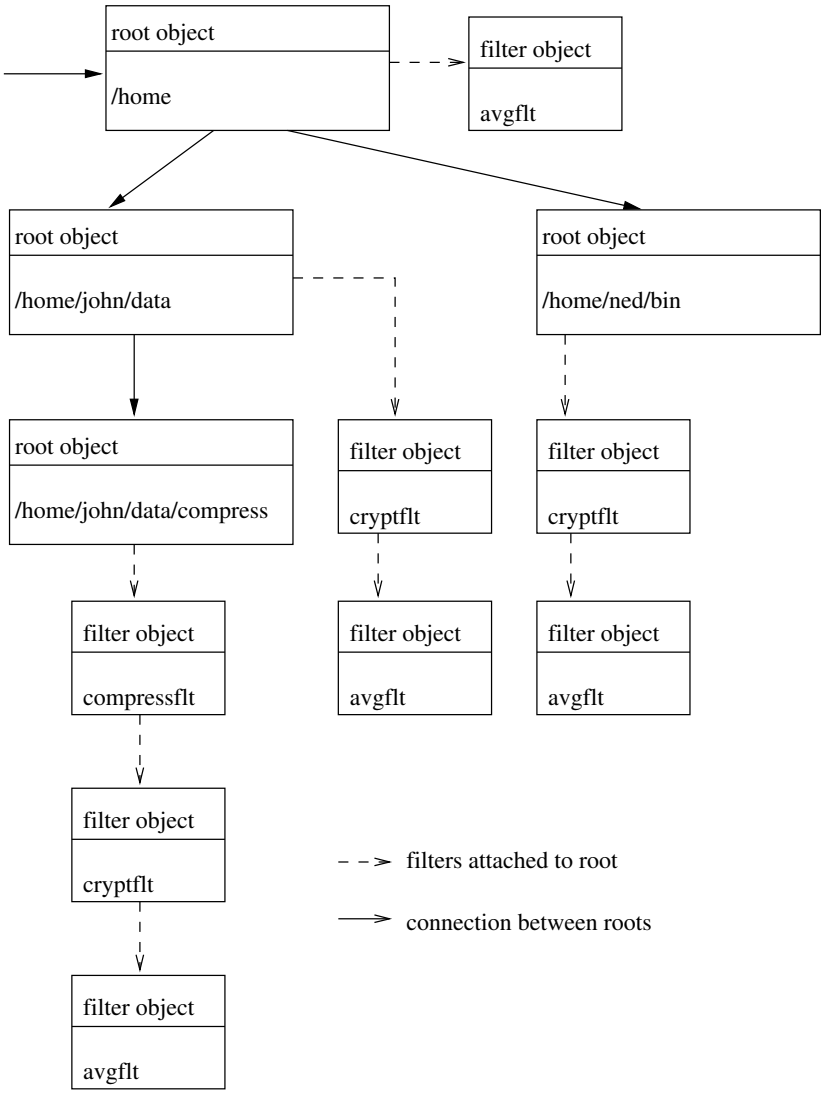


Figure 6.4: Example: RedirFS root objects for three filters

When a filter wants to remove or exclude some previously set directory subtree, the RedirFS Framework detaches it from all corresponding root objects. If it is the last attached filter then the root object is destroyed.

6.3.2 Filter Object

Each registered filter is in the RedirFS Framework represented by the filter object which contains all its information. As filter asks RedirFS Framework for inserting and removing directory subtrees, the RedirFS Framework is attaching or detaching the filter object to or from the corresponding root objects.

6.3.3 Inode Object

The RedirFS inode object is created for each VFS inode which is in one of the directory subtrees selected by filters. It is dynamically created and destroyed along with the VFS inode. Note that the RedirFS inode is different than the VFS inode. It is used to find out which root object corresponds to the VFS inode. It is very important to find the right root object to the VFS inode object. When the VFS object operations are redirected to the RedirFS Framework, it is only the VFS inode which can be always obtained. When we have the VFS inode object we also can obtain the corresponding RedirFS inode which contains pointer to the proper root object. Then RedirFS Framework can call all filters attached to the root object. Of course only those filters which are interested in native filesystem operation which occurred.

6.4 RedirFS Data Structures

In this section are in detail described all data structures used in RedirFS Framework.

6.4.1 Root Object

The root object is represented by structure `struct redirfs_root_t`, defined in `root.h`. Each root object is attached in one of several n-ary trees. Number of these trees and their hierarchy depends on directory subtrees selected by filters. This was described in section 6.3. All roots of these trees are linked in a list which header `redirfs_root_list` is defined in `root.c`. Concurrent access to the hierarchy of root objects is protected by the `redirfs_root_list_lock` spinlock.

`struct list_head attached_flts`

Each root object can have attached several filter objects. Pointers to all attached filter objects are linked in a list whom header is stored in this field. Note that filters in this list are in specific order which depends on filters priorities.

`struct list_head subroots`

All root objects which belong directly under other root object are linked in a list which header is stored in this field. In other words all root's children are linked in this list.

`struct list_head sibroots`

Through this field are linked all root objects on the same level (siblings).

`struct list_head remove`

Through this field are linked all root objects which are going to be removed. List header `redirfs_remove_root_list` of this list is defined in `root.c`. List is protected by `redirfs_remove_root_list_lock` which is also defined in `root.c`.

`struct list_head inodes`

List header of all RedirFS inode objects which belong to the root object. RedirFS inode object is different than the VFS inode object and is used to find out right root object.

`struct redirfs_root_t *parent`

Pointer to the parent root object. If root object has no parent then this field is set to NULL.

`struct dentry *dentry`

Every root object represents one directory subtree which is defined by full directory path. This field contains pointer to the dentry objects which represents this directory in VFS.

`spinlock_t lock`

Is used to protect all data fields in root object against concurrent access.

`char *path`

Directory subtree path which is represented by root object.

`atomic_t flt_cnt`

Number of attached filters.

`struct redirfs_operations_t *fw_ops`

Every root object contains a pointer to the RedirFS Framework operations. These operations are common for all root objects and their purpose is to call all Filters attached to the root.

`struct redirfs_operations_t new_ops`

This field contains pointers to the structures of new operations for all VFS objects belonging to the directory subtree which is represented by the root object.

`struct redirfs_operations_t orig_ops`

This field contains pointers to the structures of original native filesystem operations originally used by VFS objects.

`struct redirfs_vfs_operations_t vfs_ops`

This field contains structures of new VFS operations pointers.

`struct redirfs_operations_counters_t new_ops_cnts`

In this field are stored counters for each operation in the `vfs_ops` field. Note that each operation can be used by several filters.

6.4.2 Filter Object

Filter object is represented by structure `struct redirfs_flt_t`, defined in `filter.h`. All filter objects are linked in a list which header `redirfs_flt_list` is defined in `filter.c`. The list is protected against concurrent access by `redirfs_flt_list_lock` spinlock.

`struct list_head flt_list`

Filter objects are linked through this field in the filters list.

`spinlock_t lock`

Is used to protect all data fields in the filter object against concurrent access.

`char *name`

Filter's name.

`int turn`

Filter's priority.

`unsigned int flags`

Filter's flags. For now this field is not used and is reserved for future usage. Here could be stored for example information if the filter wants follow submounts or shrink dentry cache.

`atomic_t active`

Contains information whether the filter is active (1 – active, 0 – not active).

`struct redirfs_vfs_operations_t vfs_pre_ops`

Filter's pre callback functions.

`struct redirfs_vfs_operations_t vfs_post_ops`

Filter's post callback functions.

`struct redirfs_operations_t pre_ops`

Pointers to the filter's pre callback functions.

`struct redirfs_operations_t post_ops`

Pointers to the filter's post callback functions.

6.4.3 Inode Object

Inode object is represented by the structure `struct redirfs_inode_t`, defined in `inode.h`. For RedirFS inode allocation and deallocation is used the Linux kernel slab allocator. Each RedirFS inode is located in the `redirfs_ihash` hash table.

`struct hlist_node inode_hash`

All RedirFS inode objects with same hash value are linked in list through this field.

`struct list_head inode_root`

Each RedirFS inode object belongs to the one root object. All RedirFS inode objects which belong to the root are linked through this field.

`struct list_head priv`

For now this field is not used and is reserved for future usage. Filters will be able through this field to attach their private data to the VFS inode object.

`struct redirfs_root_t *root`

Pointer to the root object to which this inode object belongs.

`spinlock_t lock`

Is used to protect all data fields in filter object against concurrent access.

`struct super_block *sb`

Pointer to the VFS super block object.

`unsigned long ino`

Inode number of corresponding VFS inode object. This and the previous field is used to find corresponding RedirFS inode object to the VFS inode object.

6.4.4 Operations Object

The operations object is represented by the structure `struct redirfs_operations_t`, defined in `operations.h`. It contains pointers to the inode and file operations for regular file and directory. It also contains a pointer to the dentry operations. Moreover it defines for each VFS operation the structure array which is used for more comfortable way how to individually access elements in structure. Each operations object has to be initialized by `redirfs_init_ops` or `redirfs_init_orig_ops` function. These functions properly set all arrays and pointers in operations object. Which function to use for initialization depends if operations structures are known or not. The `redirfs_init_ops` function is used when we know them and `redirfs_init_orig_ops` otherwise.

6.5 RedirFS Implementation

This section describes implementation of the RedirFS Framework. RedirFS Framework as it is implemented right now has several limitation because not all planned features were implemented. Note that RedirFS is well defined and all features which were not for now implemented can be easily added in the future because RedirFS is ready for them.

For now redirfs allows to redirect inode, file and dentry operation for regular files and directories. Filters can register only inode permission function and all submounts directories are ignored.

6.5.1 Replacement of VFS Object Operations

Linux VFS layer provides API which allows to work with VFS objects. RedirFS Framework uses this API and connection between VFS objects to replace operations in VFS objects. Note that operations replacement is done only by assigning new pointer value to the VFS object.

Replacement of Existing Dentry and Inode Object Operations

Dentry objects are in the Linux VFS connected in n-ary tree. VFS provides the `path_lookup()` function to convert filename path to the corresponding dentry object. RedirFS Framework uses this function when the filter wants to include or exclude some directory subtree to find directory dentry object. Now it can goes through dentry subtree and replace operations in all existing dentry and inode objects. Note that each dentry has pointer to the corresponding inode. RedirFS provides `redirfs_walk_dcache()` function which locks dentry and inode cache and goes through specific dentry subtree. It takes five arguments. Start dentry object, callback function which is called for every dentry object in the dentry subtree, callback data for this function, callback function which is called when the mount point dentry is found and its callback data. Note that to some dentry objects in the dentry subtree can be already attached other RedirFS root objects. If this is true then the `redirfs_walk_dcache()` function has to skip those dentries. We can not change operations for other root objects because they can have attached different filters and can replace different operations. This function is called only in two cases. When filter inserts new directory subtree which is not already added and when the last filter is detached from root object.

Dentry and inode object operations are replaced in `redirfs_set_new_ops()` function, which is called for every dentry object in dentry subtree, by `redirfs_walk_dcache()` function. This function beside replacing operations has to save old VFS objects operations. It is important because RedirFS has to set original operations back when the last filter is detached from the root object. Note that not all native filesystem operations are known at the beginning. For example if `redirfs_set_new_ops()` function gets dentry object of regular file for the first time it saves original operations to the `orig_ops` structure in corresponding root object.

Function `redirfs_set_orig_ops()` is called in the same way as `redirfs_set_new_ops()` and assigns VFS objects back original operations. This is very easy because RedirFS knows about all operations which previously replaced.

Replacement of Existing File Object Operations

Replacing file object operations is a little bit complicated. The file object, when is created, copies the pointer to the file operations from corresponding inode object. So replacing file operations only in the inode objects is not enough. The problem is that there is no way how to get file objects related with inode or dentry object. File object contains pointer to the corresponding dentry object but dentry object does not contain pointer to the file object. It is logical because dentry object can be used by several files. But this is the same situation as with inode and dentry objects. Each inode object can be used by several dentries and in this relation inode object contains list of all dentry objects which are using it. However this is not true in relation between dentry and file object and RedirFS has to find other way how to replace operations in file objects. All file objects created over one filesystem are linked in a list which header is stored in the super block object. Each dentry object contains pointer to the super block object. So when RedirFS knows dentry object for the directory subtree it can go through the list of all file objects and replace their operation. Other problem is that in this list are linked all file object. It means that there are file objects which do not belong under specified directory subtree. Here RedirFS has to compare file names and replace operations only in file objects which belongs under the directory subtree. RedirFS provides `redirfs_replace_files_ops` which locks the files list lock and goes through all file objects in the list. It takes four arguments. Directory path for which will be file objects operations replaced, dentry objects representing directory path, new file operations and file type for which will be the operations replaced. Here again RedirFS has to take care when the file object does not belong under other root object. This is called in same cases as `redirfs_walk_dcache` function and when original file operations are obtained.

6.5.2 Replacement of Newly Created Objects Operations

This is easier because all operations of VFS objects in caches are already replaced. RedirFS only need to know which operations are creating new objects. Then let the native filesystem to create these objects and replace their operations before they will be returned back to the VFS. Follows functions which create new VFS objects.

Create operation in directory inode

This operation creates new regular file in specified directory.

Lookup operation in directory inode

This operation is called by VFS if it does not found corresponding dentry object in path lookup operation. This operation is called for every file type.

Mkdir operation in directory inode

This function creates new directory object.

This functions can handle only regular files and directories. In future RedirFS will be able to handle all file types. For example to handle creating of special files like char devices, fifos or block devices, RedirFS need to implement mknod operation for directory inodes. There is one more problem because for example each char device has its own operations. So it is not possible to handle them all in one way like regular files and directories operations which are same for whole file system. RedirFS Framework is ready for this.

6.5.3 Root Objects Manipulation

RedirFS provides several functions which operate over root hierarchy. Base function are `redirfs_add_root`, `redirfs_remove_root`, `find_root_parent` and `redirfs_find_root`.

Most important is `redirfs_walk_roots()`, defined in `root.c`. This function goes through all root objects. It takes three arguments. Root object from which to start, callback function and callback function data. If start root object is NULL then callback function is called for every root object in root hierarchy. This function is used for attaching and detaching filter to and from root objects (`redirfs_attach_flt` and `redirfs_detach_flt`).

6.5.4 Operations

Each root object contains two sets of operations. First are operations in which are stored pointers to the native filesystem operations. These operations are never changed and are used when RedirFS Framework wants to set original operations back to the VFS objects. These operations are stored in the `orig_ops` field in root object. Note that pointers in `orig_ops` field cannot be changed because any change affects the whole native filesystem. The second operations contains copies of native filesystem operations and are stored in the `new_ops` field in root object. These pointers can be modified because root object has its own copy. To all VFS objects which belong to specific root object are changed operations to the operations stored in `new_ops`. At the beginning `new_ops` contains copies of native filesystem operations. Only several function needed directly by RedirFS Framework like create, mkdir or lookup are redirected.

RedirFS Framework contains global operations `redirfs_fw_ops` defined in `redir.c`. When filter wants to register new callback function then corresponding function pointer in `new_ops` is replaced by pointer from `redirfs_fw_ops`. So only one pointer assignment changes specific operation for all involved VFS objects. Operation from `redirfs_fw_ops`

calls all filters which are interested in VFS event which occurs. Each operation in `new_ops` has counter. If counter for specific operation is zero then pointer for this operation in `new_ops` is replaced back to original operation from `orig_ops`.

6.5.5 RedirFS Inodes Manipulation

Selected VFS objects operations are redirected to the RedirFS Framework operations. VFS then calls RedirFS operations instead of native filesystem operations. Interfaces of native filesystem operations does not know anything about RedirFS Framework objects. For example, VFS calls permission function which is redirected to the `redirfs_reg_permission()` function. This function has three arguments. Inode object, mode, and nameidata structure. So RedirFS Framework has to introduce some way how to obtain corresponding root object. The root object contains attached filters which have to be called. For this purpose RedirFS Framework has its own inodes which contains pointers to the corresponding root objects. VFS inode object can be somehow obtained in every function. Each VFS inode is unambiguously identified by super block pointer and inode number. So RedirFS uses these two values as a key how to find correspond RedirFS inode to the VFS inode. RedirFS inode is created for each vfs object which has redirected operations.

Note that RedirFS inodes have to stay consistent with VFS inodes. In VFS can exist dentry object without inodes and inodes without dentry object. For example Linux kernel can in every time prune dentry cache. For this reason RedirFS Framework redirect dentry operation `d_iput` which is called when dentry object is going to be removed from dentry cache.

6.5.6 Example of Including New Path

1. Filter calls `redifs_include_path()` with path argument.
2. Function `redirfs_find_root()` tries to find corresponding root object in root hierarchy. If the root has been found, go to 3 else go to a.
   (a) `redirfs_alloc_root()` allocates new root object.
   (b) `redirfs_find_root_parent()` tries to find parent root for newly created root. If the parent has been found, goto i else goto c.
      i. `redirfs_inherit_root()` copies all operations from parent to the new root object.
   (c) `redirfs_add_root()` adds new root to the root hierarchy.
   (d) `redirfs_walk_dcache()` with `redirfs_set_new_ops()` function redirects all operations of VFS objects which belong to the new root.
   (e) `redirfs_replace_files_ops()` replaces operations in corresponding file objects.
3. `redirfs_walk_roots()` with `redirfs_attach_flt()` function attaches filter object to the all corresponding root objects in root hierarchy.

6.6 RedirFS Interface

The whole interface to the RedirFS Framework is stored in the `redirfs.h` header file. In this section are described all data types and operations provided by RedirFS Framework to filters.

6.6.1 Data types

File and Operations group identifiers

Linux distinguishes between seven file types – regular files, directories, fifos, sockets, pipes, char devices and block devices. Each file type has its own groups of operations. For example inode and file operations. When a filter wants to register some callback function it has to specify to which file type and operation group the callback function is related. For example `REDIRFS_I_DIR` specifies that the filter callback function is related to the inode operations group for directory. All other file types and operation groups identifiers are defined in `redirfs.h`.

Specific Operation identifier

From previous paragraph we know how to distinguish between file types and operations groups. Other thing which filter has to specify in callback function registration is specific operation in operations group. RedirFS Framework defines identifier for each operation in each operations group. For example `REDIRFS_IOP_MKDIR` represents mkdir inode operation. So when filter wants to register callback function it has to specify two identifiers. First is file type and operations group and second is specific operation. These two identifiers provide way how to unambiguously identify each VFS operation.

`enum redirfs_retv`

This enum contains all return values which can be returned from the filter callback function back to the RedirFS Framework. For now it has only two fields. `REDIRFS_RETV_STOP` tells RedirFS Framework to interrupt calling of subsequent filters and `REDIRFS_RETV_CONTINUE` allows to call next filter in chain.

`typedef void* redirfs_filter`

After registration filter receives handler of this type. Filter then has to, in every further interaction with RedirFS Framework, identify itself with this handler.

`typedef void* redirfs_context`

This type has for now no functionality. It is reserved for future usage. In future it will keep operation context data and will allow filter for example to attach its private data to VFS objects. For now it is used only for filter callback functions interface entireness.

`union redirfs_op_args_t`

Each native file system call has several arguments. Arguments for each file system function are grouped into a structure and `redirfs_op_args_t` contains union of all these structures. For example all arguments for inode create function are grouped in `i_create` structure. Filter callback function receives pointer to this union and in proper field will be all arguments for the native filesystem function.

`union redirfs_op_retv_t`

Contains union of all data types which can be returned from native file system calls. Filter callback function has to return proper value in specific field so that RedirFS Framework can return this value to the VFS. This value has to be set only when filter forbids to call other filters with lower priority.

`struct redirfs_op_exts_t`

Stores extended data which might be useful for some filters. At this moment this structure is not used and is reserved for future usage.

DATA FIELDS

`const char* fill_path` – Full path name of accessed inode.

`struct redirfs_args_t`

Contains all information which filter callback function can receive from the RedirFS Framework. It just encapsulates several other structures and provide a flexible way for future extensions without interface modification.

DATA FIELDS

`union redirfs_op_args_t args` – Arguments of original native file system function.

`struct redirfs_op_exts_t exts` – Extended information.

`union redirfs_op_retv_t retv` – Return value of original native filesystem function.

`struct redirfs_op_t`

Is used for filter callback function registration and unregistration. Pre and post callback functions pointers can be set to NULL. This has different behavior for registration and unregistration. For registration NULL pointer means that filter doesn't want to register this function, but for unregistration it means that filter wants to remove this callback function. For example if the filter wants to register only pre callback function it sets the post callback function pointer to NULL. If the filter wants to remove the post callback function, but keep pre callback function it has to set the post callback function pointer to NULL and the pre callback function pointer to some value different from NULL.

DATA FIELDS

`int type` – File and operation type identifier. As mentioned earlier, Linux has seven file types. Here has to be specified to which file type callback function belong. Please note that every file is in Linux represented by several structures and each structure has its own operations. So this field identifies aside from a file type also operation type. All usable values are defined in `redirfs.h`. For example `REDIRFS_I_REG` says that callback function will be used for regular file and inode operations.

`int op` – By `type` field is selected group of operations. This field selects specified function in this group. If `REDIRFS_I_REG` value is set in `type` field then `op` field has to contain inode operation identifier. For example `REDIRFS_IOP_LOOKUP` value.

`enum redirfs_retv (*pre_op)(redirfs_context context, struct redirfs_args_t *args)` – Pre callback function pointer.

`enum redirfs_retv (*post_op)(redirfs_context context, struct redirfs_args_t *args)` – Post callback function pointer.

Filter Callback Function Interface

Each filter callback function has to have following interface
`enum redirfs_retv func_name(redirfs_context context, struct redirfs_args_t *args)`.

6.6.2 Operations

```
redirfs_filter redirfs_register_filter(const char *name, int priority,
unsigned long flags)
```

ARGUMENTS

`name` – Filter name.

`priority` – Filter priority.

`flags` – Filter flags. For now this argument is not used. It is reserved for future usage.

RETURN VALUE

On success, filter handler is returned. On error, a negative value is returned. For return value testing has to be used Linux kernel functions `IS_ERR` and `PTR_ERR` defined in `include/linux/err.h`. It is because `redirfs_filter` type is defined as a void pointer and integer error value is returned as a pointer value. Kernel uses those functions to return pointer or error in the same return value.

ERRORS

`-EEXIST` – Other filter with same priority is already registered.

-EINVAL – Invalid filter name.

-ENOMEM – Out of kernel memory.

`int redirfs_unregister_filter(redirfs_filter filter)`

ARGUMENTS

filter – Filter handler.

RETURN VALUE

On success, zero is returned. On error, negative value is returned.

ERRORS

-EINVAL – Invalid filter handler.

`void redirfs_activate_filter(redirfs_filter filter)`

ARGUMENTS

filter – Filter handler.

`void redirfs_deactivate_filter(redirfs_filter filter)`

ARGUMENTS

filter – Filter handler.

```
int redirfs_set_operations(redirfs_filter filter,
struct redirfs_op_t ops[])
```

ARGUMENTS

filter – Filter handler.

ops – Pointer to the array of redirfs_op_t objects.

RETURN VALUE

On success, zero is returned. On error, negative value is returned.

ERRORS

-EINVAL – Invalid filter handler or operations array.

`int redirfs_remove_operations(redirfs_filter filter, struct redirfs_op_t ops[])`

ARGUMENTS

`filter` – Filter handler.

`ops` – Pointer to the array of `redirfs_op_t` objects.

RETURN VALUE

On success, zero is returned. On error, negative value is returned.

ERRORS

`-EINVAL` – Invalid filter handler or operations array.

`int redirfs_include_path(redirfs_filter filter, const char *path)`

ARGUMENTS

`filter` – Filter handler.

`path` – Directory subtree path.

RETURN VALUE

On success, zero is returned. On error, negative value is returned.

ERRORS

`-EINVAL` – Invalid filter handler or path name.

`-ENOMEM` – Out of kernel memory.

`-ENOTDIR` – Path name is not directory.

`int redirfs_exclude_path(redirfs_filter filter, const char *path)`

ARGUMENTS

`filter` – Filter handler.

`path` – Directory subtree path.

RETURN VALUE

On success, zero is returned. On error, negative value is returned.

ERRORS

`-EINVAL` – Invalid filter handler or path name.

`-ENOMEM` – Out of kernel memory.

`-ENOTDIR` – Path name is not directory.

Chapter 7

Dummy Filter

Here is an example of a Dummy Filter. It registers only one pre callback function for regular file permission function. This function is called by the VFS every time when user space process wants to open or execute a regular file. Directory subtrees over which will be this Filter's callback function called are `/bin` and `/usr/bin`. The Dummy Filter only prints the dentry name of an accessed file every time the file is opened or executed.

7.1 Sample Code

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/dcache.h>
#include <linux/namei.h>
#include "redirfs.h"


/*
   Filter's handler
 */
static redirfs_filter dummyflt_filter;

/*
   Filter's callback functions
 */
static enum redirfs_retv dummyflt_pre_permission(redirfs_context context,
                struct redirfs_args_t *args)
{
        const char *name;


        name= args->args.i_permission.nd->dentry->d_name.name;
```

```
        printk(KERN_ALERT "dummyflt: permission '%s'\n", name);
        return REDIRFS_RETV_CONTINUE;
}

/*
   Filter's callback functions for registration
 */
static struct redirfs_op_t dummyflt_ops[] = {
        {REDIRFS_I_REG, REDIRFS_IOP_PERMISSION, dummyflt_pre_permission, NULL},
        REDIRFS_OP_END
};

static int __init dummyflt_init(void)
{
        int retval = 0;
        int priority = 999;
        unsigned long flags = 0;

        /*
           Dummy Filter registration to the Redirfs Framework
         */
        dummyflt_filter = redirfs_register_filter("dummyflt", priority, flags);
        retval = PTR_ERR(dummyflt_filter);
        if (IS_ERR(dummyflt_filter))
                goto exit;

        /*
           Set Filter's callback functions.
         */
        retval = redirfs_set_operations(dummyflt_filter, dummyflt_ops);
        if (retval)
                goto unregister;

        /*
           Set Filter's directory subtrees
         */
        retval = redirfs_include_path(dummyflt_filter, "/bin");
        if (retval)
                goto unregister;

        retval = redirfs_include_path(dummyflt_filter, "/usr/bin");
        if (retval)
                goto unregister;

        redirfs_activate_filter(dummyflt_filter);
```

```
        return 0;

unregister:
        redirfs_unregister_filter(dummyflt_filter);
exit:
        return retval;
}

static void __exit dummyflt_exit(void)
{
        redirfs_unregister_filter(dummyflt_filter);
}

module_init(dummyflt_init);
module_exit(dummyflt_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_VERSION("v0.001");
MODULE_DESCRIPTION("Dummy Filter");
```

Chapter 8

AVG Anti-Virus Filter

The AVG Anti-Virus Filter or avgflt uses the RedirFS Framework for on-access scanning. It is based on dummyflt which has been introduced in chapter 7. The whole on-access scanning consists of three parts – avgflt kernel module registered to the Redirfs Framework, user space script Avghelper which ask the AVG Anti–Virus daemon to check the accessed file and the AVG Anti–Virus daemon.

8.1 Avgflt

Avgflt registers in the Redirfs Framework pre permission callback function. This function is called before a process opens or executes a file in specified directory subtree. Avgflt interact with Avghelper through `call_usermodehelper()` function, defined in `kernel/kmod.c`. This function allows to call userspace program within Linux kernel. It takes four arguments. Path for the application, null-terminated argument list, null-terminated environment list and wait flag. Wait flag specifies if the kernel should wait for the application to finish and return status or to continue. Avgflt waits until the Avghelper returns and then allows or disallows access to the file.

Avgflt module takes two arguments which have to be specified when the module is inserted to the Linux kernel through `insmod` or `modprobe` command. First `omit_pid` argument specifies pid of process which will be omitted from controlling. In this case it is pid of AVG daemon process. Second `cmd_path` specifies path to the application in user space which will be called by Avgflt. In this case it is the Avghelper script.

8.2 Userspace script – Avghelper

Avghelper receives from Avgflt full path to the file which should be scanned. It contacts the AVG daemon and waits for the scanning result. If the scanned file is not infected it returns back to the Avgflt positive value, otherwise zero or negative value is returned.

8.3 AVG Anti–Virus Daemon

The AVG Anti–Virus Daemon by default listens at localhost on port 55555. It provides TCP/IP socket interface to the AVG scanning kernel.

Chapter 9

Epilogue

This thesis describes the problems and limitations of the existing solutions for filesystem control access. The only solution supported directly by the Linux kernel is the LSM framework. It was integrated into the Linux kernel 2.6 and it provides a set of callback functions which can be registered by the security modules. The LSM is intended as a general framework which provides a way to control access to all Linux kernel structures. It does not enhance Linux kernel security by itself. It just provides an interface for security modules. The LSM has a big problem with the stacking of security modules. Only one security module can be registered directly to the LSM framework. Other modules are always registered to the security module which was registered before it. If the security module is not able to register other modules then no other module can be used. Modules are called in the order in which they were registered. There is no way of changing the calling order. With this stacking approach it is also not possible to safely remove security modules which have registered other security modules.

Another method of filesystem access control is to use the overlay filesystem. The linux kernel doesn't provide any special support for overlay filesystems. The whole idea behind it is that the linux kernel allows the mounting of a filesystem over another already mounted filesystem. The original filesystem is covered by the new one. Overlay filesystem uses this feature. When the overlay filesystem is mounted it covers the original one and links its VFS objects with the VFS objects of the original filesystem. This means that the overlay filesystem duplicates all VFS objects (file, inode, dentry) created for the original filesystem. Overlay filesystems can be stacked and again, as in the LSM, there is no way of specifying the order of overlay filesystems and how to safely unmount them (overlay filesystem could be used by other overlay filesystem).

It is also possible to use a general access control framework (RSBAC, Medusa), but these projects are not integrated into the Linux kernel. This means that the kernel has to be patched and maintaining patches for different kernel versions is also difficult. Another special project for filesystem access control is the Dazuko. It is used mainly by anti–virus companies for on–access scanning. It uses several methods of interaction with the Linux kernel. For Linux kernels 2.2 and 2.4 it replaces the syscall table. This is generally not a good idea (rootkit detectors) and syscall function are too high (NFS kernel daemon). For Linux kernel 2.6 Dazuko uses the LSM framework whose problems were mentioned above. Moreover, the LSM framework does not provide any way to catch on close event which is very important for anti–virus applications. As a solution, Dazuko uses RSBAC framework.

At this moment Dazuko maintainer John Ogness tries to solve all these problems with the overlay filesystem.

I have proposed a completely new framework called Redirfs to overcome all the problems and limitations mentioned above. It creates a new layer between the Linux VFS and the native filesystems. It interacts directly with the VFS layer and replaces the VFS objects operations. It provides a well defined interface for third–party kernel modules called filters. Each filter can register pre and post callback function for each native filesystem operation. It can also specify directory sub–trees over which their callback functions will be called. Filters can even exclude directory sub–trees. Filters are called in a specific order defined by their priorities. Redirfs modifies only those VFS object operations for which there are registered Filter callback functions. This means that all other operations go directly to the native filesystem with absolutely no overhead. Redirfs also modifies only the VFS objects which have to be modified. It returns VFS objects operations back to the native filesystem operations as soon as possible (no filter needs them).

Base parts of the Redirfs framework were implemented. The current implementation has several limitations. Note that these limitations are not design limitations! Only that some features have not been implemented yet. The Redirfs framework is well defined and ready for all intended features. Here are some limitations that will be removed in the future. At the moment filters can only register callback functions for dentry, inode and file operations over regular files and directories. All submounted directories are ignored.

Some parts of the Redirfs framework need to be implemented in a more sophisticated way. For example, replacement of file objects operations is implemented in a very unsophisticated way. Redirfs goes through all of the created file objects and checks if the file object belongs to the specified directory sub-tree. This check is comparing the full filename path with the directory path, which is too slow. This and several other implementation parts should be improved and optimized. This could be work for future projects, as well as some other user-space tools for filters, the interface for Redirfs in the /proc filesystem, the Redirfs port to BSD systems, and implementation of several filters.

The current implementation was tested on Linux kernels 2.6.10 and 2.6.11.3 and several filters were written for testing, including avgflt. Redirfs works without problems on uniprocessor systems without preemptive kernel. On SMP systems and preemptive kernels there is a problem with correct locking. This will be fixed.

Redirfs, with the help of GRISOFT, s.r.o, will continue as an open source project.

Bibliography

[1] Bovet, P. B., Cesati, M.: Understanding the Linux Kernel. 2, U.S.A., O'Reilly 2002

[2] Solomon, D. A., Russinovich, M. E.: Inside Microsoft Windows 2000. 3, Redmond, Washington, Microsoft Press 2000

[3] Rajeev, Nagar: Windows NT File System Internals, U.S.A, O'Reilly 1997

[4] Brown, M., Christiansen, N., Dewey, B., Olsen, D., Pudipeddi, R. Thind, R.: File System Filter Manager Technical Preview, 2002

[5] Smalley, S., Fraser, T., Vance, Ch.: Linux Security Modules: General Security Hooks for Linux. Document available at URL http://lsm.immunix.org/docs/overview/linuxsecuritymodule.html (Jan. 2005)

[6] Wrigh, Ch., Cowan, C., Morris, J., Smalley, S., Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux Kernel. Document available at URL http://lsm.immunix.org/docs/lsm-usenix-2002/html/ (Jan. 2005)

[7] Microsoft, Filter Manager Overview. Document available at URL http://download.microsoft.com/download/f/0/5/f05a42ce-575b-4c60-82d6-208d3754b2d6/Overview-May04.ppt (Jan. 2005)

[8] Love, R.: Linux Kernel Development. 2, Indianapolis, Indiana, Novel Press 2005

[9] McKusick, M. K., Neville-Neil G. V: The Design and Implementation of the FreeBSD Operating System, U.S.A, Addison-Wesley 2005