Extending File Permission Granularity for Linux

A dissertation submitted for the Degree of Master of Science in Information Security

P. M. Wanigasinghe
University of Colombo School of Computing
2018

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Students Name:

____

Signature: Date:

This is to certify that this thesis is based on the work of

Mr./Ms.

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name:

____

Signature: Date:

Abstract

In the recent years, threats against intellectual properties, unauthorized information disclosures and data breaches have been raised high. Most of the above attacks initiated within the network, with the help of internal employees. Whether intentionally malicious or unintentionally hazardous, are by far the greater problem in online security. Therefore for system administrators and authorize parties, another level of security definition is needed to secure important files and documents while continuing business as it was.

In Linux, traditional file system uses READ, WRITE and EXECUTE permissions over individual users and groups in order to control the file access and file operations. Although this has been the de-facto methodology, it has some disadvantages when it comes to highly secure and sensitive environments like banks and military. There is no straight forward way of defining permissions in operational aspect.

The goal of this project is to increase the granularity levels of defining permissions of the Linux file system going beyond traditional read, write and execute permissions. It consists of mainly two modules. In meta header module, authorized user can define what, who and how things are accessible related to files. He can define on what operations needs to restrict on what devices, locations and domains. The other module is responsible for handling the file operations and restrict according to the defined permissions.

Evaluation carried out on security, performance and usability aspects. There may have slight performance impact but it's negligible compared to the requirements of security. Further, this implementation can extends go beyond ext4 and apply into main kernel development.

Acknowledgement

I would like to express my special appreciation and thanks to my supervisor Dr. Kadun De Zoysa, who has been a tremendous mentor for me. I would like to thank you for the motivation and guidance provided to me throughout the endeavor, allowing me to successfully finish the project. Your advice on every aspect of the project has been priceless.

My special thank goes to UCSC for providing me with such an opportunity to improve my academic knowledge and implementation skills and for all the lecturers in the panel for sharing their invaluable time and knowledge to push us towards the degree. Also I would like to thank the technical, non-academic staff in UCSC for all the help given to us throughout the research period and prior.

I would like express my special gratitude towards my colleagues Sithum Nissanka, Charith Kulathilaka, Pathum Athukorala who helped me throughout this period in various ways to make this project a success. I would also like to give my thanks to Danuka Geeganage and many other friends who supported me throughout this endeavor make it successful.

I would also like to give my thanks to various researchers and online community contributors which I referred and gathered knowledge throughout the project.

Last but not least words cannot express the gratitude I have for my parents, wife and her parents for their support kindness and understanding throughout this whole time.

Contents

List of Figures

List of Tables

Abbreviations

ACL	Access Control List
AD	Active Directory
API	Application Protocol Interface
CD	Compact Disk
CPU	Central Processing Unit
FGP	Fine Grained Permissions
GUI	Graphical User Interface
HPFS	High Performance File System
IBM	International Business Machines
IP	Internet Protocol
IP	Intellectual Properties
IRM	Information Rights Management
ISO	International Standardization Organization
ISV	Independent Software Vendors
LKM	Loadable Kernel Module
OS	Operating System
PDF	Portable Document Format
RWE	Read, Write, Execute
SCP	Secure copy
SELinux	Security Enhanced Linux
USB	Universal Serial Bus

CHAPTER 1

Introduction

1.1 Problem domain

One of the main weapons in modern economical warfare is information about competitors, their offerings, finances, sales and marketing strategies. To fight in the market and to grab the market share, competitive intelligence is used. That is legally gathered information by closely monitoring all public channels, associated with competitors, networking with industry insiders and analyzing latest trends.

However, since those are not enough. Many turn to much shadier tactics in order to gain advantage over competition. Industrial espionage is much more common. [1]

The main goal of industrial espionage is to get a hold of protected confidential information belonging to competitors. [2] Usual targets of such actions are:

Trade secrets Generally means protected information about existing products or products in development. This is one of the most popular targets of industrial espionage. This information may help rival companies to increase competitiveness of their products or even bring a similar product to the market faster than original.

Client information Private data of clients, including their financial information, can be used to steal business, or can be leaked to damage the reputation of the company.

Financial information Financial information of the company can be used to offer better deals to clients and partners, win bids, or even make a better offer to valuable employees.

Marketing information This will allow competitors to prepare a timely answer for your marketing campaign, which, in turn, may render it completely ineffective.

Malicious insiders are one of the main key factors for industrial espionage. Competitors frequently place their employees who will act as a regular employees, while secretly gathering intelligence for their actual employer. They can approach trusted employees with privileged access to trade secrets and other valuable information. [3]

Unintentionally trusted employees can also perform or aid in corporate espionage. Various social engineering techniques can be used to gather secret information or extract credentials from employees. Random USB stick, left in a hallway for curious employee to pick up and use, or carefully written email that prompts to click on a link, are only two of a large number of ways through which malware can infect your system, giving your competitors full access to sensitive data.

Terminated employees are another source of danger. Disgruntled employee looking for a way to get back at company, or simply one of the trusted insiders leaving for a competition could easily take sensitive data with them.

So the insider threats has been considered as a main security threat in last year 2017. Lots of data breaches, information disclosures, IP theft reported last few years. Its very difficult to handle insider threats.

In order to deal with above situations, there is a need of defining security permissions in to another level, especially on files. File systems in Linux, and all Unix like file systems, supports file security based on RWE on top of users and groups. By it's design, permissions can define based on users aspect. But defining permissions on operational aspect, keeping existing permission structure as it is, can drastically increase the granularity of defining permissions.

Providing file system security in operational aspect in a more granular way is a lacking feature in Linux. All most all file systems security features focusing on user perspective rather than operational perspective. Implementing operational level permissions increase the granularity of defining permissions and the file owner has more control over the file. There are number of questions asked in Linux forums and other forums regarding increasing granularity levels. But there's no direct way of providing this. [4] [5] [6]

In NTFS file system in Windows, Microsoft Office 2007 system has granular permission defining system called IRM (Information Rights Management) [7] using AD. It allows individuals and administrators to specify access permissions to documents, and it helps to prevent sensitive information being printed, forwarded or copied by unauthorized people. Also there are third party applications like Safeguard PDF security by Locklizard, which follows the same concept to protect PDF files. But similar feature is not available in Linux , in OS level directly.

1.2 Limitations in current file permissions

Increasing granularity levels of file permissions in Linux is highly required in secure and sensitive environments like banks and military. They are using third party systems like Safe guard PDF security, AD RMS like systems in order to secure their files. Due to increase of internal attacks and specially unauthorized information disclosure, such feature inbuilt in to the OS is more useful for the system administrators to protect important files like configuration files and authorized persons to protect files like business sensitive files. But currently, Linux file systems (like ext / hpfs/ jfs etc) allow only a basic combination of permission levels.

Standard Linux file system's security provides permission levels, READ (R), WRITE (W) and EXECUTE (E).

READ User's capability to read the contents of a file.

WRITE User's capability to write or modify a file or directory.

EXECUTE User's capability to execute a file or view the contents of a directory.

And each file (and directory) has three user based permission groups:

Owner Apply only to the owner of the file, they will not impact the actions of other users.

Group Apply only to the group that has been assigned to the file or directory, they will not effect the actions of other users.

Others Apply to all other users on the system. [8]

Combination of file permission levels and permission groups became a standard and provides control to the owner (or authorized user) to maintain a security of a file. Since the design of Linux system focused primarily on providing generalized solution, go beyond this combination is not supported by default.

Apart from the standard file permissions, a user can use ACL [9] or SELinux [10] like approaches in order to increase security. But those are focuses on user perspective rather than operations perspective which describes in section 3.1.

To give more restrictions to a file in operational perspective, there are alternative approaches available, in which most of them are not directly associated with files itself and are not centralized. Those alternative solutions are not meant to handle in file level. Most of the times, it applies to the entire system. Therefore it's very difficult to configure and manage while continuing the business. Also sometimes it leads to security breaches where vulnerabilities found in the solution itself.

For example, if file owner wants to restrict a file being transferred/copied via USB drive, either he wants to block USB drives for the entire system or he wants to restrict write permission for a particular user or group. There's no granularity level in between to specify permissions such as, restrict copying file into USB drives for only a particular user.

Another example would be, if user wants to restrict transferring file via SCP to certain IP addresses, he needs to have another alternative solution to restrict those IP addresses. Doing above approaches not only affect to that particular file only, but also it applies to the entire file system.

1.3 Motivation

Motivation of this project is to increase the security of the Linux file system in OS level itself rather than using third party software. This project extends the granularity levels of defining permissions of a file. It goes beyond RWE permissions and users will be able to specify what actions and what limitations will be provided with a file itself.

1.4 Aims and objectives

In this section objectives of this project is explained in detail. One of the most important parts of this project is to do a comprehensive study of existing work done.

- Build a file system that provides more granularity, on which user can define permissions on his own choice according to a pre-defined structure.

  In current Linux, user will be able to provide permissions for file or directory using existing READ, WRITE, EXECUTE and based on USER, GROUP and ALL, But using this implementation, user will be able to define permissions in more granular way in operations perspective. He will be able to control more by defining what operations need to restrict based on what parameters.

- File will have extra security meta header which defines the parameters.

  To achieve above, user will be able to define actions and related parameters in a friendly manner. System will provide pre-defined structure of defining actions and parameters.

- Build an interceptor module, which intercepts each operation and check against permissions define in the header.

  To validate defined actions and parameters, there will be an interceptor module, where it centrally operates and check executing operations against them.

1.5 Scope and limitation

- This implementation applies only to files. But it can further extends to directory level. Since Linux treats everything as a file, such expansion is not hard.
- Further this implementation can extends when the file is transferred, the security meta header should transfer along with file with change of ownership. Since the meta header associated with inode, it will automatically transfer when file get transferred. But new namespace, which introduced in this project needs to be in the new system in order to handle the permissions.
- In order to prove the concept, cp, scp and lpr commands will use in this project. As a future work, this implementation can extends easily in order to work all the file related operations which uses 'open' system call internally.
- This implementation only focuses on Linux kernel 4.4 and this will suggest to the kernel development to bind this into a core kernel to avoid conflicts in system upgrades.
- This only applicable for ext4 file system and this can be further extends to other file systems as well.
- This implementation works on the kernel versions greater than 2.6 kernels and this should be changed in order to apply it for older kernels. This is because mainly due to the more restrictions introduced to the system table after kernel version 2.6.
- This implementation is limited to file operations which are called 'open' system call internally.

1.6 Report Overview

Chapter 2 provides a literature review on how Linux and Linux file systems works. It also includes systems that tries to resolve the same issue and how and what level they have achieved it.

Chapter 3 provides the design of the system. It describes about the two main modules, Meta Header module and Interceptor module, and the design approach.

Chapter 4 provides an comprehensive description about the implementation of the mechanism.

Chapter 5 provides a critical evaluation of the research by performing various tests and obtaining and plotting those results in to graphs.

Chapter 6 provides a conclusion to this research by providing the motivation, what was done in the research, challenges faced, future work and status of the final outcome

CHAPTER 2

Literature review

This chapter will discuss about the current Linux file system, underline structures, security and related OS features. Further this will discuss about similar approaches.

2.1 Linux file system

2.1.1 File system in general

Without a file system, information gathered and stored in a storage medium like hard drive, would be a one large block of data. So there is no way to tell where and how one piece of information start and stop. Therefore in computers, a file system is used to define and control how data is stored and retrieved. Each group of data is called 'file' and it has attributes like name, id to identify and isolate that particular group of information separately from others. File system's responsibility is to manage the structure and logic rules used by the file. [11]

2.1.2 File system in Linux

On a Linux system, everything is a file. If something is not a file, it is a process. There's no difference between a file and a directory, since a directory is just a file containing names of other files. In order to manage all those files in an orderly fashion, ordered tree like structure is used.

A file system is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be meaningless. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. [12]

Individual drive partitions can be setup using one of the many different available file systems. Each has its own advantages, disadvantages. Some of the file systems supported by Linux are, ext, ext2, ext3, ext4, hpfs, iso9660, JFS, minix, msdos, ncpfs nfs, ntfs, proc, Reiserfs, smb, sysv, umsdos, vfat, XFS, xiafs. [13]

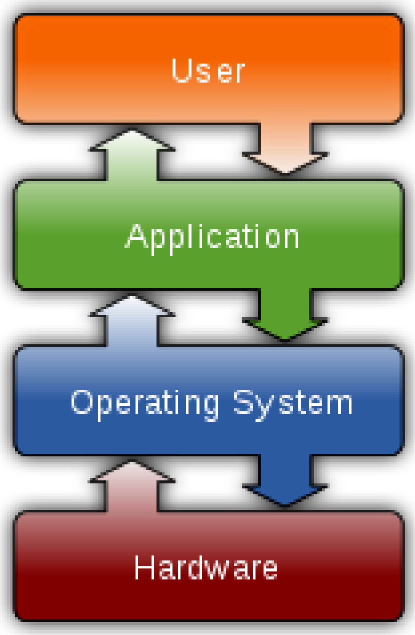


Figure 2.1: File System

To switch or use one file system, you need to insert corresponding module or recompile the kernel and mount it.

Below a short description of the available file systems in the Linux kernel.

ext is an elaborate extension of the minix file system. It has been completely superseded by the second version of the extended file system (ext2) and has been removed from the kernel (in 2.1.21).

ext2 is the high performance disk file system used by Linux for fixed disks as well as removable media. The second extended file system was designed as an extension of the extended file system (ext).

ext3 is a journaling version of the ext2 file system. It is easy to switch back and forth between ext2 and ext3.

ext4 is a set of upgrades to ext3 including substantial performance and reliability enhancements, plus large increases in volume, file, and directory size limits.

hpfs is the High Performance File system, used in OS/2. This file system is read-only under Linux due to the lack of available documentation.

iso9660 is a CD-ROM file system type conforming to the ISO 9660 standard.

High Sierra Linux supports High Sierra, the precursor to the ISO 9660 standard for CD-ROM file systems. It is automatically recognized within the iso9660 file system support under Linux.

Rock Ridge Linux also supports the System Use Sharing Protocol records specified by the Rock Ridge Interchange Protocol.

JFS is a journaling file system, developed by IBM, that was integrated into Linux in kernel 2.4.24.

For convenience, the Linux file system is usually thought of in a tree structure. On a standard Linux system you will find the layout generally follows the scheme presented below.

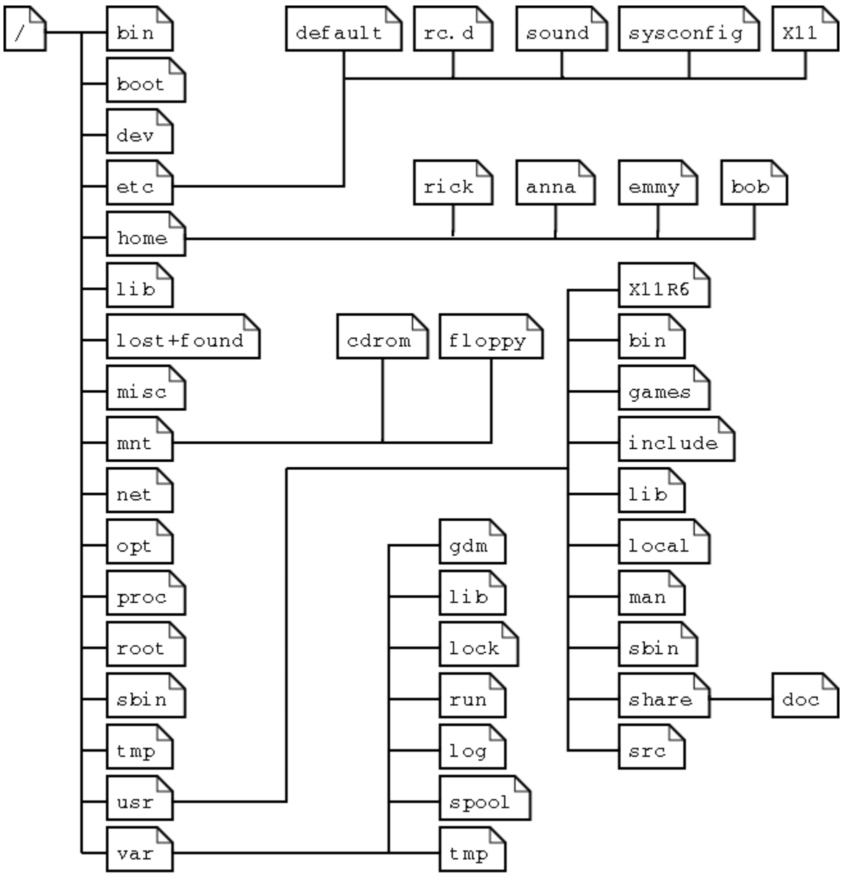


Figure 2.2: Linux file system layout

2.1.3 Ext4

Ext 4 has more improvements compared to Ext3 file system. It has improved design, better performance, reliability, capacity and features. Most Linux distributions use Ext4. Ext3 was mostly about adding journaling to Ext2, but Ext4 modifies important data structures of the file system such as the ones destined to store the file data. [14]

Metadata and journal checksums were added to improve reliability. To meet various mission-critical requirements, the filesystem timestamps were improved with the addition of intervals down to nanoseconds.

In EXT4, data allocation was changed from fixed blocks to extents (described by its starting and ending place on the hard drive). So it's possible to describe very long, physically contiguous files in a single inode pointer entry, which can significantly reduce the number of pointers required to describe the location of all the data in larger files.

EXT4 reduces fragmentation by scattering newly created files across the disk so that they are not bunched up in one location at the beginning of the disk.

Aside from the actual location of the data on the disk, EXT4 uses functional strategies, such as delayed allocation, to allow the filesystem to collect all the data being written to the disk before allocating space to it. This can improve the likelihood that the data space will be contiguous.

2.1.4 Inode

An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data. It is a key component of the metadata in EXT file systems. Access to a file is via the directory entry, which itself is the name of the file and contains a pointer to the inode. The value of that pointer is the inode number. Each inode in a filesystem has a unique ID number.

A data structure is a way of storing data so that it can be used efficiently. Different types of data structures are suited to different types of applications, and some are highly specialized for specific types of tasks.

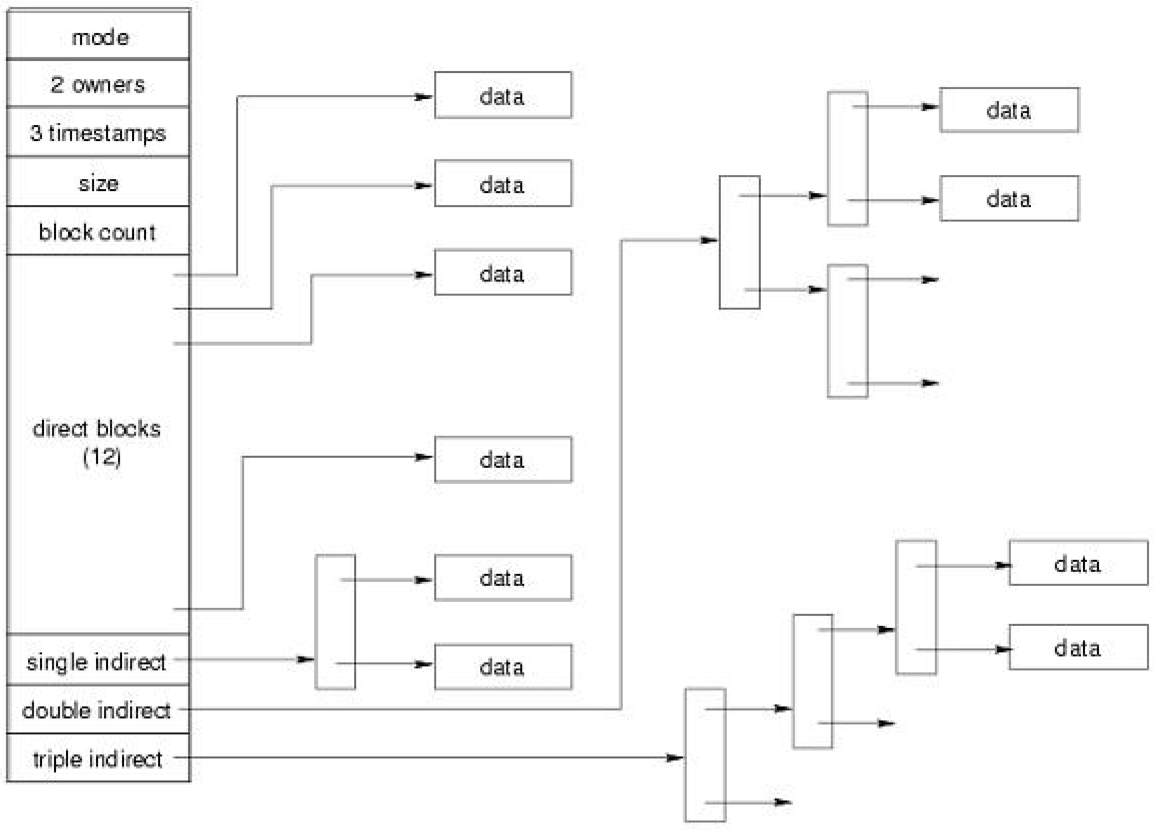


Figure 2.3: Inode structure

2.2 Security in Linux File Systems

The Linux security model is based on the one used on UNIX systems which is quite robust. On a Linux system, every file is owned by a user and a group user. There is also a third category of users, those that are not the user owner and don't belong to the group owning the file. For each category of users, read, write and execute permissions can be granted or denied. [15]

ls -l displays file permissions for these three user categories; they are indicated by the nine characters that follow the first character, which is the file type indicator at the beginning of the file properties line. First three characters in this series of nine display access rights for the actual user that owns the file. The next three are for the group owner of the file, the last three for other users. The permissions are always in the same order: read, write, execute for the user, the group and the others.

2.3 Extended Attributes in Linux

Extended attributes are additional meta data for a file, which are not interpreted by the file system. In Linux, the ext2, ext3, ext4, JFS, Squashfs, Yaffs2, ReiserFS, XFS, Btrfs, OrangeFS, etc file systems support extended attributes. It consists of name and associated data. Name should be a null terminated string, prefixed by namespace. [16]

Currently four namespaces available,

user

trusted

security mostly use by SELINUX

system primarily use by ACL

The Linux kernel allows extended attribute to have names of up to 255 bytes and values of up to 64KiB, as do XFS and ReiserFS, but ext2/3/4 and btrfs impose much smaller limits, requiring all the attributes (names and values) of one file to fit in one "filesystem block" (usually 4 KiB). [17]

Extended attributes can be accessed and modified using the arrt, getfattr and setfattr commands from the attr package on most distributions.

2.4 Linux process

Processes carry out tasks within the operating system. It is a dynamic entity, constantly changing as the machine code instructions are executed by the processor. As well as the program's instructions and data, the process also includes the program counter and all of the CPU's registers as well as the process stacks containing temporary data such as routine parameters, return addresses and saved variables. [18]

The current executing program, or process, includes all of the current activity in the microprocessor. Linux is a multiprocessing operating system. Processes are separate tasks each with their own rights and responsibilities. If one process crashes it will not cause another process in the system to crash. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel managed mechanisms.

During the lifetime of a process it will use many system resources. It will use the CPUs in the system to run its instructions and the system's physical memory to hold it and its data. It will open and use files within the file systems and may directly or indirectly use the physical devices in the system. So that Linux can manage the processes in the system, each process is represented by a task_struct data structure. The task vector is an array of pointers to every task_struct data structure in the system. To make it easy to find, the current, running, process is pointed to by the current pointer.

As well as the normal type of process, Linux supports real time processes. These processes have to react very quickly to external events and they are treated differently from normal user processes by the scheduler. Although the task_struct data structure is quite large and complex, but its fields can be divided into a number of functional areas.

2.4.1 State

As a process executes it changes state according to its circumstances. Linux processes have the following states:

Running The process is either running (it is the current process in the system) or it is ready to run (it is waiting to be assigned to one of the system's CPUs).

Waiting The process is waiting for an event or for a resource. Linux differentiates between two types of waiting process; interruptible and uninterruptible. Interruptible waiting processes can be interrupted by signals whereas uninterruptible waiting processes are waiting directly on hardware conditions and cannot be interrupted under any circumstances.

Stopped The process has been stopped, usually by receiving a signal. A process that is being debugged can be in a stopped state.

Zombie This is a halted process which, for some reason, still has a task_struct data structure in the task vector. It is what it sounds like, a dead process.

2.4.2 Identifiers

Every process in the system has a process identifier. The process identifier is not an index into the task vector, it is simply a number. Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system,

2.4.3 Links

In a Linux system no process is independent of any other process. Every process in the system, except the initial process has a parent process. New processes are not created, they are copied, or rather cloned from previous processes. Every task_struct representing a process keeps pointers to its parent process and to its siblings (those processes with the same parent process) as well as to its own child processes. You can see the family relationship between the running processes in a Linux system using the pstree command:

2.4.4 File system

Processes can open and close files as they wish and the processes task_struct contains pointers to descriptors for each open file as well as pointers to two VFS inodes. Each VFS inode uniquely describes a file or directory within a file system and also provides a uniform interface to the underlying file systems.

2.5 Evaluation methods

2.5.1 Throughput

Throughput is a measure of how many units of information a system can process in a given amount of time. It is applied broadly to systems ranging from various aspects of computer and network systems to organizations. Related measures of system productivity include , the speed with which some specific workload can be completed, and response time, the amount of time between a single interactive user request and receipt of the response.

2.5.2 CPU Utilization

CPU utilization refers to a computer's usage of processing resources, or the amount of work handled by a CPU. Actual CPU utilization varies depending on the amount and type of managed computing tasks. Certain tasks require heavy CPU time, while others require less because of non-CPU resource requirements.

Linux has also got set of utilities to monitor CPU utilization. With these commands you can find out total CPU utilization, individual CPU utilization, your system's average CPU utilization since the last reboot, determine which process is eating up your CPU(s) etc.

2.6 Similar Systems

2.6.1 AD RMS

AD RMS can be use to help prevent sensitive information—such as financial reports, product specifications, customer data, and confidential e-mail messages—from intentionally or accidentally getting into the wrong hands.

At high level, you can use organization's security strategy and persistent usage policies to protect information, no matter where it is moved by using Active Directory Rights Management Services (AD RMS) and the AD RMS client.

2.6.1.1 What is Active Directory Rights Management Services?

There are three main parts in AD RMS system,

- Windows Server 2008 running the Active Directory Rights Management Services
- (AD RMS) server role that handles certificates and licensing a database server
- AD RMS client. (included as part of the Windows 7 and Windows Vista operating systems)

2.6.1.2 Benefits to an organization

Safeguard sensitive information Users can define who can open, modify, print, forward, or take other actions with the information. Applications such as word processors, e-mail clients, and line-of-business applications can be AD RMS-enabled to help safeguard sensitive information. Organizations can create custom usage policy templates such as "confidential - read only" that can be applied directly to the information.

Persistent protection AD RMS augments existing perimeter-based security solutions, such as firewalls and access control lists (ACLs), for better information protection by locking the usage rights within the document itself, controlling how information is used even after it has been opened by intended recipients.

Flexible and customize technology Independent software vendors (ISVs) and developers can AD RMS-enable any application or enable other servers, such as content management systems or portal servers running on Windows or other operating systems, to work with AD RMS to help safeguard sensitive information. ISVs are enabled to integrate information protection into server-based solutions such as document and records management, e-mail gateways and archival systems, automated workflows, and content inspection.

2.6.2 Safeguard PDF Security by Locklizard

Another implementation similar to the concept, which is granular permission definition, is Safeguard PDF Security by Locklizard. This is a third party application and it enables user to restrict document access by locking PDF documents. So they can only be used by certain individuals on specific devices and from controlled locations. Restrictions can be given to,

- individual users
- a group of users
- all authorized users
- specific devices
- a location (range of IP addresses)
- a domain (specific IP address)

2.6.2.1 Restrict Document Access to authorized users

Safeguard PDF Security can be used to protect PDF files to individual users, all the users or group of users (for example, only the Sales department given access to sales reports). As their documents says,

All users Selecting this option grants document access to all users you have added to your Safeguard administration system.

Selected users Selecting this option grants document access to selected users. You select the users you want to grant document access to in the Safeguard administration system.

Users who have been given access to a publication (group of protected documents) Selecting this option grants document access to all users who have been granted access to a specific publication. Users are granted access to publications in the Safeguard administration system.

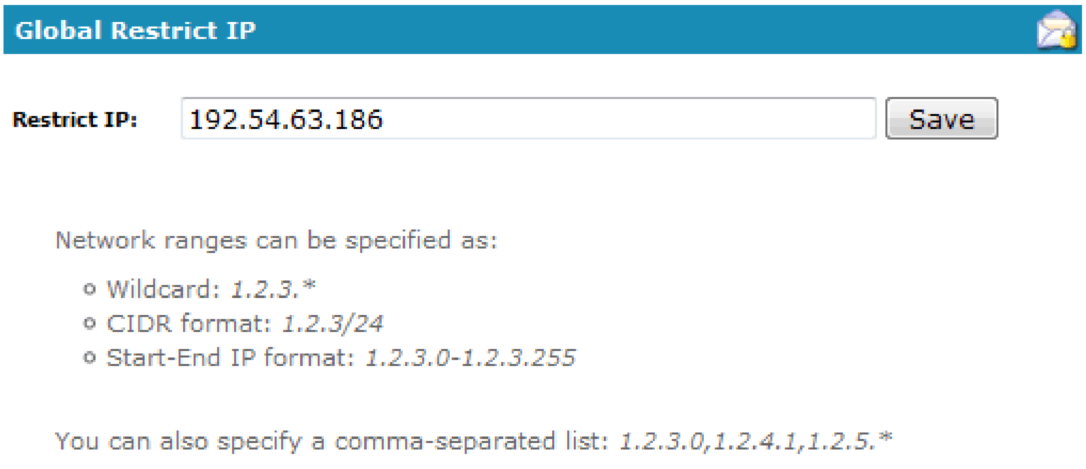
2.6.2.2 Restricting Document Access to locations and/or domains

Examples for restricting for domains are place of work or third party site etc. Restricting document access to a specific location ensures documents cannot be used outside that location and therefore minimizes confidential documents being compromised. This is especially useful if your workforce uses laptops or other mobile devices to access protected documents.

2.6.2.3 Locking PDF files to an IP address or range of IPs

Very similar to this implementation, By implementing a global IP address users can lock document access for all users to a domain/location.

In the Settings Tab in the Safeguard Admin system and select the 'Restrict IP' link. Enter an IP address or range of IP addresses you want to restrict document access to. On each user's account you can enter an IP address or a range of addresses from which a user can register their license from and also use a secure PDF document.


Global Restrict IP

Restrict IP: 192.54.63.186

Network ranges can be specified as:

- Wildcard: *1.2.3.**
- CIDR format: *1.2.3/24*
- Start-End IP format: *1.2.3.0-1.2.3.255*

You can also specify a comma-separated list: *1.2.3.0,1.2.4.1,1.2.5.**

Figure 2.4: Safegaurd security - IP restriction

If a user tries registering their license from an IP address outside this range then then they will not be able to register.

If a user registers their license on say a laptop from an approved IP address range (for example an office location) and then takes their laptop home, they will not be able to view your secure documents as long as the document controls are set to check with the administration server (verify document access = each time the document is opened).

2.6.2.4 Locking Documents to specific Devices

Safeguard PDF security can use to restrict access on some devices, such as devices with Windows OS or mobile devices. You can also restrict printing to just Windows or Macintosh devices. You can also stop documents from being accessed in thin client and virtual environments. This prevents multiple users accessing documents from a single system all at the same time.

Protected PDF files are locked to specific devices and cannot be moved to non-authorized devices – if a user copies or sends a protected document to a non-authorized device then the document will not open.

2.6.2.5 How Locklizard differentiate from this implementation

Locklizard's Safeguard PDF security is a application level system and designed specifically for PDF files. But this implementation works directly in a OS system kernel space providing more security by the design itself.

Also this implementation not limited to PDF files, Since it directly deals with inode, this applies to all kind of files and easily enhance to directories as well.

Feature	AD RMS	Locklizard	This Project
Implementation	External to the OS	External to the OS	Internal to the OS
Supported files	All	PDF	All
Integration	Easily integrated	Easily integrated	Not easily integrated

Table 2.1: Feature comparison

CHAPTER 3

Design

In this chapter, first two sections will discussed the design about main modules, meta header module and interceptor module in this project. Then the subsequent sections some approaches and design concerns will be discussed.

3.1 User perspective VS Operational perspective

Traditional Linux file permissions, by design, are based on user perspective. User can define file permissions (Read, Write and Execute) on top of owner, user group and all users. So the defined permissions acts on user or group of users, allowing or restricting on users point of view. For an example, file owner has Read, Write and Execute permissions while restricting some group only to Read.

Defining permissions on operational level increases the granularity of defining permissions. For an example, file owner can define what operations like copy, delete needs to restrict on what users for a particular file. It has more flexibility and security by design.

3.2 How to interrupt executed commands

Executed commands can be interrupt in several ways.

3.2.1 Function call hooking with LD_PRELOAD

Function call hooking refers to a range of techniques used to intercept calls to pre-existing functions and wrap around them to modify the function's behavior at runtime. [19]

function hooking in linux using the dynamic loader API, which allows us to dynamically load and execute calls from shared libraries on the system at runtime, and allows us to wrap around existing functions by making use of the LD_PRELOAD environment variable.

The LD_PRELOAD environment variable is used to specify a shared library that is to be loaded first by the loader. Loading our shared library first enables us to intercept function calls and using the dynamic loader API we can bind the originally intended function to a function pointer and pass the original arguments through it, effectively wrapping the function call. If you set LD_PRELOAD to the path of a shared object, that file will be loaded before any other library (including the C runtime, libc).

3.2.2 Inotify

The inotify API provides a mechanism for monitoring file system events. Inotify can be used to monitor individual files, or to monitor directories. When a directory is monitored, inotify will return events for the directory itself, and for files inside the directory. [[20]

Inotify was created by John McCutchan, and it was merged into the Linux kernel in kernel version 2.6.13, released on August 29, 2005. later kernel versions included further improvements. The required library interfaces were added into the GNU C library (glibc) in its version 2.4, released in March 2006, while the support for inotify was completed in glibc version 2.5, released in September 2006.

One major use is to permits re indexing of changed files without scanning the filesystem for changes every few minutes, which would be very inefficient. Inotify can also be used to automatically update directory views, reload configuration files, log changes, backup, synchronize, and upload.

3.2.3 Intercept system calls

If we want to intercept a system call, we have to write our own fake system call, then make the kernel call our fake function instead of the original call. At the end of our fake call, we can invoke the original call. In order to do this, we must manipulate the system call table array (sys_call_table). Which described in /usr/src/linux/arch/i386/kernel/entry.S (in i386 architecture). This file contains a list of all the system calls implemented within the kernel and their position within the sys_call_table array.

3.2.4 System calls and System table

The Linux kernel maintains a table of pointers that reference various functions made available to user space as a way of invoking privileged kernel functionality from unprivileged user space applications. These functions are collectively known as system calls. [21]

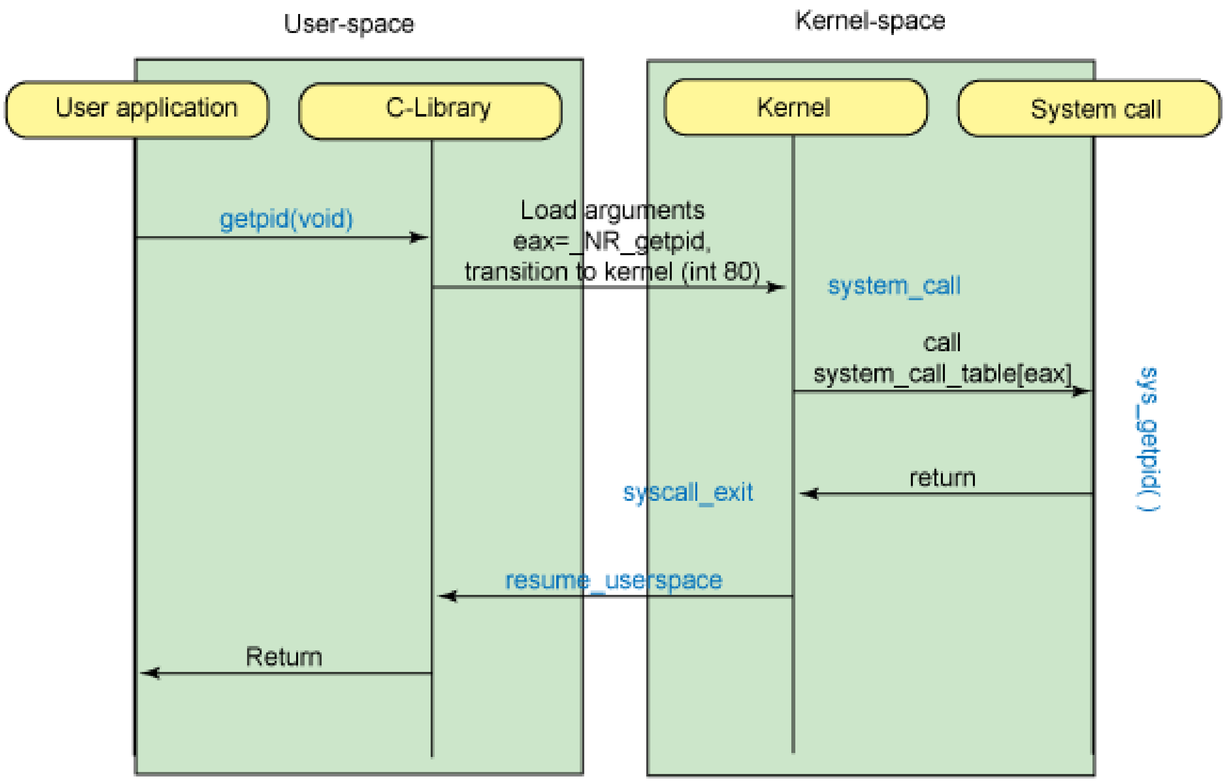


Figure 3.1: System call

Any legitimate software looking to hook kernel space functions should first consider using existing infrastructure designed for such uses like the Linux kernel tracepoints framework or the Linux security module framework.

3.3 Kernel module

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. A module can be configured as built-in or loadable. To dynamically load or remove a module, it has to be configured as a loadable module in the kernel configuration.

3.4 FGP Modules

This system designed to have mainly two modules. Both of the modules designed to run within the Linux kernel in kernel space.

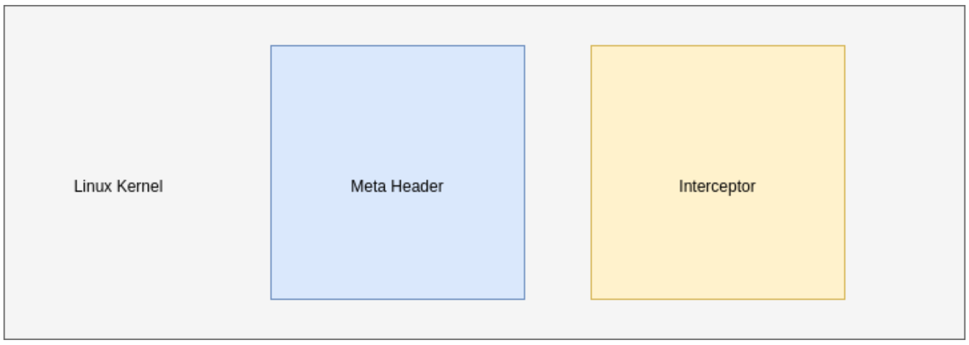


Figure 3.2: FGP Modules

3.5 Meta Header Module

Meta header module defines the permissions. When file owner creates a file, along with the file, system will creates a separate security meta header, in which file owner can specify the permissions in a defined way.

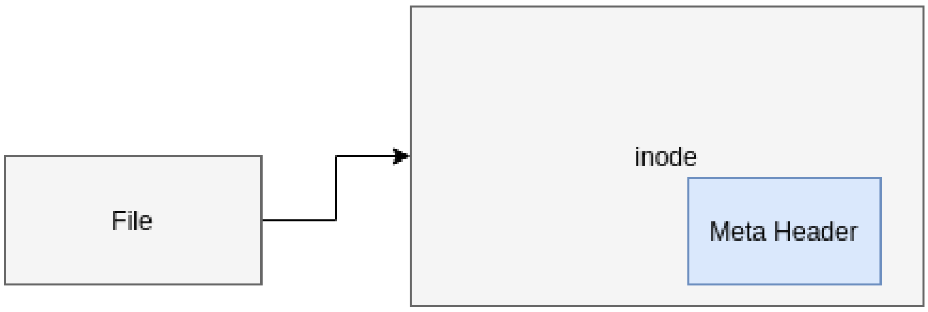


Figure 3.3: Meta Header

User will be able to add permissions in a more granular way as below,

command : <user/group> <type> <value>

In this way, file owner has more control over defining file permissions in operational aspect. He can specify the operation foremost and then followed by defining to whoem this applies, what is the type and related value. More description and examples available in implementation chapter

3.6 Interceptor Module

The primary objective of the interceptor module is to intercept the current running process and validate against defined permissions. This module will implement as a centralized module where every file system operation go through it.

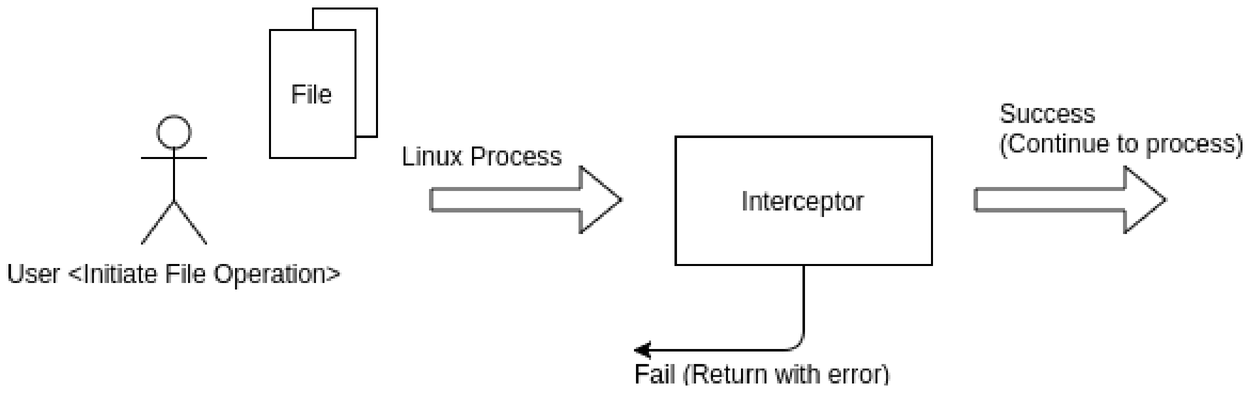


Figure 3.4: Interceptor

It will check whether the current running process is matched against the defined operations in the security meta header's access controllers. If it does, then it checks whether the performed action is allowed or not. If it's not allowed, the module will generate a error message and notify the user.

This module will implement as kernel module. LD_PRELOAD has not used because it will run in user space and has security issues. Also this module cannot implement as inotify since, inotify works after inode get changed. So the pre-valiadtion cannot be done.

3.7 Sub modules in interceptor module

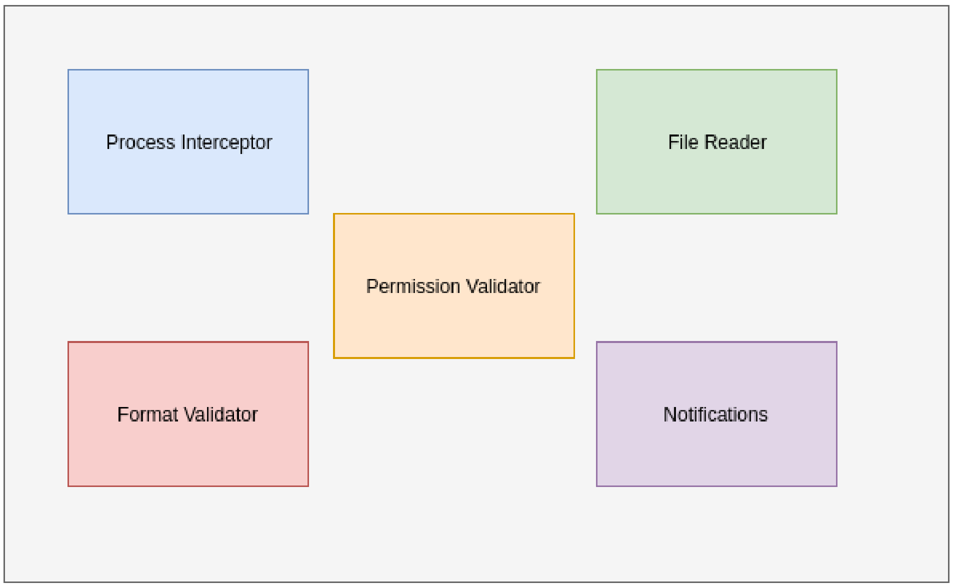


Figure 3.5: Sub modules

Interceptor module contains five sub modules. Process interceptor sub module intercepts the file system operations (open(), read(), write()) and though that it will take the current running process. After obtaining current process, file reader sub module will extract files attached to the current running process. Then the format validator sub module validates the permission format defined by the user using meta header module. If the format and values are correct, Permission validator sub module will take care of validating permissions against user entered command. Notification sub module is responsible for format the output according the required output. (whether it's terminal or GUI).

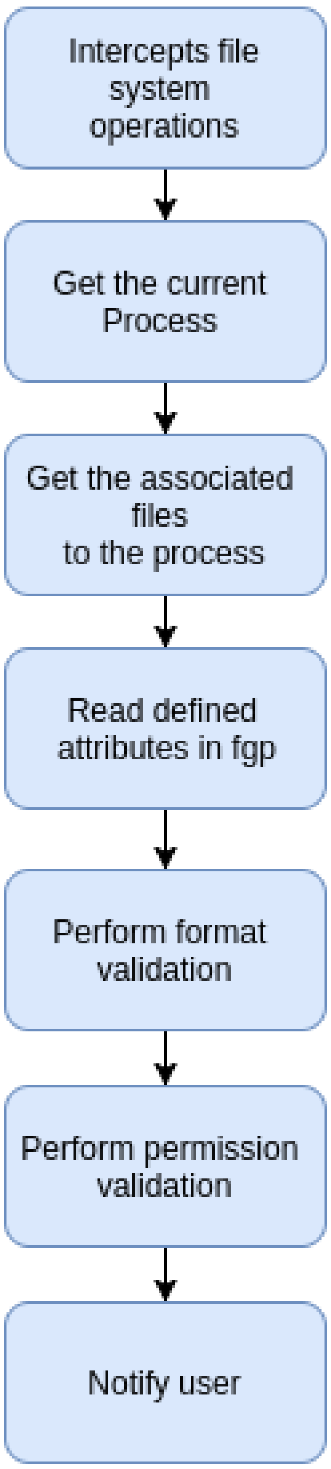

Intercepts file system operations
Get the current Process
Get the associated files to the process
Read defined attributes in fgp
Perform format validation
Perform permission validation
Notify user

Figure 3.6: Intercept process

Full activity flow of the design shows below. When file owner creates a file, meta header module will automatically creates a meta header. Then file owner can define standard permissions and extended permissions as per the requirement. Meta header module will save defined permissions in file's inode.

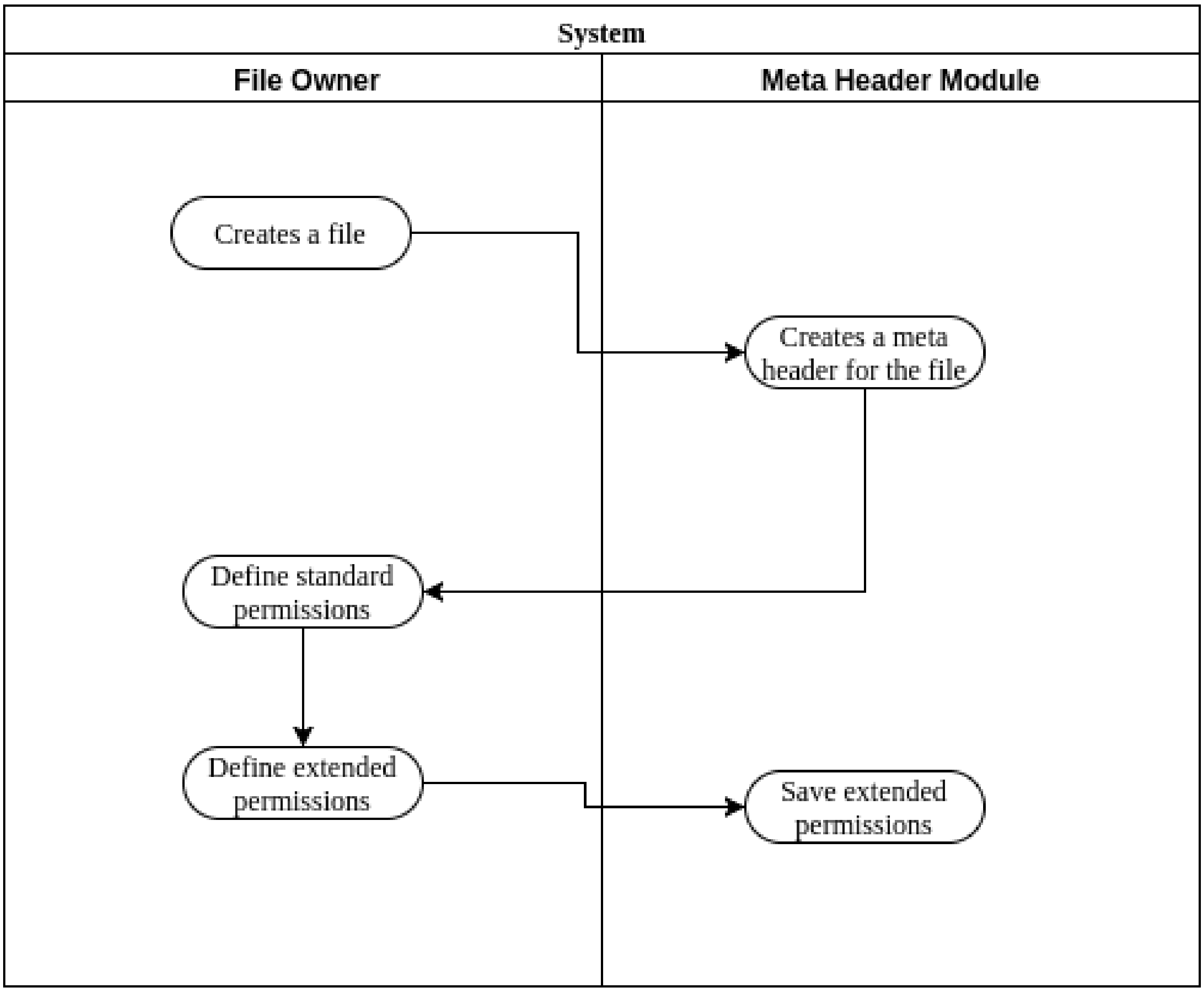

System
File Owner
Meta Header Module
Creates a file
Creates a meta header for the file
Define standard permissions
Define extended permissions
Save extended permissions

Figure 3.7: Flow - define permissions

When user tries to access the file (execute operation against that file), interceptor module will intercept the process and read defined permissions on that file. Then it validates permissions and decide whether to allow or notify restricting the operation.

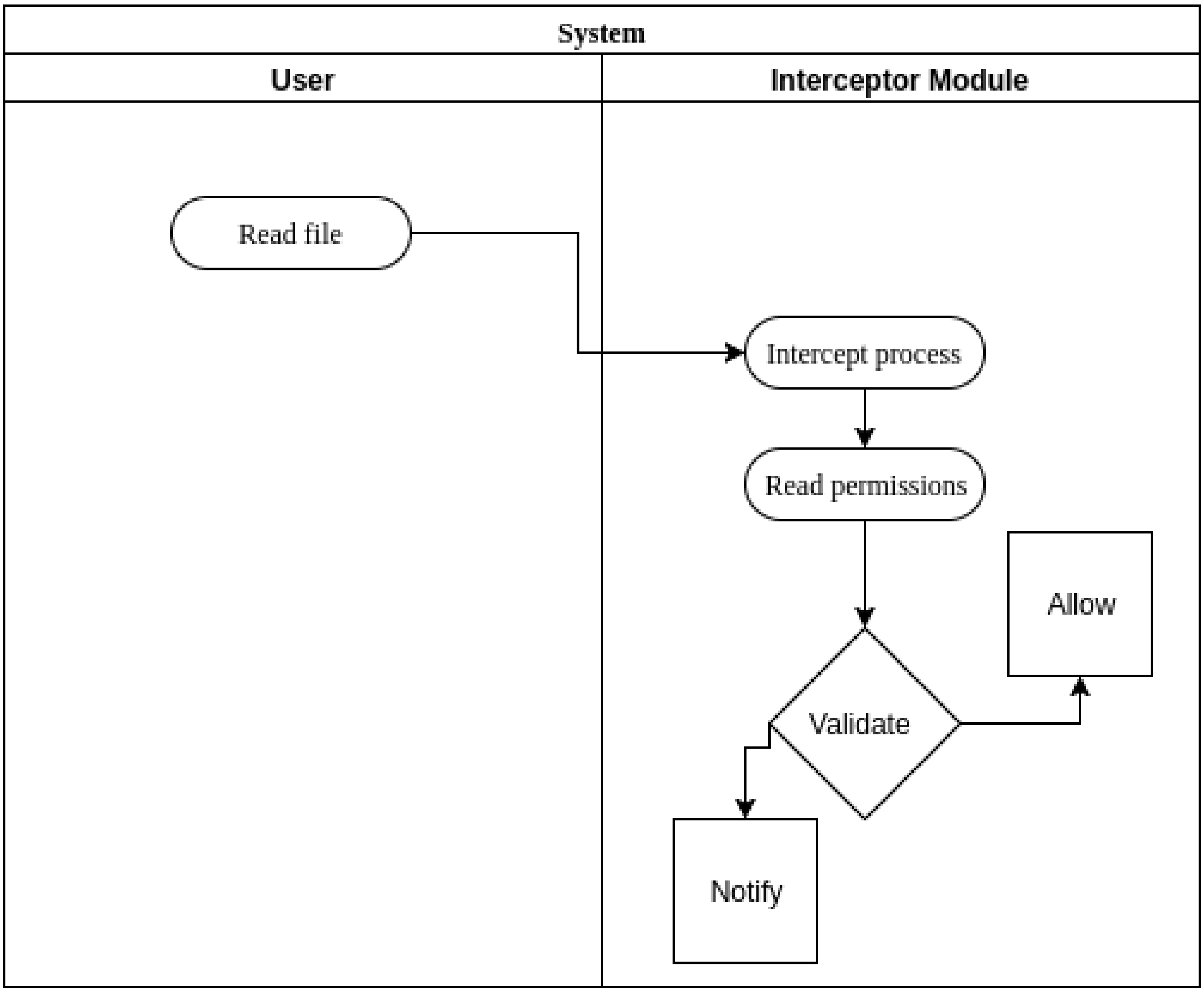


Figure 3.8: Flow - intercept process

3.8 Limitations of the design

One of the limitation of this design is, this will combine OS layers. For example, defined parameters can contain IP addresses, devices which will be used to restrict file operations. But defining network related features in a file (application level) can be considered as a design concern.

There are some application level programs who allowed to define such configurations in application level. For example, application server like Apache Tomcat allows to define/restrict certain IP addresses to access the server.

To overcome this kind of design limitations, there can be a separate policy module, which resides outside the system, can contain permissions defined as policies. This is a enhancement to this implementation and it will not cover in this scope.

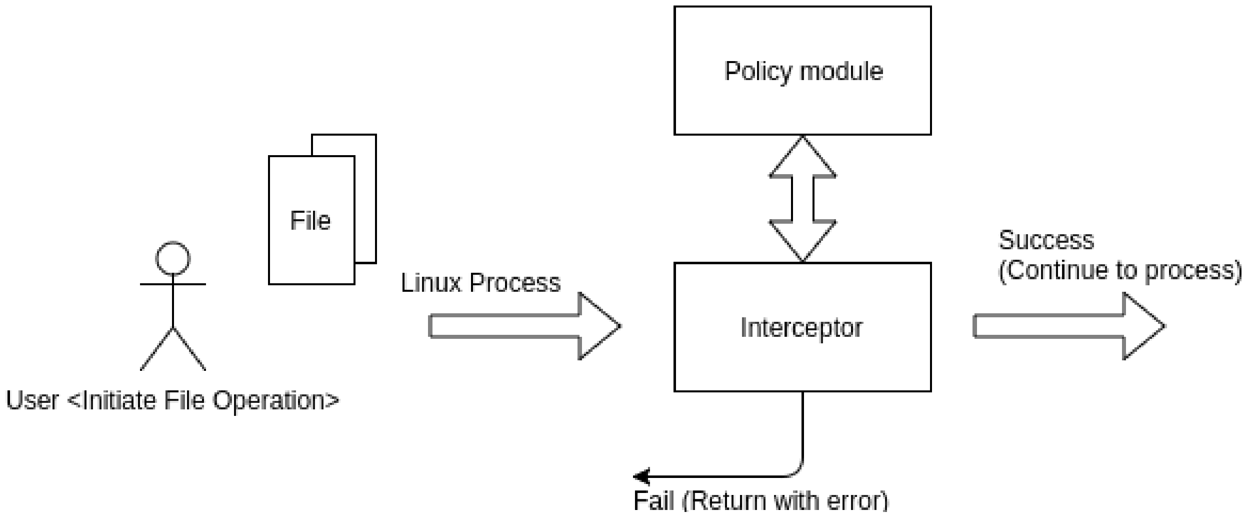


Figure 3.9: Separation of concerns

CHAPTER 4

Implementation

4.1 Meta header module implementation

As described in the previous chapter, under design of the meta header module, this module is responsible of defining the file related permissions in a more granular way. Extended attributes feature has been used to define these permissions and a new extended attribute called FGP has been introduced.

4.1.1 Extends Extended Attributes

To implement file system meta header, extended version of extended attributes has been used. Currently there are four namespaces available in standard extended attributes in Linux.

- user
- trusted
- security
- system

New namespace have introduced (fgp) in order to define permissions in this module. Therefore user can specify permissions in a simplified text based manner. Namespace is fgp (abbreviate for fine grained permissions). user can define as

fgp.<command>_<U/G: value>_<type>_<value>

ex: fgp.cp_U:prasad_NETWORK_192.168.11.12

fgp is the namespace. This should followed by the string, which describes the permission.

<command> Specify the operation user needs to restriction

<U/G: value> Specify the user or group needs to restricts from. U- User, G- Group followed by value. (* for all)

<type> Specify whether the destination type (NETWORK, DEVICE or DIRECTORY) Network can be a single IP or IP range.

User can use standard extended attributes set and get functions, setfattrt() and getfattr(). And user can define any number of records for a file within the file block size. (for ext4, must fit in a single filesystem block (1024, 2048 or 4096 bytes). If them are overridden, last defined entry has taken as a valid one.

4.2 Approach for a new namespace

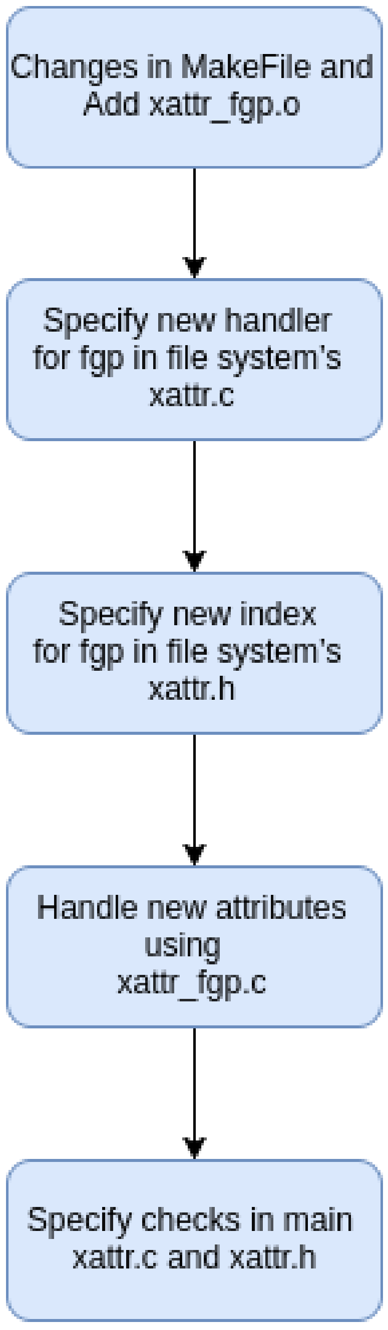

Changes in MakeFile and Add xattr_fgp.o
Specify new handler for fgp in file system's xattr.c
Specify new index for fgp in file system's xattr.h
Handle new attributes using xattr_fgp.c
Specify checks in main xattr.c and xattr.h

Figure 4.1: New namespace

4.3 Interceptor module implementation

As described in design, there are five sub modules in the interceptor module. Among those, process interceptor module is the main sub module and intercepting the current running file operation is the responsibility of it.

As described in the literature survey, three intercepting methods were analyzed and due to the applicability, process interception via system call interception has been selected.

4.3.1 System call interception

Intercept system call described in detail in Appendix D. Since Linux 2.6 > kernels not exposed system table, there are couple of ways to implement this.

4.3.1.1 Brute force method

To start brute forcing, system_utsname is selected as a staring address since it's known that system_utsname structure exists before the system call table. Therefore, iteration starts for system_ustname location and iterates advancing byte each time. On the iteration, function checks whether the current location is equal to sys_read, since the sys_read is available to LKM. So, if it is equal to sys_read, current location contains the system table.

```
while(i)
        {
                if(sys_table[__NR_read] == (unsigned long)
                   sys_read)
                {
                        sys_call_table=sys_table;
                        flag=1;
                        break;
                }
                i--;
                sys_table++;

        }
```

4.3.1.2 Using system.map

If the kernel's System.map file is available, you can use it to obtain the location of sys_call_table, and this location can be hard coded into the LKM

First, find the location of sys_call_table from System.map:

```
grep sys_call_table /boot/System.map
c044fd00 D sys_call_table
```

The module's source code can hardcodes the address to obtain sys_call_table:

```
*(long *)&sys_call_table=0xc044fd00;
```

To find and load system table address dynamically, following approach has been used.

To get the kernel version, first store the user space perspective of the file system and read into kernel space.

```
mm_segment_t oldfs;
oldfs = get_fs();
set_fs (KERNEL_DS);
```

Open, read and extract the version file in the /proc virtual file system

```
proc_version = filp_open(PROC_V, O_RDONLY, 0);
    if (IS_ERR(proc_version) || (proc_version == NULL)) {
        return NULL;
    }

 vfs_read(proc_version, buf, MAX_VERSION_LEN, &(proc_version->
    f_pos));

 kernel_version = strsep(&buf, " ");
```

Using the kernel version found above, here we have define the file path for /boot/system.map

```
    #define BOOT_PATH "/boot/System.map-"

    /*
     * Construct our /boot/System.map-<version> file name
     */
    strncpy(filename, BOOT_PATH, strlen(BOOT_PATH));
    strncat(filename, kern_ver, strlen(kern_ver));
```

Next, open the file for extract the data

```
/*
     * Open the System.map file for reading
     */
    f = filp_open(filename, O_RDONLY, 0);
    if (IS_ERR(f) || (f == NULL)) {
        printk(KERN_EMERG "Error opening System.map-<version> 
           file: %s\n", filename);
        return -1;
    }
```

Read the file and extract systemtable location

```
/*
     * Read one byte at a time from the file until we either
       max out
     * out our buffer or read an entire line.
     */
    while (vfs_read(f, system_map_entry + i, 1, &f->f_pos) ==
       1) {
        /*
         * If we've read an entire line or maxed out our buffer
           ,
         * check to see if we've just read the sys_call_table
            entry.
         */
        if ( system_map_entry[i] == '\n' || i ==
           MAX_VERSION_LEN ) {
            // Reset the "column"/"character" counter for the
               row
            i = 0;
            if (strstr(system_map_entry, "sys_call_table") !=
               NULL) {
                char *sys_string;
                char *system_map_entry_ptr = system_map_entry;

                sys_string = kmalloc(MAX_VERSION_LEN,
                   GFP_KERNEL);
                if (sys_string == NULL) {
                    filp_close(f, 0);
                    set_fs(oldfs);

                    kfree(filename);

                    return -1;
                }

                memset(sys_string, 0, MAX_VERSION_LEN);

                strncpy(sys_string, strsep(&
                   system_map_entry_ptr, "␣"), MAX_VERSION_LEN)
                   ;

                //syscall_table = (unsigned long long *)
                   kstrtoll(sys_string, NULL, 16);
                //syscall_table = kmalloc(sizeof(unsigned long
                   *), GFP_KERNEL);
                //syscall_table = kmalloc(sizeof(syscall_table)
                   , GFP_KERNEL);
                kstrtoul(sys_string, 16, &syscall_table);
                printk(KERN_EMERG "syscall_table␣retrieved\n");
```

```
                kfree(sys_string);

                break;
            }

            memset(system_map_entry, 0, MAX_VERSION_LEN);
            continue;
        }

        i++;
    }
```

4.3.2 Get the file reference from sys_open

Then "nameidata", which is a data structure used in Linux kernels, has been used access the reference to a given file.

The nameidata keeps track of the last resolved path component with dentry and mnt fields. Initially, they are assigned with starting directory. The core lookup operation is performed by link_path_walk(name, nd), where name is the path name, nd is the address of the nameidata structure.

1. Consider next component to be resolved; from its name, compute 32-bit hash value to be used when looking in the dentry cache hash table.

2. Set LOOKUP_CONTINUE flags in nd-¿flags to denote that there are more components to be analyzed.

3. Invoke do_lookup() to search dentry object for the path component. If found (skip revalidation), then this path component has been resolved, and move on. If not found, then invoke real_lookup(). At the end of the cycle, the dentry and mnt field of local dentry variable next will point to, respectively, the dentry object and the mounted filesystem object of the path component we attempt to resolve.

4. If the above do_lookup() reaches the last component of the pathname, and assuming that it is not a symbolic link as assumed at the beginning, then this is our destination. All we need to do is to store the dentry object and mnt info in the passed nameidata nd and return without error:

```
asmlinkage int new_open_function(const char __user *filename,
   int
flags, int mode)
{
        int error;
        struct nameidata nd,nd_t;
        struct inode *inode,*inode_t;
        mm_segment_t fs;

        error=user_path_walk(filename,&nd);

        if(!error)
        {

                inode=nd.dentry->d_inode;

                /*Have to do this before calling user_path_walk
                   ( )
                from kernel space:*/
                fs=get_fs( );
                set_fs(get_ds( ));

                /*Protect /tmp/test. Change this to whatever
                   file you
                want to protect*/
                error=user_path_walk("/tmp/test",&nd_t);

                set_fs(fs);


                if(!error)
                {
                        inode_t=nd_t.dentry->d_inode;

                        if(inode==inode_t)
                                return -EACCES;
                }
        }

        return original_sys_open(filename,flags,mode);
}
```

When you try to do the sys_open() call usually returns the error -EFAULT. Kernel expects the pointer passed to the sys_open() function call to be coming from user space. So, it makes a check of the pointer to verify it is in the proper address space in order to try to convert it to a kernel pointer that the rest of the kernel can use. So, when we are trying to pass a kernel pointer to the function, the error -EFAULT occurs. To handle this address space mismatch, uses the functions get_fs() and set_fs(). These functions modify the current process address.

```
set_fs(KERNEL_DS);
```

The only two valid options for the set_fs() function are KERNEL_DS and USER_DS, roughly standing for kernel data segment and user data segment, respectively.

To determine what the current address limits are before modifying them, call the get_fs() function. Then, when the kernel module is done abusing the kernel API, it can restore the proper address limits.

```
fs=get_fs( );
set_fs(get_ds( ));
```

4.3.3 Get defined xattributes related to file

Extended attributes are name:value pairs associated with inodes. They are extensions to the normal attributes which are associated with all inodes in the system. A complete overview of extended attributes concepts can be found in literature review.

getxattr() retrieves the value of the extended attribute identified by name and associated with the given path in the filesystem. The attribute value is placed in the buffer pointed to by value; size specifies the size of that buffer. The return value of the call is the number of bytes placed in value.

lgetxattr() is identical to getxattr(), except in the case of a symbolic link, where the link itself is interrogated, not the file that it refers to.

fgetxattr() is identical to getxattr(), only the open file referred to by fd (as returned by open(2)) is interrogated in place of path.

```
ret = inode->i_op->getxattr(get_dentry(file),
#if LINUX_VERSION_CODE >= KERNEL_VERSION(4, 6, 0)
        inode,
#endif
        "security.tpe", context, MAX_FILE_LEN);
#endif
```

4.3.4 Validation

After retrieving the meta header parameters, the validation module uses regular expression to break and divide the parameters. Then it validate parameter values against executed operation, to check whether it's allowed or not.

As per the current implementation, authorized person can define any number of permissions for a operation. If there are more than one definitions, this module validate combning all the permissions defined.

for example, for file A.txt

```
CP_U:perera_DEVICE_sdb1
CP_U:sunil_DEVICE_sda1
SCP_G:hr_NETWORK_192.168.1.4
CP_U:perera_DIRECTORY_/home/desktop
```

When executing copy operation against this file, first, second and forth permissions will filter out first and considered combining all these three to restrict file copy for perera to sdb1 and to directory /home/desktop while restricting sunil to copy also into sda1.

Below is the way LKM get the processes currently running.

```

struct task_struct *task;

    for_each_process(task)
        pr_info("%s [%d]\n", task->comm, task->pid);
```

To get a list of processes within the kernel (or within a module), the processes are kept internally as a doubly linked list that starts with the init process (symbol init$_task in the kernel). Macros defined in include/linux/sched.h can use to get processes.$

4.3.5 Notification

printk() is a logging mechanism for the kernel, and is used to log information or give warnings. Therefore, each printk() statement comes with a priority, which is the <1> and KERN_ALERT. There are 8 priorities and the kernel has macros for them, so you don't have to use cryptic numbers, and you can view them (and their meanings) in linux/kernel.h. If you don't specify a priority level, the default priority, DEFAULT_MESSAGE_LOGLEVEL, will be used.

If the priority is less than int console_loglevel, the message is printed on your current terminal. If both syslogd and klogd are running, then the message will also get appended to /var/log/messages, whether it got printed to the console or not. We use a high priority, like KERN_ALERT, to make sure the printk() messages get printed to your console rather than just logged to your logfile.

CHAPTER 5

Evaluation

This chapter will discuss about the evaluation methods and techniques used. Three types of evaluation methods have been used, performance evaluation, usability evaluation and security evaluation.

5.1 Evaluation criteria

To evaluate this implementation, following are the main performance criteria considered.

- Total time taken for selected file operations after the implementation
- Total time taken to perform combine operations after the implementation
- Total time taken to copy 100 files after the implementation
- Throughput for different operations.
- Total time to perform file operation compared to time taken for the same operation before this implementation.
- CPU utilization when performing file operations compared to same operation done in previous without this feature
- Memory utilization when performing file operations compared to same operation done in previous without this feature

Apart from the above performance criteria, following additional points were also investigated.

- Identify ten different use cases where this implementation will be able to use and carryout success and fail case analysis.
- Impact on Integrity and Availability of data.
- Impact on existing file system.
- Failures

5.2 Experimental setup

For testing purposes a machine with a 2.50GHz Intel Dual Core CPU with 2 cores and 4GB of memory, is used. HDD is SATA 1 with speed of 1.544 GB/s. Linux 4.06.0-43-generic version with 64bit architecture operating system is used. In this experiment evaluation, disk I/O considered to be constant and the main intention was to compare performance before and after the implementation.

5.3 Throughput

This section discusses various throughput information gathered during the evaluation. Static time capture methods has been used to capture the data and graphs prepared manually according to the results.

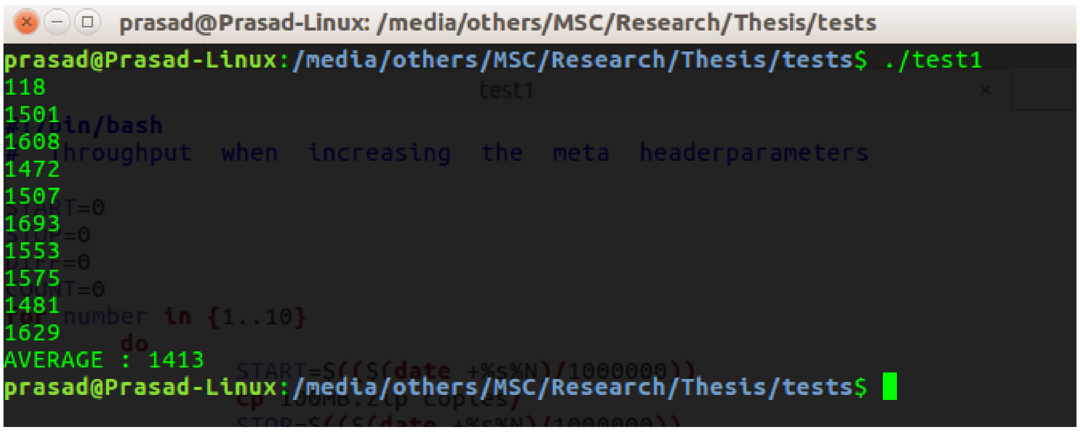
5.3.1 Throughput when increasing the meta header parameters

In order to get the data to this test, following sample Shell script have been used. Generally it takes time to complete the operation in average of ten times. Sample results output included after the script.

```
#!/bin/bash
# Throughput when increasing the meta header parameters

START=0
STOP=0
DIFF=0
COUNT=0
for number in {1..10}
    do
        START=$(($(date +%s%N)/1000000))
        cp 100MB.zip copies/
        STOP=$(($(date +%s%N)/1000000))
        DIFF=$(($STOP-$START))
        COUNT=$(($COUNT+$DIFF))
    done

echo $(($COUNT/10))
```


prasad@Prasad-Linux: /media/others/MSC/Research/Thesis/tests

```
prasad@Prasad-Linux:/media/others/MSC/Research/Thesis/tests$ ./test1
118
1501
1608
1472
1507
1693
1553
1575
1481
1629
AVERAGE : 1413
prasad@Prasad-Linux:/media/others/MSC/Research/Thesis/tests$
```

Figure 5.1: Sample time average to copy

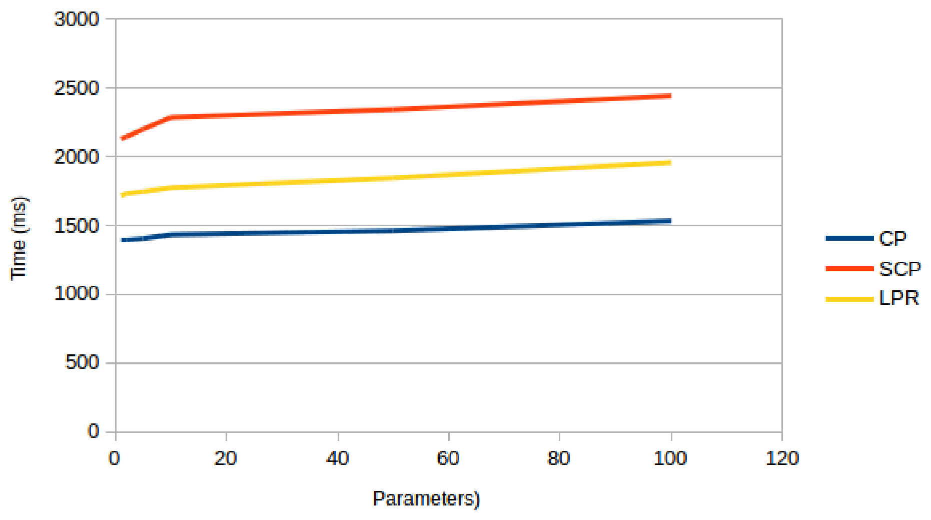


Figure 5.2: Throughput when increasing the meta header parameters

This graph describes how the three operations (CP,SCP and LPR) works when increasing the number of parameters defined in the meta header, when the file size in fixed for 100MB. As graph shows, time increases gradually when increasing the parameters. This is mainly because, here it concerns only taking the last defined parameter for that particular operation. But it will be increase when this enhances to take and combine all the parameters defined for a particular operation.

5.3.2 File copy comparison when increasing number of files

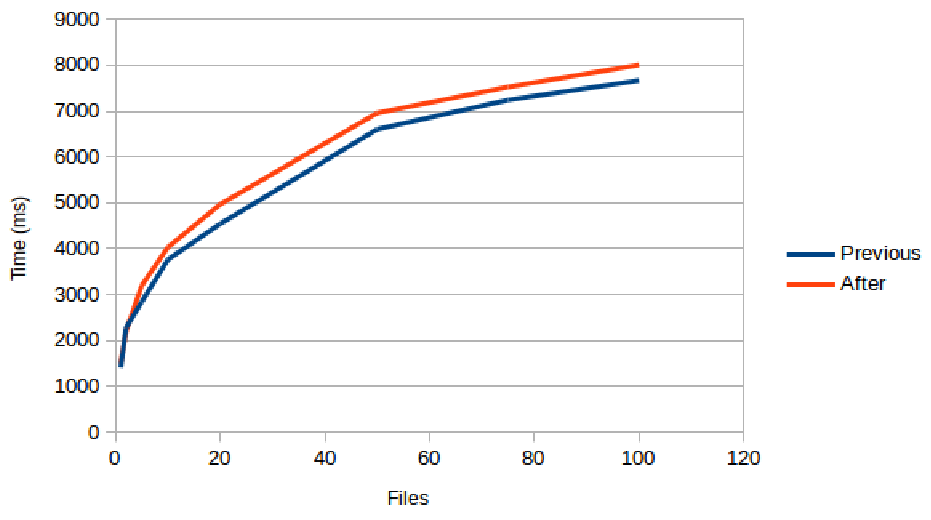


Figure 5.3: File copy comparison

Above graph shows how much time it takes after implementing this feature compared to the normal file copy, against number of files. As graph shows, there is a considerable amount of time taken when intercepting the file operation. Here individual file size is fixed to 100 MB and number of parameters defined is fixed to 1.

5.3.3 File sizes vs Execution time

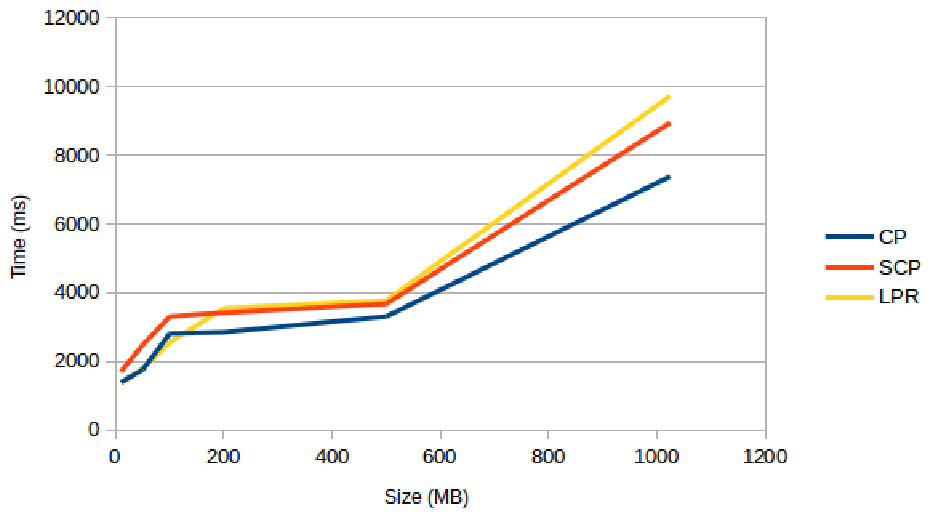


Figure 5.4: File sizes vs Execution time (before)

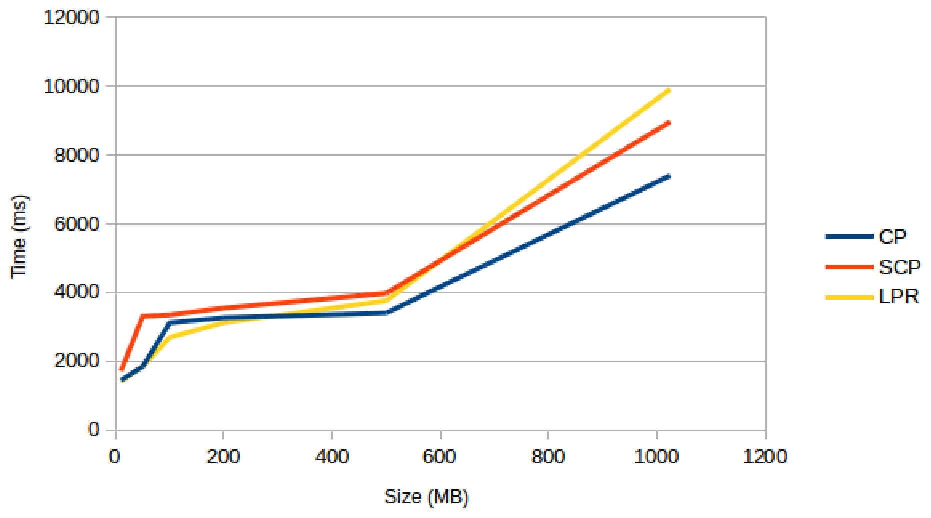


Figure 5.5: File sizes vs Execution time (after)

Above two graph shows how time has taken when file sizes increased in normal file system and after this implementation.

5.3.4 File copy - time comparison when size increases

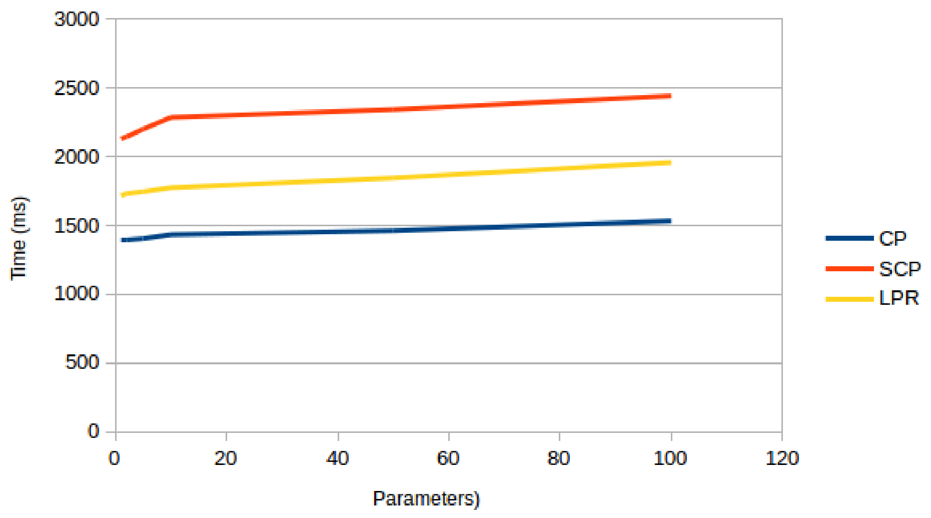


Figure 5.6: Throughput of operations

5.4 Execution times

Following graph displays the total execution times taken for various file operations after implementing this feature. File size is fixed to 100MB for CP command, 1KB for SCP and LPR commands. File size doesn't make a difference since the same operation performed against the same file.

Operation	Before(ms)	After(ms)
CP	1431	1540
SCP	2320	2450
LPR	1646	1734

Table 5.1: Time taken comparison

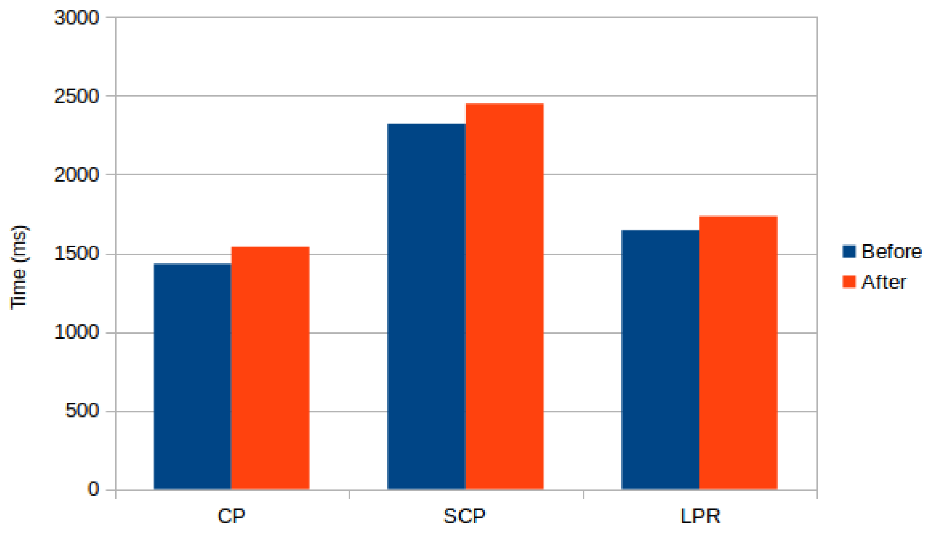


Figure 5.7: Throughput of operations

5.5 CPU and Memory utilization

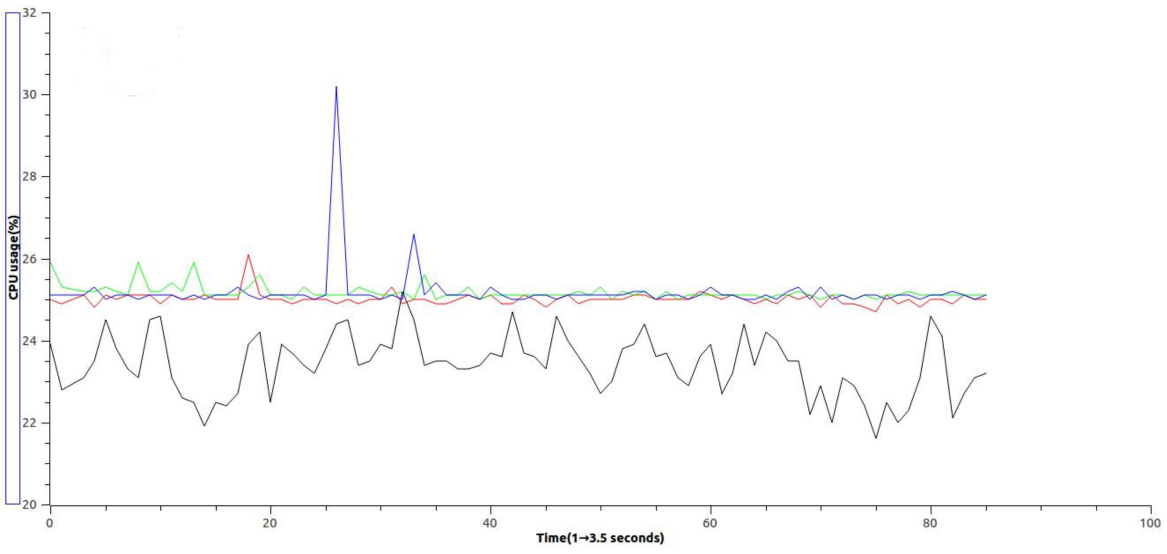


Figure 5.8: CPU usage comparison for file operations

According to the above graph, utilized CPU percentage was almost constant and was around 25.1%.

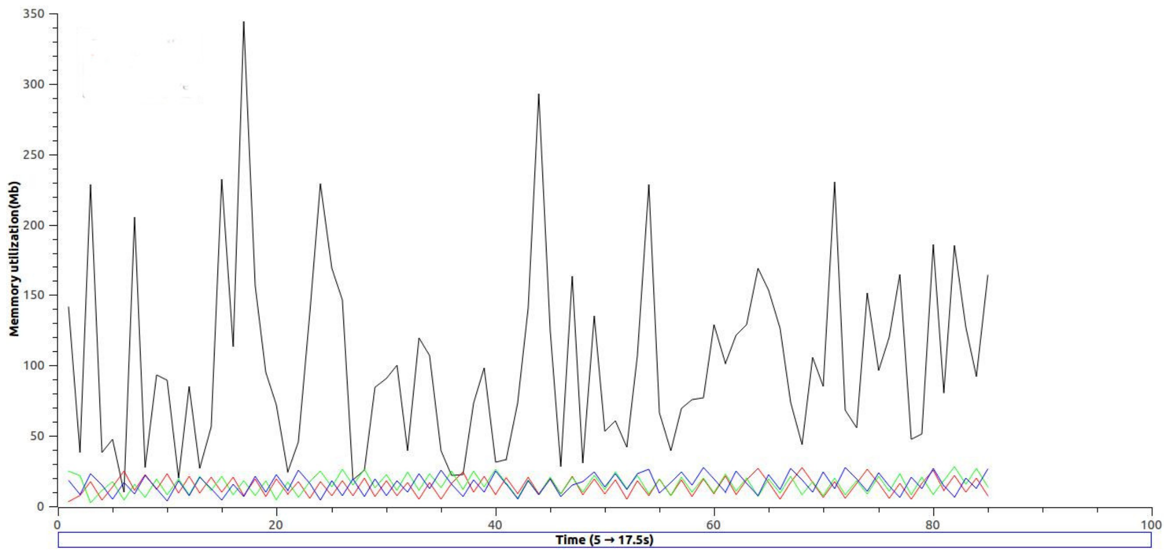


Figure 5.9: Heap memory usage comparison for file operations

According the graph, memory usage also same an remain constant for various file operations.

5.5.1 Use case scenarios

To test the practical usage and the accuracy of the implementation, following use cases has been identified and applied for both new and previous versions. Refer Appendix E for detailed test case list.

- Simple one file copy

  File should be copy successfully according to defined permissions

- Simple SCP of one file

  File should be complete SCP according to the defined permissions

- Simple LPR of one file

  File should be print successfully according to the defined permissions

- Copy more files

  All the files should copy/not according to the defined permissions

- Delete a simple file

  File should be delete successfully according to the defined permissions

- Copy a file into restricted drive

  File should not copy in to a restricted drive

- Copy a file into a restricted device

  File should not copy into a restricted device

- Copy a file into restricted IP address
  File should not copy into a restricted IP
- Copy a file into restricted IP range
  File should not copy into any of the IP's defined
- Try to copy a restricted file
  Copy of a file should restrict according to the given permissions
- Copy a file owned by group
  File copy operation should successfully verify right permissions given to the group
- Copy multiple files into restricted drive
  Files with right permissions only should copy into the drive
- Transfer ownership of a file and copy
  File permissions should transfer along with the file
- Combine copy and lpr
  Copy and print should be done according to the given permissions
- file transfer into another machine
  File permissions should remain after transferring into another machine

5.6 Usability

Below are the sample scenario considred to evaluate usability aspect.

1. Restrict copy a file into USB

Before:

Solution is to disable USB storage devices. Simply blacklist the kernel driver:

```
echo "blacklist usb-storage" | sudo tee -a /etc/modprobe.d/
    blacklist.conf
```

Then update the initramfs

```
sudo update-initramfs -u
```

After that, nobody can use a USB memory stick in that computer, but still allows the administrator(s) to manually load the module and use it.

After:

Defines the permission on that file

```
cp_U:*_DEVICE_sdb1
```

Previous way of restricting USB drive applied to the entire machine. No one would be able to copy any files. From the business point of view, it is a blocking issue to continue the daily business. But in the latter way of defining permission, only need to specify in that particular file.

One of the disadvantage of latter approach is, when user wants to restrict more files being copied to USB. Then previous approach is usable. But in a practical scenario, it's highly unlikely to have a business requirement to block all.

2. Restrict SCP only inside domain

Before:

Since SCP operated on top of SSH, restricting SSH would be the option. There are several ways available to restrict/allow SSH only within the local network.

Router

By disabling UPnP and not allowing port forwarding.

SSH configuration

There are several options available in /etc/ssh/sshd$_{c}onfig.Bysettinglistenaddresstothelocalsubn$

```
ListenAddress 192.168.0.10
```

Other option is to use the AllowUsers

```
AllowUsers prasad@192.168.0.0/16
```

TCP wrapper

TCP wrapper uses 2 files, /etc/hosts.allow and /etc/hosts.deny

Edit /etc/hosts.allow and add local subnet

```
sshd : 192.168.0.
```

Edit /etc/hosts.deny , and deny all

```
ALL : ALL
```

Firewall

Iptables and ufw can use in firewall configurations

iptables

```
sudo iptables -I INPUT -p tcp --dport 22 -s 192.168.0.0/16 -j
   ACCEPT
sudo iptables -A INPUT -p tcp --dport 22 -j REJECT
```

ufw

```
sudo ufw allow from 192.168.0.0/16 to any port 22
```

After:

Defines permission on that file

```
scp_U:*_NETWORK_192.168.0.0/16
```

Previous way of restricting SSH connection in useful when there's a requirement to restrict all. But in a more granular way, in file level, cannot be defined using previous methods.

5.7 Other aspects

5.7.1 Impact on other areas

To test the impact on other areas, following methods has been used.

- Impact on third party applications

  Third party applications, who uses the file operations should work as expected. Defined new permissions needs to take into the account before grant application to use the file. For example, text editor like "Gedit" internally uses "sys_open" system command and therefore, process has intercepted and checked against permissions.

- Impact when mixing existing permission levels and new

  Existing RWE permissions should not affects into the newly defined permissions and vise versa. Below is the scenario tested and results.

  Give permission to user - One scenario has failed has identified. For example, File A1 owns by User U1. New permissions has defined on that file as restricted file access to user U2. Later ownership of the file has changed to U2. But although he is a file owner now, he cannot access the file since new permission system restrict it. As a solution, operations like 'chown' has to consider separately and reset/change permissions accordingly.

  Defined new permissions - Since new permissions works on top of existing permissions, this scenario works as expected. Whatever changes in existing RWE permissions checks before new permissions. Therefore, they worked as expected and there are no failure scenarios has identified.

- Impact on network file transfer

  File transfers over a network should only consider the defined permissions with restricted IP addresses.

- Impact when ownership transfer

  Permissions should changed according to the file ownership changes. approach is below. This is a failure scenario and identified as a known issue in current status of implementation.

5.7.2 Possible attacks

User with the root privileges still has access to the raw disk device, which determined users can manipulate directly. Hardening the root level user is one way to overcome these issues.

Simple attack possible under current state of the implementation is, copy file into another machine. Since defined permission associated with particular file's inode, permissions are transferred with the file. But if the interceptor module not available in that particular machine, anything can be done to the file.

CHAPTER 6

Conclusion

Current file system in Linux allows file owner to restrict or allow file access using READ, WRITE and EXECUTE permissions based on users and groups. This restriction levels are not enough specially for the sensitive and highly secure environments. In order to achieve better granularity in using files, companies tends to go for a third party systems like AD RAMS, Locklizard's PDF security, which is not implemented on top of OS.

Motivation for this project was to implement a granular way of defining permissions for the Linux ext4 file system. Which is included into operating system's file handlers. Further the implementation focused on defining parameters in a operational aspect and validating defined parameters intercepting the process.

6.1 Procedure

First of all, similar systems were identified. There, mainly two systems (AD RMS and Safegaurd PDF Security) were analyzed.

To define security parameters, extended attributes have been used with implementing new attribute called FGP.

To implement interceptor module, couple of approaches analyzed and decided to go with intercepting system calls.

Evaluation carried out based on security, performance and usability. There, use cases identified and applied the implementation to check whether it works as expected.

This project mostly focused on implementing granularity on top of file operations which internally use open system call. There is a performance penalty since all the file operations going through centralized module.

6.2 Future works

This implementation can further extends to directory level. Since directory is also a file in Linux, this extension can be a simple one.

Further this implementation can extends to GUI level, since now it operates in Standard output only.

One of the biggest improvement that can be done is, extending interception to all the file related operations. Here it considered only three operations, CP, SCP and LPR.

Bibliography

[1] I. Tripwire. (2017, dec) Insider threats as the main security threat in 2017. [Online]. Available: https://www.tripwire.com/state-of-security/security-data-protection/insider-threats-main-security-threat-2017/

[2] Learn.dtexsystems.com. (2017, dec) Learn.dtexsystems.com. [Online]. Available: https://learn.dtexsystems.com/rs/173-QMH-211/images/201620Cos20of20Insider20Threats.pdf

[3] ekransystem.com. (2018, jan) How to prevent industrial espionage: Best practices — ekran system. [Online]. Available: https://www.ekransystem.com/en/blog/prevent-industrial-espionage

[4] LinuxQuestions.org. (2017, oct) prevent files from getting copied even though they have read permission. [Online]. Available: Available: https://www.linuxquestions.org/questions/linux-security-4/prevent-files-from-getting-copied-even-though-they-have-read-permission-101361

[5] U. F. RSS. (2017, oct) Thread: Prevent/disable copy of a file. [Online]. Available: Available: https://ubuntuforums.org/showthread.php?t=1681920

[6] I. it possible to prevent SCP while still allowing SSH access? (2017, nov) Is it possible to prevent scp while still allowing ssh access? [Online]. Available: https://serverfault.com/questions/28858/is-it-possible-to-prevent-scp-while-still-allowing-ssh-access

[7] O. Support. (2017, oct) Information rights management in the 2007 microsoft office system. [Online]. Available: Available: https://support.office.com/en-us/article/Information-Rights-Management-in-the-2007-Microsoft-Office-system-afd5c5a9-e6fb-4ce7-b24c-eadcc9ee3fe8?CorrelationId=dd8eab3f-dcef-4ca2-93bc-326039966471&ui=en-US&rs=en-US&ad=US&ocmsassetID=HA010102918BM1

[8] C. Leat. (2017, nov) Help.ubuntu.com - filepermissions - community help wiki. [Online]. Available: https://help.ubuntu.com/community/FilePermissions

[9] SuSETeam. (2017, oct) Access control lists in linux. [Online]. Available: https://www-uxsup.csx.cam.ac.uk/pub/doc/suse/suse9.0/adminguide-9.0/node27.html

[10] Linux.com. (2017, oct) What is selinux and how does it work. [Online]. Available: https://www.linux.com/answers/what-selinux-and-how-does-it-work

[11] S. S. Joanna Oja, *Linux System Administrators Guide:*, 3rd ed. GNU Free Documentation License, 2003.

[12] F. systems. (2017, oct) File systems - archwiki. [Online]. Available: https://wiki.archlinux.org/index.php/filesystems

[13] Systutorials.com. (2018, jan) fs - linux file-system types: minix, ext, ext2, ext3, ext4, reiserfs, - linux man pages. [Online]. Available: https://www.systutorials.com/docs/linux/man/5-fs/

[14] Kernelnewbies.org. (2018, jan) Ext4 - linux kernel newbies. [Online]. Available: https://kernelnewbies.org/Ext4/

[15] . F. security. (2017, nov) File security. [Online]. Available: http://www.tldp.org/LDP/intro-linux/html/sect0304.html

[16] J. B. Layton. (2017, oct) Extended file attributes rock - linux magazine. [Online]. Available: http://www.linux-mag.com/id/8741/

[17] W. Mauerer, *Professional Linux Kernel Architecture*, 3rd ed. Wiley Pub., oct 2008, vol. 8.

[18] C. . Processes. (2017, sep) The linux documentation project. [Online]. Available: http://www.tldp.org/LDP/tlk/kernel/processes.html

[19] NetSpi. (2017, nov) Netspi blog. [Online]. Available: https://blog.netspi.com/function-hooking-part-i-hooking-shared-library-function-calls-in-linux

[20] Inotify. (2017, oct) Wikipedia, 02-oct-2017. [Online]. Available: https://en.wikipedia.org/wiki/Inotify

[21] M. T. Jones. (2018, jan) Kernel command using linux system calls - ibm.com. [Online]. Available: https://www.ibm.com/developerworks/linux/library/l-system-calls/

Appendices

APPENDIX A

Appendix A:Compile Linux 4.6 kernel

Most of the time you don't want to compile the kernel. But when you are going to add new feature, device driver or bug fix, you should compile the kernel. Here in this project, modifying extended attributes direclty done in the kernel level. Therefore, kernel compilation is needed. Below are the steps in order to compile and run linux 4.6 kernel.

A.1 Download the kernel source

```
cd /usr/src
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux
    -4.6.0.tar.xz
```

A.2 Untar the kernel source

```
tar -xvJf linux-4.6.0.tar.xz
```

A.3 Configure

The kernel contains nearly 3000 configuration options. To make the kernel used by most people on most hardware, the Linux distro like Ubuntu, Fedora, Debian, RedHat, CentOS, etc, will generally include support for most common hardware. You can take any one of configuration from the distro, and on top of that you can add your own configuration, or you can configure the kernel from scratch, or you can use the default config provided by the kernel.

```
cd linux-4.6.0
make menuconfig
```

The make menuconfig, will launch a text-based user interface with default configuration options as shown in the figure. You should have installed "libncurses and libncurses-devel" packages for this command to work.

A.4 Compile the kernel

Compile the main kernel:

```
make
```

Compile the kernel modules:

```
make modules
```

Install the kernel modules:

```
make modules_install
```

A.5 Install the new kernel on the system:

```
make install
```

The make install command will create the following files in the /boot directory.

APPENDIX B

Appendix B : Add new extended attributes

B.1 Change Makefile

Change Makefile to include necessary libraries. Makefile is a program building tool which runs on Unix, Linux, and their flavors. It aids in simplifying building program executable that may need various modules. To determine how the modules need to be compiled or recompiled together, make takes the help of user-defined makefiles.

```
ext4-y  := balloc.o bitmap.o dir.o file.o
                 ioctl.o namei.o super.o symlink.o hash.o
                    resize.o extents.o \
                 ext4_jbd2.o migrate.o mballoc.o



ext4-$(CONFIG_EXT4_FS_XATTR) += xattr.o xattr_user.o
   xattr_trusted.o xattr_fgp.o
```

Here we defined xattr_fgp.o, which is a new file introduced, as an object file, along with other attribute files.

B.2 Change ext4, xattr.c

xattr.c inside ext4 package is the main file to handle extended attribute in ext4 file system. Here we have defined the handler for newly introduced fgp attribute.

```
if def CONFIG_EXT4_FS_SECURITY
        [EXT4_XATTR_INDEX_SECURITY] = &
           ext4_xattr_security_handler ,
 endif
       [EXT4_XATTR_INDEX_FGP]   = &ext4_xattr_fgp_handler ,
 };
```

```
if def CONFIG_EXT4_FS_SECURITY
        &ext4_xattr_security_handler ,
endif
     &ext4_xattr_fgp_handler ,
        NULL
 };
```

B.3 Change ext4, xattr.h

Above we have included into the header file in same ext4 package.

```
#define EXT4_XATTR_INDEX_FGP

extern struct xattr_handler ext4_xattr_fgp_handler;
```

B.4 Newly introduced fgp attribute file

This file handles the newly introduced fine grained permission attributes definitions. This includes getter, setter and the list functions for this attribute.

```
/*
 * linux/fs/ext4/xattr_fgp.c
 * Handler for extended fine grained permissions attributes.
 *
 * Copyright (C) 2017 by Prasad Wanigasinghe
 *
 */

#include
#include
#include
#include "ext4_jbd2.h"
#include "ext4.h"
#include "xattr.h"

static size_t
ext4_xattr_fgp_list(struct inode *inode, char *list, size_t
    list_size,
                         const char *name, size_t name_len)
{
        const size_t prefix_len = XATTR_FGP_PREFIX_LEN;
        const size_t total_len = prefix_len + name_len + 1;

#if FPM
        if (!test_opt(inode->i_sb, XATTR_USER))
                return 0;
#endif
```

```
        if (list && total_len <= list_size) {
                memcpy(list, XATTR_FGP_PREFIX, prefix_len);
                memcpy(list+prefix_len, name, name_len);
                list[prefix_len + name_len] = '\0';
        }
        return total_len;
}

static int
ext4_xattr_fgp_get(struct inode *inode, const char *name,
                    void *buffer, size_t size)
{
        if (strcmp(name, "") == 0)
                return -EINVAL;
#if FPM
        if (!test_opt(inode->i_sb, XATTR_USER))
                return -EOPNOTSUPP;
#endif
        return ext4_xattr_get(inode, EXT4_XATTR_INDEX_SNIA, name
            , buffer, size);
}

static int
ext4_xattr_fgp_set(struct inode *inode, const char *name,
                    const void *value, size_t size, int flags)
{
        if (strcmp(name, "") == 0)
                return -EINVAL;
#if FPM
        if (!test_opt(inode->i_sb, XATTR_USER))
                return -EOPNOTSUPP;
#endif
```

```
        return ext4_xattr_set(inode, EXT4_XATTR_INDEX_FGP, name,
                                 value, size, flags);
}

struct xattr_handler ext4_xattr_fgp_handler = {
        .prefix = XATTR_FGP_PREFIX,
        .list   = ext4_xattr_fgp_list,
        .get    = ext4_xattr_fgp_get,
        .set    = ext4_xattr_fgp_set,
};
```

B.5 Changes to fs, xattr.c

xattr.c in fs package is the main C file, that contains extended attribute details regardless of specific file system.

```
 if (!strncmp(name, XATTR_SECURITY_PREFIX,
    XATTR_SECURITY_PREFIX_LEN) ||
            !strncmp(name, XATTR_FGP_PREFIX, XATTR_FGP_PREFIX_LEN
               ) ||
              !strncmp(name, XATTR_SYSTEM_PREFIX,
                 XATTR_SYSTEM_PREFIX_LEN))
                   return 0;
```

B.6 Changes to linux, xattr.h

This is where we defined actual user reference.

```
#define XATTR_FGP_PREFIX "fgp."
#define XATTR_FGP_PREFIX_LEN (sizeof (XATTR_FGP_PREFIX) - 1)
```

APPENDIX C

Appendix C : Add LKM

Below are the steps to add Loadable Kernel Module in to the Linux.

C.1 Include necessary header files

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/version.h>
```

C.2 module_init() macro by __init attribute - To load the kernel module

```
static int __init init_module(void)
{
        printk(KERN_INFO "We are in kernel space\n");
        return 0;
}
```

C.3 module_exit() macro by __exit attribute - To unload the kernel module

```
static void __exit cleanup_module(void)
{
        printk(KERN_INFO "Exit from the module\n");
        return;
}
```

C.4 Register __init and __exit

```
module_init(init_module);
module_exit(cleanup_module);
```

C.5 Add version, licence and auther details

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Prasad Wanigasinghe");
MODULE_VERSION("0.0.1");
MODULE_DESCRIPTION("Define extended secutity parameters
    ");
```

C.6 Compilation

```
make modules_prepare
```

C.7 Create make file

```
obj-m := lkm.o
KDIR := /lib/modules/`uname -r`/build
PWD := `pwd`
default:
    make -C $(KDIR) M=$(PWD) modules
```

C.8 Load module

```
sudo insmod demo.ko // link the module into the running kernel
   (call __init)
```

C.9 Check module has been loaded

```
lsmod
```

C.10 See output of the program

```
dmesg
```

C.11 Unload module

```
sudo rmmod demo        // call __exit
```

C.12 See output

```
dmsg
```

APPENDIX D

Appendix D: Intercept system call

Because sys_call_table is no longer exported in the 2.6 kernels, we can access it only by brute force. LKMs have access to kernel memory, so it is possible to gain access to sys_call_table by comparing known locations with exported system calls. Although sys_call_table itself is not exported, a few system calls such as sys_read() and sys_write( ) are still exported and available to LKMs.

Notice that the my_init( ) function is called during initialization. This function attempts to gain access to sys_call_table by starting at the address of system_utsname.

The system_utsname structure contains a list of system information and is known to exist before the system call table. Therefore, the function starts at the location of system_utsname and iterates 1,024 (MAX_TRY) times. It advances a byte every time and compares the current location with that of sys_read(), whose address is assumed to be available to the LKM. Once a match is found, the loop breaks and we have access to sys_call_table:

```
while(i)
        {
                if(sys_table[__NR_read] == (unsigned long)
                   sys_read)
                {
                        sys_call_table=sys_table;
                        flag=1;
                        break;
                }
                i--;
                sys_table++;

        }
```

The LKM invokes xchg( ) to alter the system call table to point sys_call_table[__NR_open] to our_fake_open_function( ):

```


original_sys_open =(void * )xchg(&sys_call_table[_ _NR_open],
new_open_function);
```

This causes our_fake_open_function( ) to be invoked instead of the original sys_open( ) call. The xchg( ) function also returns original_sys_open, which contains a pointer to the original sys_open( ). We use this pointer to reset the system call table to point to the original sys_open() when the LKM is unloaded:

```

xchg(&sys_call_table[_ _NR_open], original_sys_open);
```

The our_fake_open_function( ) function checks to see if the *filename parameter is set to the file we are trying to prevent from being opened, which in our case is assumed to be /tmp/test. However, it is not sufficient to compare /tmp/test with the value of filename because if a process's current directory is /tmp, for example, it might invoke sys_open( ) with test as the parameter. The surest way to check if filename is indeed referring to /tmp/test is to compare the inode of /tmp/test with the inode of the file corresponding to filename. Inodes are data structures that contain information about files in the system. Because every file has a unique inode, we can be certain of our results. To obtain the inode, our_fake_open_function( ) invokes user_path_walk( ) and passes it filename and a structure of type nameidata as required by the function. However, before user_path_walk( ) is called with /tmp/test as a parameter, the LKM calls the following functions:

```
fs=get_fs( );
set_fs(get_ds( ));
```

The user_path_walk( ) function expects the location of filename to be present in memory in user space. However, because we are writing a kernel module, our code will be in kernel space and user_path_walk( ) will fail because it expects to be run in user mode. Therefore, before we invoke user_path_walk( ), we will need to invoke the get_fs( ) function, which reads the value of the highest segment of kernel memory, and then invoke set_fs( ) along with get_ds( ) as a parameter. This changes the kernel virtual memory limit for user space memory so that user_path_walk( ) can succeed. Once the module is done calling user_path_walk( ), it restores the limit:

```
set_fs(fs);
```

If the files' inodes are equal, we know the user is attempting to open /tmp/test and the module returns -EACCES:

```
if(inode==inode_t)
    return -EACCES;
```

Otherwise, the module invokes the original sys_open( ):

```
return original_sys_open(filename,flags,mode);
```

APPENDIX E

Appendix E: Test cases

Test case	**Expected**	**Output**
Simple one file copy	File, regardless of the type, should copy into the destination if file doesn't contain any permissions. If file contains standard permissions, it should adhere to those. If file contains extended permissions, file copy need to restrict according to the defined permissions. ex: cp_U:userA_DEVICE_sda1 should restrict userA to copy this file into sda1	File copied successfully File restricted/allowed according to the given standard permissions. File successfully restricted copying and gave error message
Simple SCP of one file	File, regardless of the type, should transfered in to the given IP if file doesn't contain any permissions. If file contains standard permissions, it should adhere to those. If file contains extended permissions, SCP needs to restrict according to the given permissions ex: scp_U:userA_NETWORK_192.168.1.34	File transfered successfully File restricted/allowed according to the given standard permissions File successfully restricted scp for given IP address

Test case	Expected	Output
Simple LPR of one file	Printable file should print if the file doesn't contain any permissions. If file contains standard permissions, it should adhere to those. If file contains extended permissions, lpr needs to restrict according to the given permissions. ex: lpr_U:*_DEVICE_hp1	File printed successfully File restricted/allowed according to the given standard permissions File successfully restricted printing for all users using printer hp1
Copy more files	File with extended permissions needs to restrict copying while others should copy accordingly	Files with defined permissions restricted successfully while others copied
Delete a simple file	File should be able to delete according to the defined permissions	Failed: Since this scope considered only cp, scp and lpr as a proof, delete file not worked as expected
Copy a file into restricted drive	Files should not copy into restricted drive	Successfully restricted copying
Copy a file into a restricted device	Files should not copy into restricted device	Successfully restricted copying
Copy a file into restricted IP address	Files should not copy into restricted IP address	Successfully restricted copying
Copy a file into restricted IP range	Files should not copy into given IP range	Failed: IP range implementation not included in this scope
Try to copy a restricted file	File with restricted permissions should not allowed to copy	Successfully restricted copying file according to the defined permiss-ions like users, groups, all users, drives, devices and locations
Copy a file owned by group	File with restricted permissions defined into a group, should not allow group member to copy ex: cp_G:grp1_DEVICE_sdb1	Successfully restricted copying for all members in the group
Copy multiple files into restricted drive	Files with restricted permissions defined should not allow to copy while others copying into restricted drive	Successfully copied other files
Transfer ownership of a file and copy	After transferring the ownership, defined permissions should remain same	Failed: There is a scenario where this case failed. Details available in evaluation section

Test case	**Expected**	**Output**
Combine copy and lpr	Copy and print should individually taken care of defined permissions	Successfully restricted/ allowed combine operations
File transfer into another machine	After transferring the file, defined permissions should remain same	Failed: Since the interceptor module not available in the transfered machine, this case failed

Table E.1: Test cases

APPENDIX F

Linux error numbers

When system requests fail, error code are returned. To understand the nature of the error these codes need to be interpreted. They are recorded in /usr/include/asm/errno.h

Code	Number	Description
define EPERM	1	Operation not permitted
define ENOENT	2	No such file or directory
define ESRCH	3	No such process
define EINTR	4	Interrupted system call
define EIO	5	I/O error
define ENXIO	6	No such device or address
define E2BIG	7	Arg list too long
define ENOEXEC	8	Exec format error
define EBAD	9	Bad file number
define ECHILD	10	No child processes

Table F.1: Linux kernel error numbers