JANUARY 27, 2024

CRACCON'24
New Delhi, Bharat

माघ कृष्ण द्वितीया
वि० सं० २०८०

नई दिल्ली, भारत

POKING THE FILESYSTEM

ADHOKSHAJ MISHRA
Security Research Lead,
Detections And Threat Research,
Quick Heal Technologies Ltd.

Agenda

1. Who am I?
2. Motivation Behind Work
3. Filesystem Internals
4. Types of Filesystem Driver
5. Rogue Filesystem As An Attacker Tool

Agenda

6. Writing A Rogue Filesystem Driver
7. Attacking The User-space
8. Limitations
9. Detecting & Preventing Rogue Filesystems
10. External References / Resources

Who am I?

- Security researcher
  - Mostly into offensive side of malware, little bit into defensive side
  - Other areas: applied cryptography, side channel attacks
- Hobbyist programmer
- FOSS enthusiast
- Guilty pleasure: breaking things for fun
- Current gig: Security Research Lead @ Quick Heal
  - Figuring out how to detect new malware and attacks
- Let us connect
  - LinkedIn: adhokshajmishra

Disclaimer

The technique(s) presented hereafter are offensive in nature; and are generally considered a criminal offence if practiced without proper authorisation in place. It is presented here for educational purpose only.

In other words, if you come to me saying that you are neck-deep in mess due to these techniques, I won't feel responsible at all.

You have been warned.

2. MOTIVATION

2 Motivation Behind Work

Why bother doing this in first place?

- User-space trusts filesystem
  - User-space software treats data coming from filesystem as more trustworthy (i.e. file indeed contains this data that I am getting after read())
  - Security solutions (including but not limited to EDRs) also put too much trust in the filesystem

2 Motivation Behind Work

Why bother doing this in first place?

- It gives pretty strong foothold in system
  - Once a filesystem driver gets compromised / backdoored, it is effectively game over.
  - Can be very difficult to detect post compromise.
- It is difficult to pull off
  - Social browny points!

अच्छी टेक्नीक हो तो क्या कुछ नहीं हो सकता

3. FILESYSTEM INTERNALS

3 Filesystem Internals

Fundamentals

- A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organised on the disk.
- Making a filesystem: Process of writing bookkeeping data structures to the disk in order to initialise a partition or disk.
- Most UNIX filesystems share the same general design, however, implementation details vary.

Central Concepts

- Superblock: The superblock contains information about the filesystem as a whole, such as its size (the exact information here depends on the filesystem).
- Inode: An inode contains all information about a file, except its name. The name is stored in the directory, together with the number of the inode.
- Dentry: A dentry or directory entry consists of a filename and the number of the inode which represents the file.

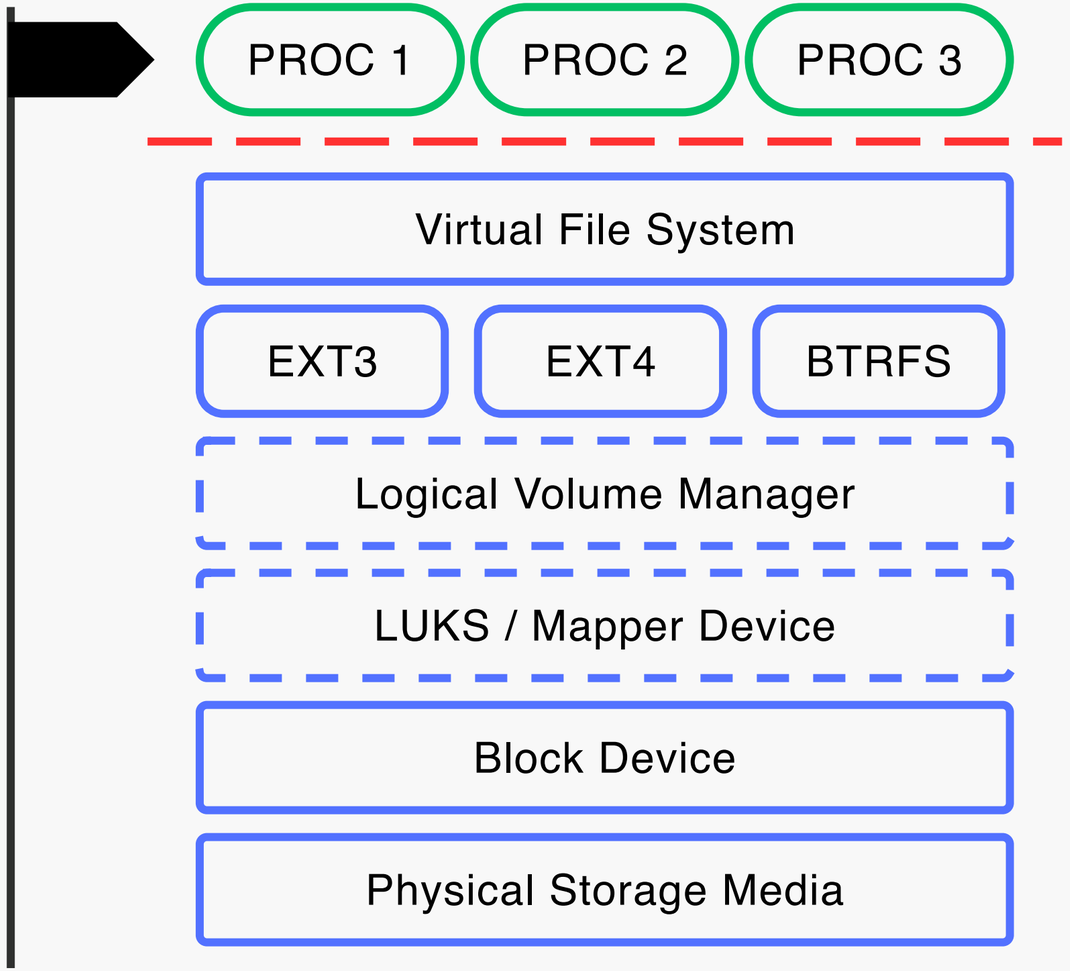
Central Concepts

- Data block: Data blocks are used to store the data in the file. Data blocks of same file are held in the same inode.
- Indirection block: It has details of dynamically allocated data blocks for any given inode.

Linux Filesystem Stack

Processes running in user-mode interact with filesystem using one of the following system calls:

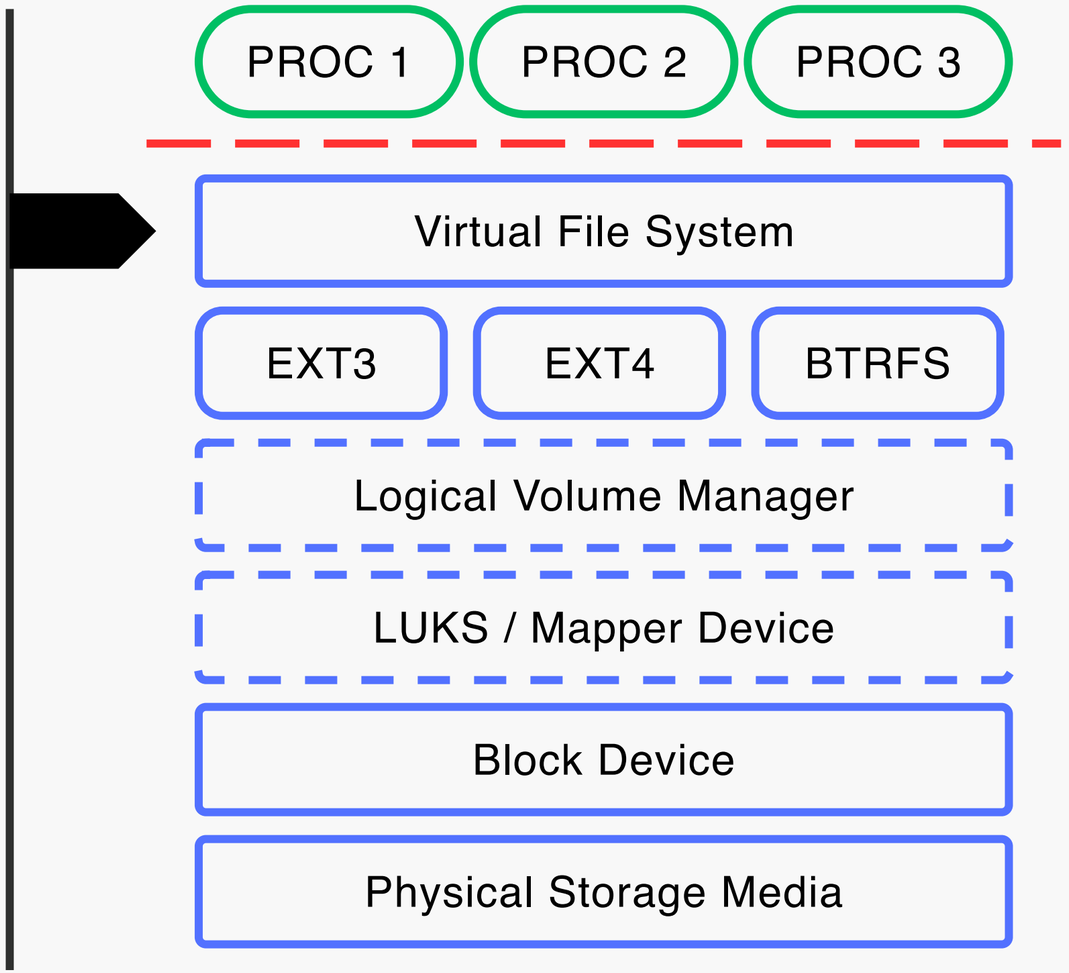
- open: opens a file
- read: reads from file
- write: write into file
- stat / fstat / lstat: get information about file
- etc.



Linux Filesystem Stack

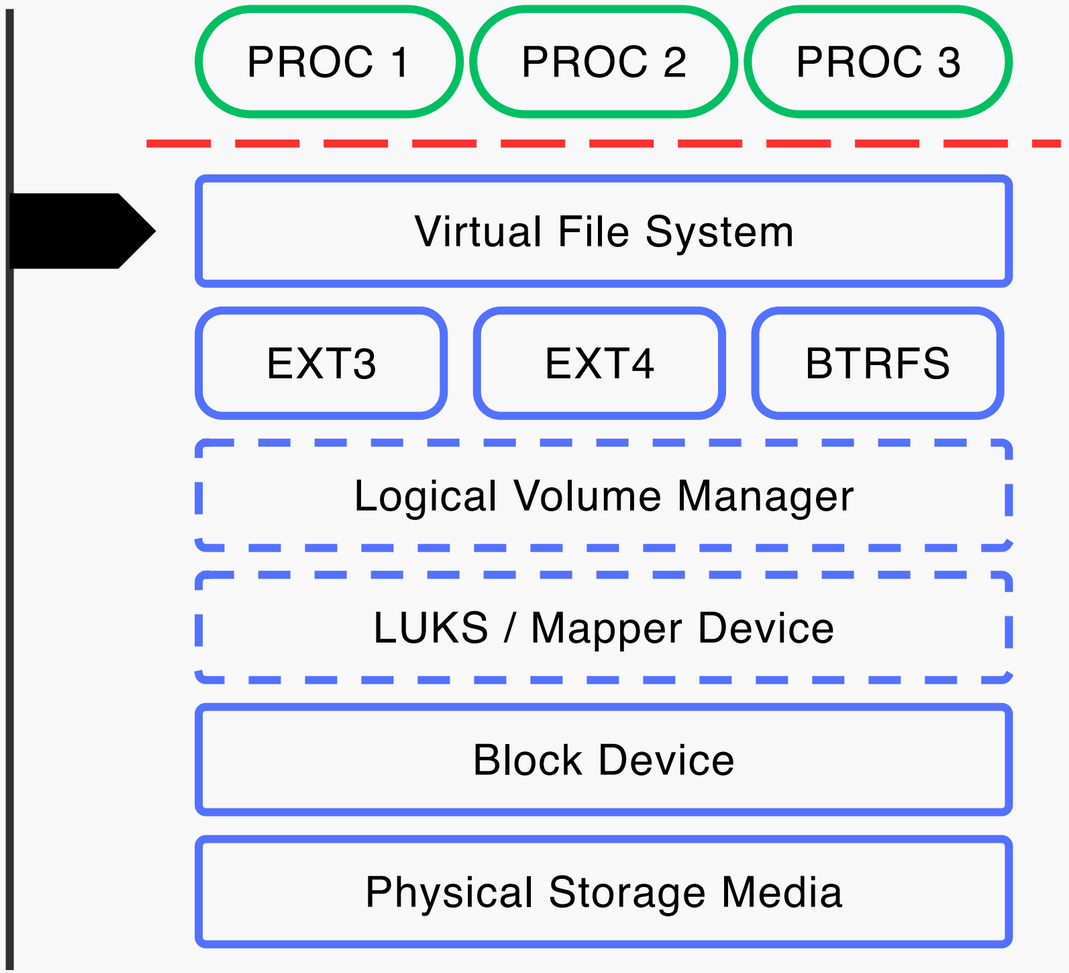
It is an abstraction within Linux kernel which allows multiple filesystems to co-exist. This layer also implements the file I/O system calls which are used by user mode processes.

Operations like finding dentry, inodes, allocating and initialising file structures (for opening a file) etc. also happen at this layer.



Linux Filesystem Stack

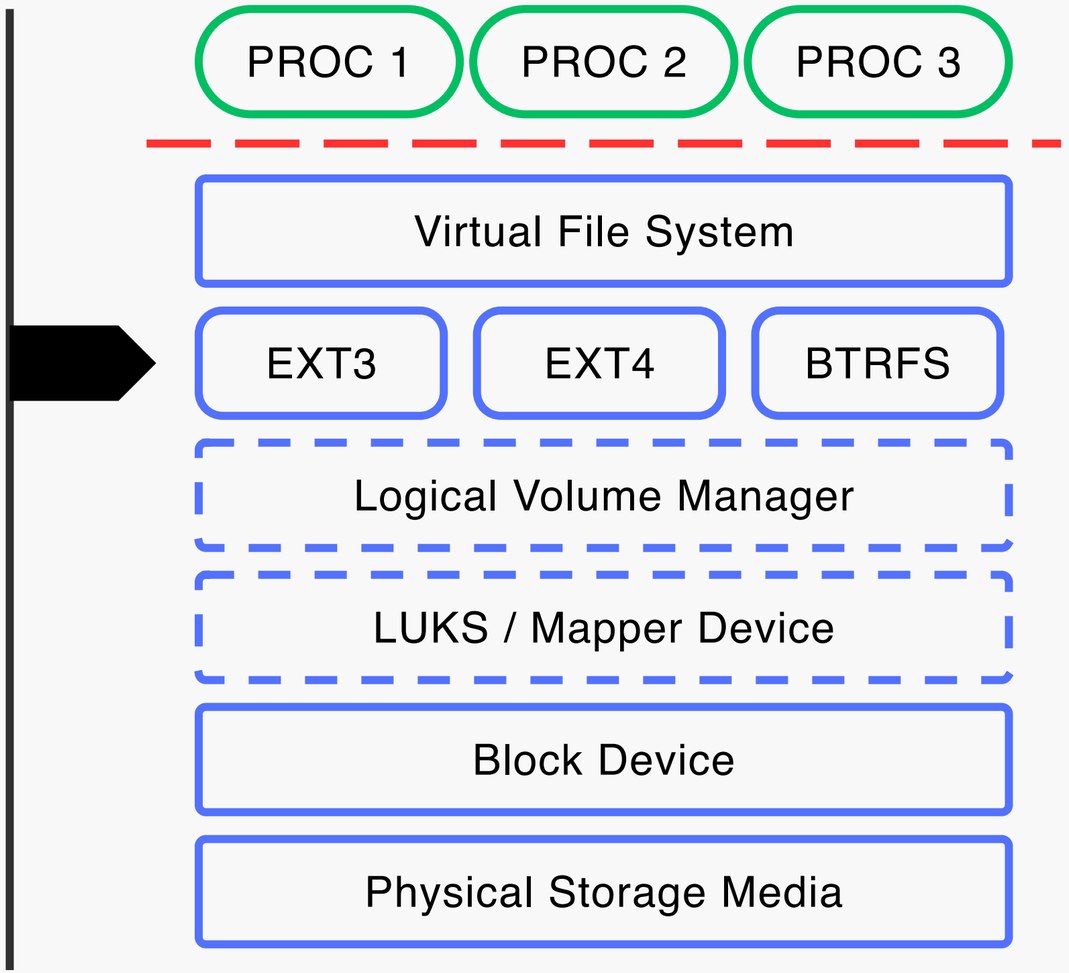
Operations which are to be done at file / dentry / inode level are passed to specific file system driver via function pointers (which point to specific implementations from different filesystem drivers).



Linux Filesystem Stack

Linux kernel defines a standard interface between VFS (virtual file system) and actual filesystem driver.

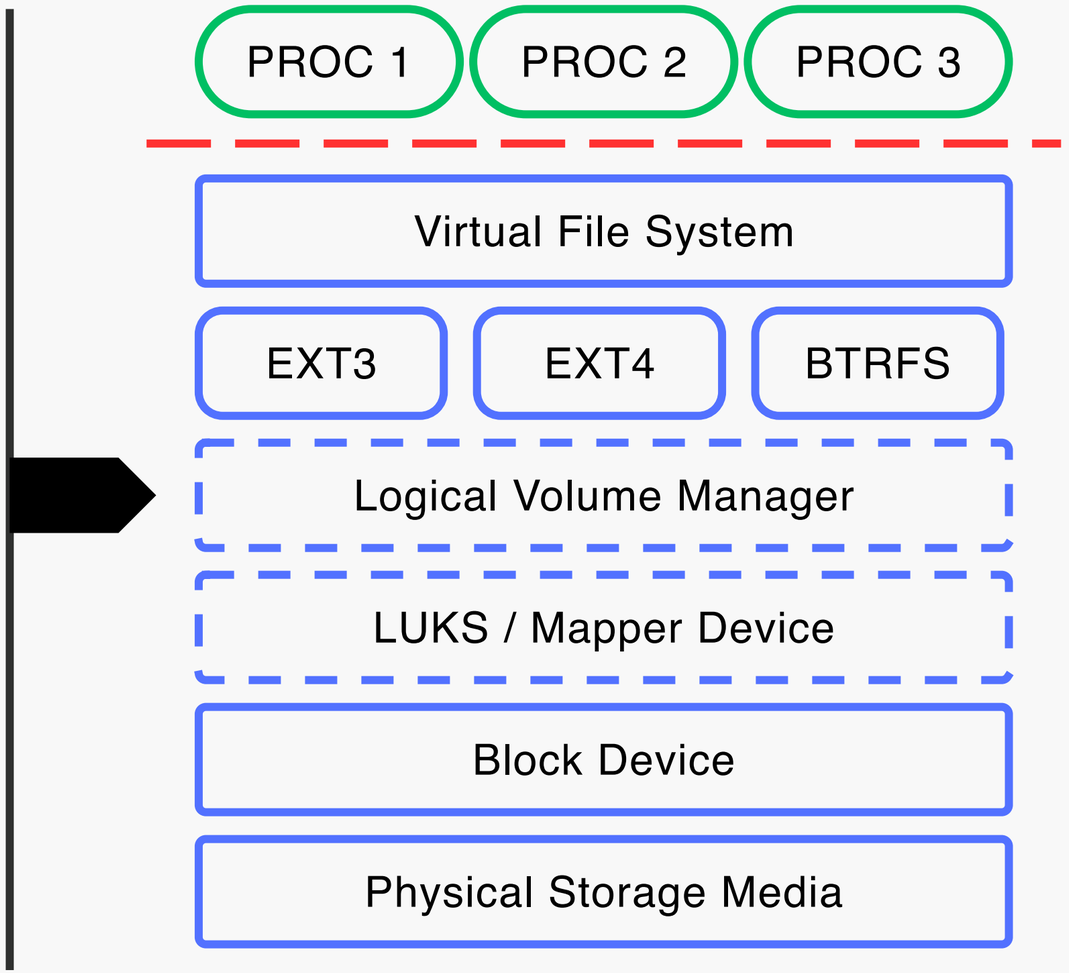
Filesystem driver defines actual functions for filesystem transactions. Pointers to these functions are passed to kernel at time of startup (when driver is loaded in kernel).



Linux Filesystem Stack

Logical volume manager provides an abstract storage device which provides extra features:

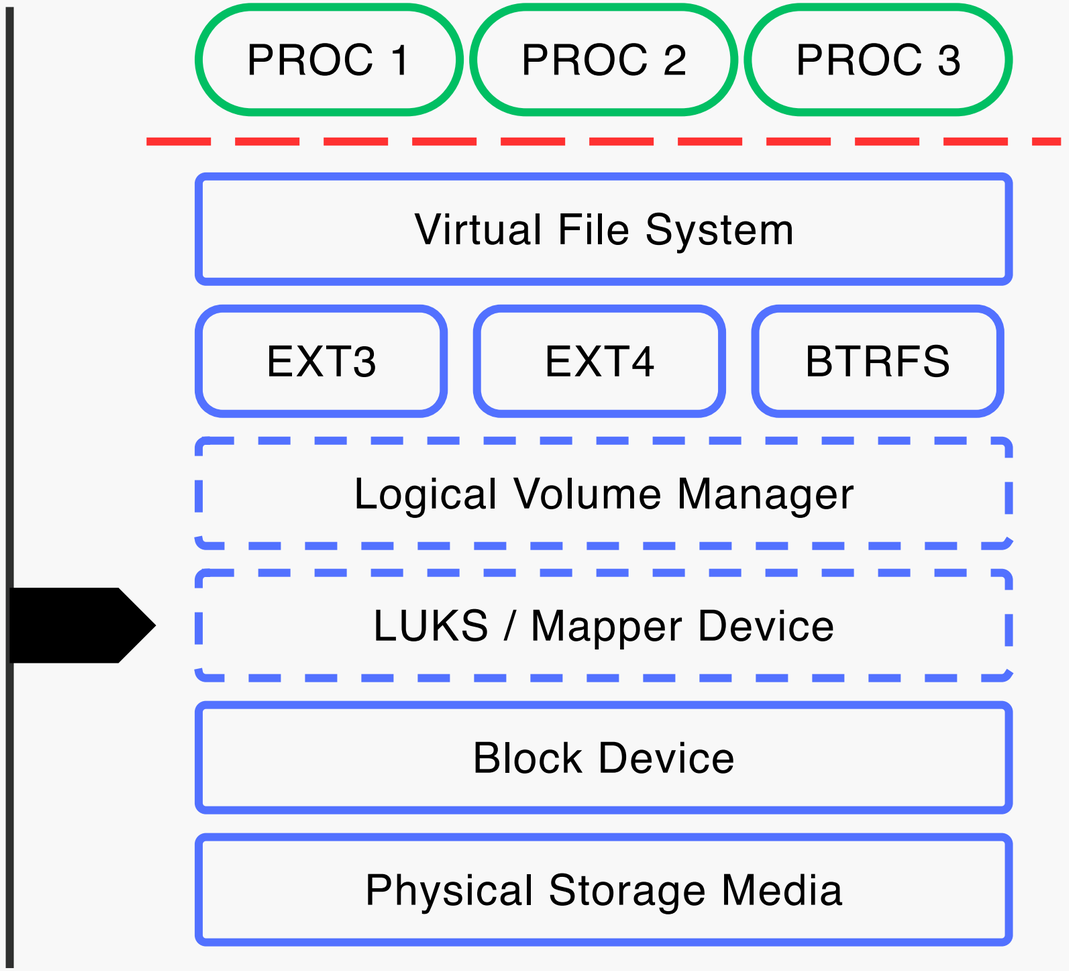
- Resizing of partitions
- Filesystem snapshots
- Multiple partitions of a single underlying storage device
- Single partition spanning multiple storage devices
- etc...



Linux Filesystem Stack

This layer provides an abstraction over physical block devices, and presents virtual block devices to Linux kernel. These virtual block devices feature full disk encryption.

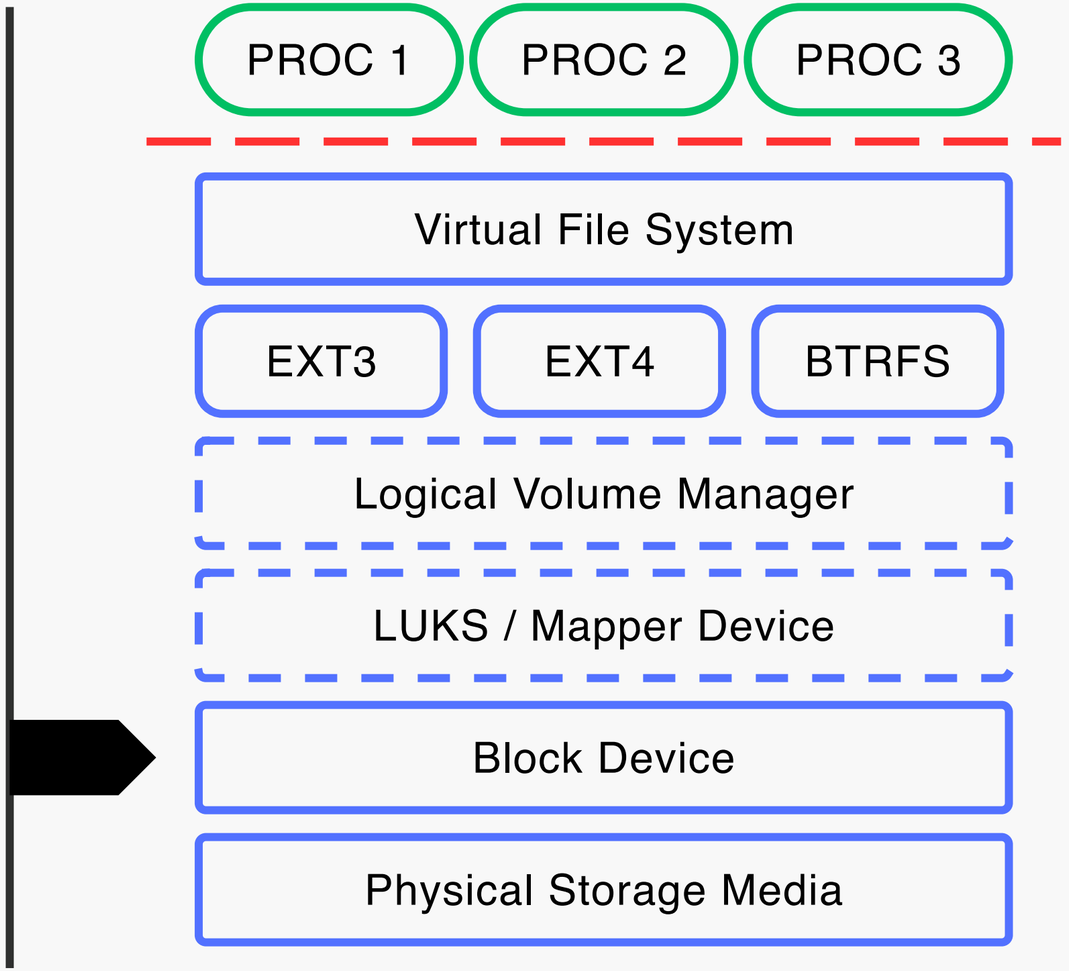
Anything above this layer works as-is, while data is being transparently encrypted/ decrypted from the underlying block device as needed.



Linux Filesystem Stack

A block device is a generic hardware device which provides access to data in blocks (continuous chunks of data), instead of providing access to individual bytes.

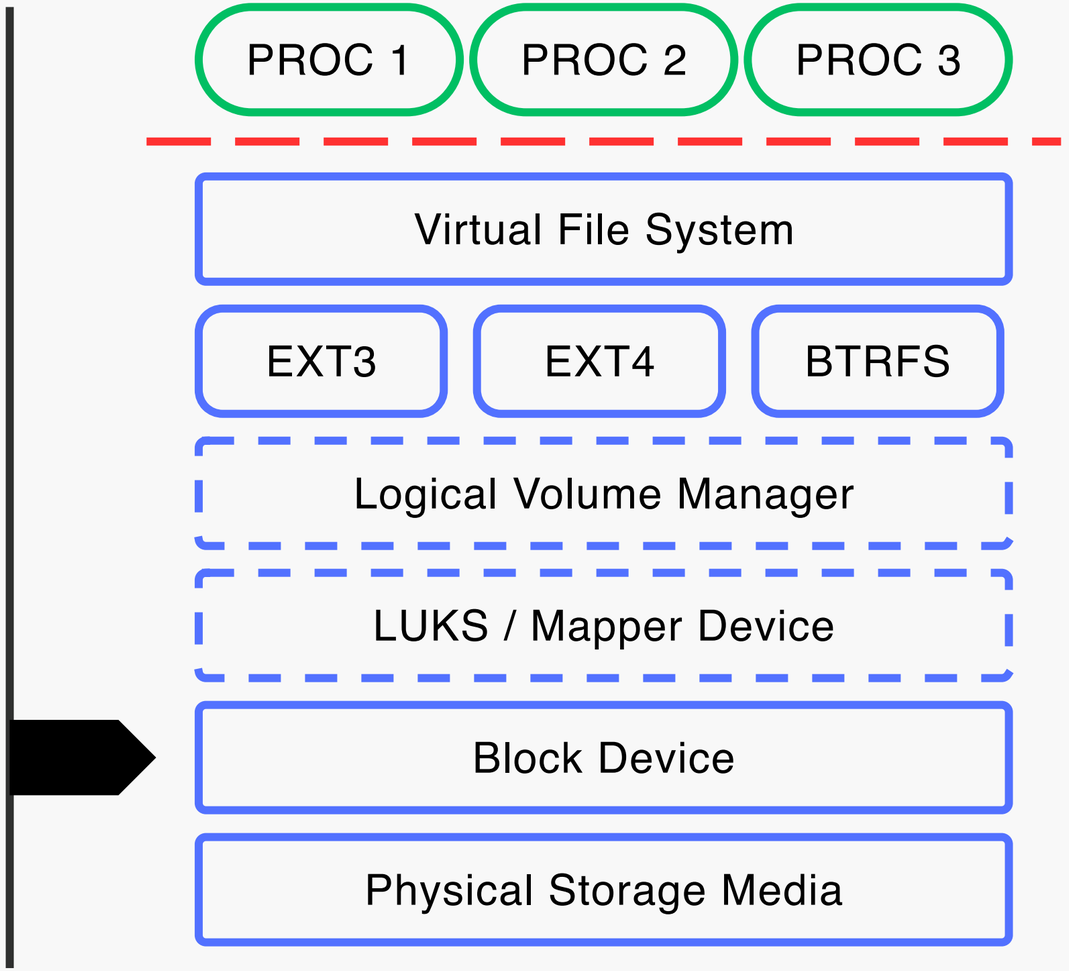
In other terms, for any read/write, we have to read/write whole block. We neither can read a single byte, nor can write a single byte at a time.



Linux Filesystem Stack

All underlying physical storage devices are exposed as generic block devices in Linux kernel.

This allows rest of the system to talk to any storage device in same way, irrespective of device specifics (USB, PCI/M.2/NVME etc.)

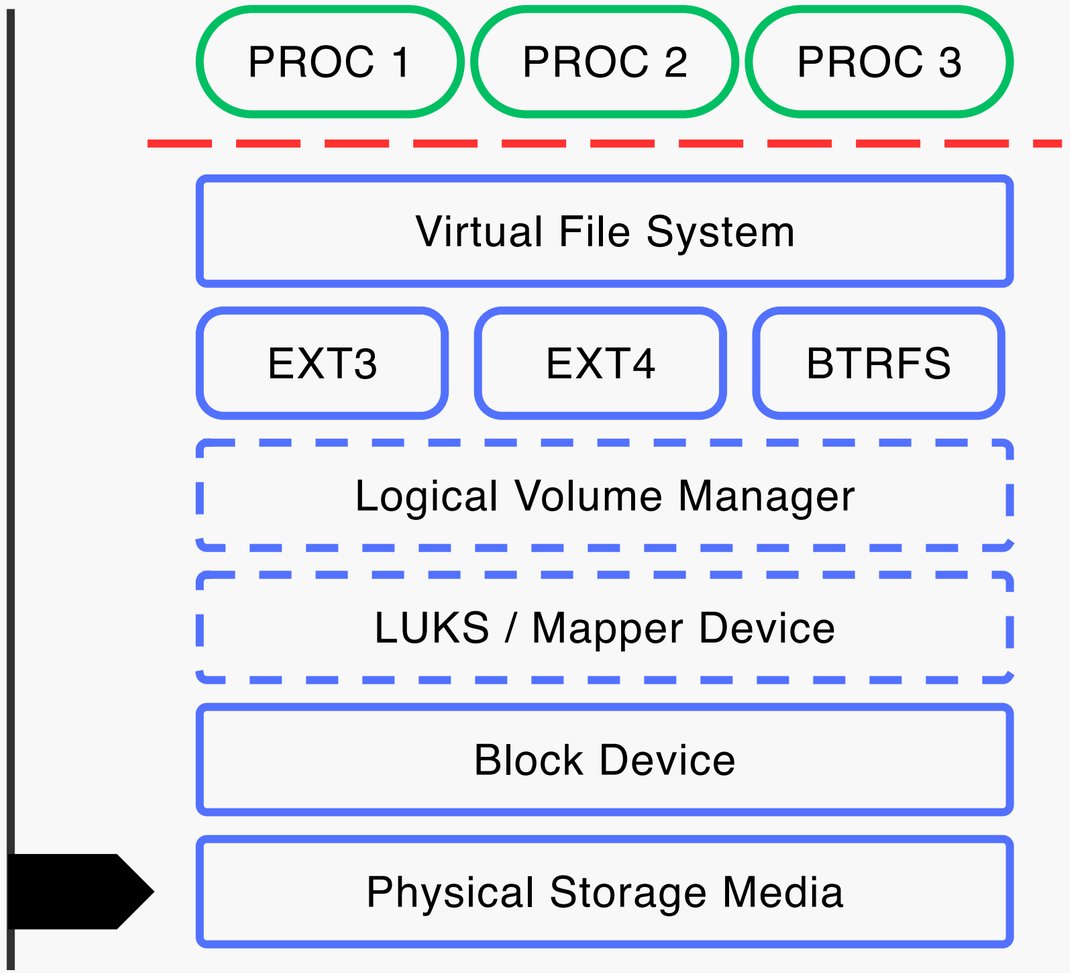


Linux Filesystem Stack

These are the physical storage medium attached to the system via any suitable bus.

This can be:

- USB storage device
- Hard disks
- NAND flash storage
- NVMe storage
- etc.



4. TYPES OF FS DRIVERS

4 Types of Filesystem Driver

Independent Filesystem Driver

- These drivers implement entire filesystem, and related functions.
- Run in kernel-space
- Can be part of kernel image itself
- Can be a loadable kernel module
- Example: drivers for EXT3, EXT4, BTRFS etc.

4 Types of Filesystem Driver

Stackable Filesystem Driver

- These drivers rely on other filesystem drivers to get a big chunk of functionality
- Generally used to add enhancement / features without having to write complete driver from scratch
- Can be used on top of multiple different filesystems
- Example: eCryptFS is implemented as a stackable filesystem which provides “encrypted file/folder” feature on top of another filesystem.

4 Types of Filesystem Driver

Filesystem In Userspace

- Commonly called FUSE
- A mechanism in Linux which allows unprivileged user-space programs create their own filesystems
- Filesystem implementation remains in user-space, and runs as a process
- Kernel interface is exposed as a “bridge” by FUSE kernel driver
- Example: sshfs, SMB etc are implemented on top of FUSE in Linux

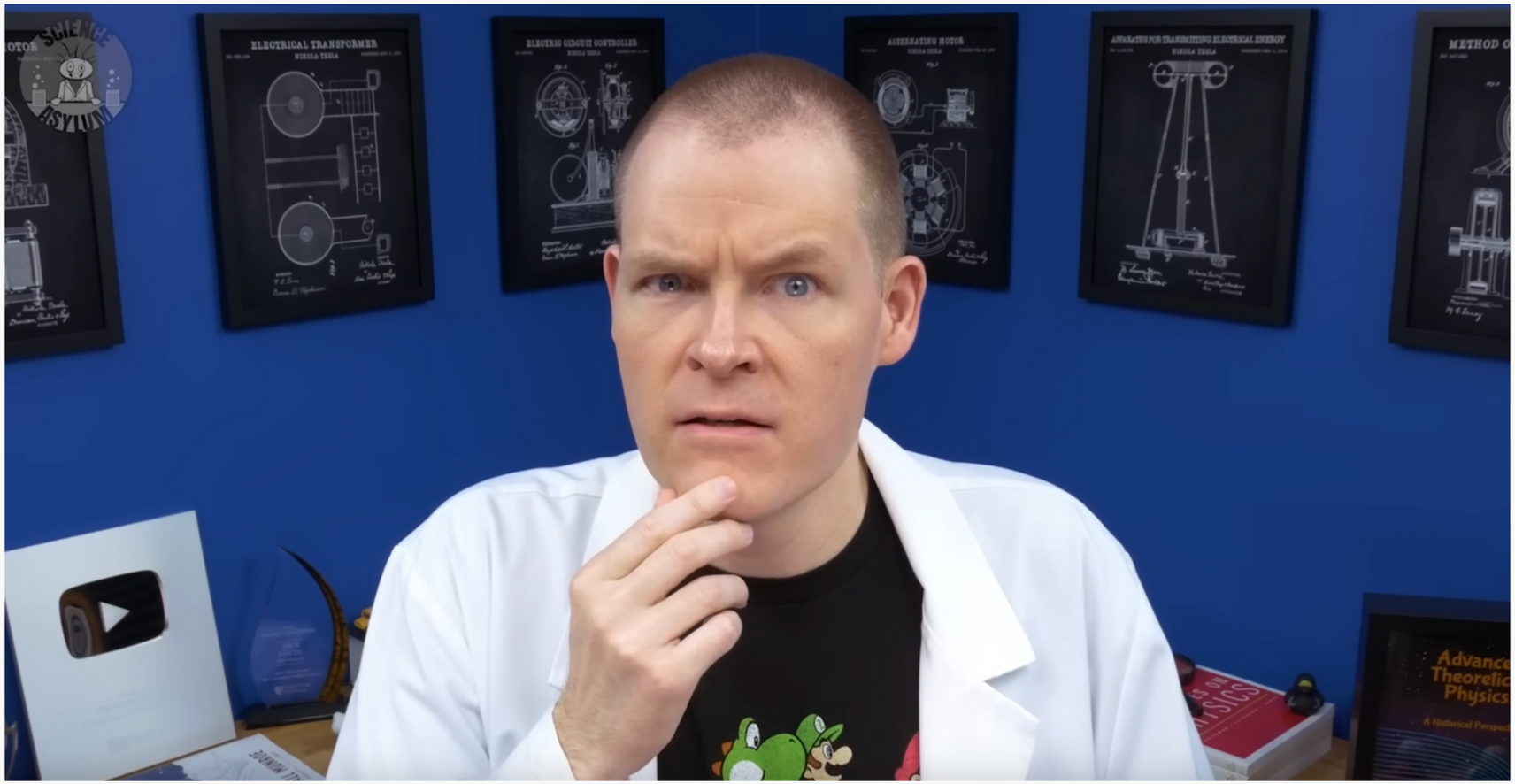
5. ROUGE FILESYSTEM AS AN ATTACKER TOOL

SOC एनालिस्ट: एक एक अटैक ब्लॉक करेगा मैं

Rogue Filesystem As An Attacker Tool

- Can be made extremely difficult to detect
- Can be used to selectively spoof file contents
- Can be used to selectively block access to certain files
- Can be used to selectively hide certain file/directories.

5 Rogue Filesystem As An Attacker Tool



That sounds suspiciously like rootkits hooking bunch of kernel stuff

5 Rogue Filesystem As An Attacker Tool

There are similarities, but:

- This does not require any hooking
  - Which means no entries in system call table will be pointing to some address outside kernel image itself.
  - Which means no function inside the core kernel itself will be changed / patched
  - Which means we do not really have to deal with memory protection in kernel-land, and raise all the alarms while disabling that (to plant the hook)

5 Rogue Filesystem As An Attacker Tool

There are similarities, but:

- This can be done in kernel-land as well as user-land
  - Swapping out good filesystem driver with rogue one
  - Installing a stackable filesystem on top of another filesystem
- This may not even require root privileges (for user-space)
  - That is the whole point of FUSE!

5 Rogue Filesystem As An Attacker Tool

6. WRITING A ROGUE FILESYSTEM DRIVER

Setting Up Filesystem Driver

```
static struct file_system_type hackerfs_fs_type = {
    .owner  = THIS_MODULE,
    .name   = "hackerfs",
    .mount  = hackerfs_mount,
    .kill_sb = kill_block_super,
    .fs_flags = FS_REQUIRES_DEV,
};
```

Setting Up Filesystem Driver

```
static int __init init_hackerfs(void)
{
    ...
    int err;
    err = register_filesystem(&hackerfs_fs_type);
    ...
}
```

Setting Up Filesystem Driver

- Other functionalities are configured from within the mount function.
- The mount function is called every time a disk with this filesystem is mounted.
- This function will mount the device, and will prepare superblock for the underlying filesystem.

Setting Up Superblock

```
static int hackerfs_fill_super(struct super_block *sb, void *data, int
silent)
{
    ...
    sb->s_maxbytes = /* maximum size */;
    sb->s_max_links = /* maximum links */;
    sb->s_op = &hackerfs_sops; // superblock operations are set here
    ...
}
```

Setting Up Superblock

In superblock operations, the following operations are to implemented:

- Allocate / free inode
- Write inode
- Evict inode
- Remount filesystem
- Stat filesystem (to get filesystem information)
- Synchronise filesystem (commit pending changes)
- Freeze / Unfreeze filesystem
- etc. etc.

Setting Up Superblock

We also have to implement the following group of operations

- Inode operations
- Directory operations
- File operations
- Address-space operations

Inode Operations

The following inode operations are to be implemented:

- create
- lookup
- link
- unlink
- symlink
- rename
- mknod
- etc. etc.

Inode Operations

```
const struct inode_operations hackerfs_dir_inode_operations = {
    .create  = hackerfs_create,
    .lookup  = hackerfs_lookup,
    .unlink  = hackerfs_unlink,
    .symlink = hackerfs_symlink,
    .rename  = hackerfs_rename,
    ... // other operations
};
```

Inode Operations

```
const struct inode_operations hackerfs_dir_inode_operations = {
    .create  = hackerfs_create,
    .lookup  = hackerfs_lookup,
    .unlink  = hackerfs_unlink,
    .symlink = hackerfs_symlink,
    .rename  = hackerfs_rename,
    ... // other operations
};
```

This is called by creat() and open() system calls, and returns a file inode for given file. File operations can be called on this file inode.

Inode Operations

```
const struct inode_operations hackerfs_dir_inode_operations = {
    .create  = hackerfs_create,
    .lookup  = hackerfs_lookup,
    .unlink  = hackerfs_unlink,
    .symlink = hackerfs_symlink,
    .rename  = hackerfs_rename,
    ... // other operations
};
```

This is called to search a directory for an inode corresponding to a file name specified in a given dentry.

Inode Operations

```
const struct inode_operations hackerfs_dir_inode_operations = {
    .create  = hackerfs_create,
    .lookup  = hackerfs_lookup,
    .unlink  = hackerfs_unlink,
    .symlink = hackerfs_symlink,
    .rename  = hackerfs_rename,
    ... // other operations
};
```

This is used to delete the inode specified by given dentry from directory.

Inode Operations

```
const struct inode_operations hackerfs_dir_inode_operations = {
    .create  = hackerfs_create,
    .lookup  = hackerfs_lookup,
    .unlink  = hackerfs_unlink,
    .symlink = hackerfs_symlink,
    .rename  = hackerfs_rename,
    ... // other operations
};
```

This is used to create a symbolic link which points to another given file path.

Inode Operations

```
const struct inode_operations hackerfs_dir_inode_operations = {
    .create  = hackerfs_create,
    .lookup  = hackerfs_lookup,
    .unlink  = hackerfs_unlink,
    .symlink = hackerfs_symlink,
    .rename  = hackerfs_rename,
    ... // other operations
};
```

This is used to rename file name for a given inode to new given name.

Directory Operations

Since directories also have corresponding inodes, many inode operations can be re-used for them with little changes. Some of the operations, which only make sense for directories, are to implemented separately.

Example: reading directories, iterating over directory contents etc.

File Operations

These operations deal with file related operations like:

- open
- read / write
- read_iter / write_iter
- flush / release
- flock
- fsync
- etc. etc.

File Operations

```
const struct file_operations hackerfs_file_operations = {
    .open = hackerfs_file_open,
    .read = hackerfs_file_read_sync,
    .write_iter = hackerfs_file_write_iter,
    .release = hackerfs_release_file,
    ...
};
```

File Operations

```
const struct file_operations hackerfs_file_operations = {
    .open = hackerfs_file_open,
    .read = hackerfs_file_read_sync,
    .write_iter = hackerfs_file_write_iter,
    .release = hackerfs_release_file,
    ...
};
```

This is called by open() system call to notify the driver that file operations will be performed on the file in future. Please note that, file is not actually opened here by driver. It is merely a notification call, and it is optional to be implemented.

File Operations

```
const struct file_operations hackerfs_file_operations = {
    .open = hackerfs_file_open,
    .read = hackerfs_file_read_sync,
    .write_iter = hackerfs_file_write_iter,
    .release = hackerfs_release_file,
    ...
};
```

This function, and its counterpart for writing, implement blocking read/write actions for a given file. These are called from read() and write() system calls respectively.

File Operations

```
const struct file_operations hackerfs_file_operations = {
    .open = hackerfs_file_open,
    .read = hackerfs_file_read_sync,
    .write_iter = hackerfs_file_write_iter,
    .release = hackerfs_release_file,
    ...
};
```

This function, and its counterpart for writing read_iter(), implement asynchronous / non-blocking read/write actions for a given file.

File Operations

```
const struct file_operations hackerfs_file_operations = {
    .open = hackerfs_file_open,
    .read = hackerfs_file_read_sync,
    .write_iter = hackerfs_file_write_iter,
    .release = hackerfs_release_file,
    ...
};
```

This is called when file structure is being released. This may not happen after each call to close(), if same structure is shared at multiple places (fork/clone/dup). In that case, it will be called once all copies are closed.

Address-space Operations

These operations generally deal with page level reads/writes, and other low level details. Generally the following operations are implemented here:

- write_begin / write_end
- readpages / writepages
- direct_IO
- Page folio operations
- etc. etc.

Address-space Operations

```
const struct address_space_operations hackerfs_aops = {
    .write_begin = hackerfs_write_begin,
    .write_end  = hackerfs_write_end,
    .writepage  = hackerfs_writepage,
    ...
};
```

Address-space Operations

```
const struct address_space_operations hackerfs_aops = {
    .write_begin = hackerfs_write_begin,
    .write_end  = hackerfs_write_end,
    .writepage  = hackerfs_writepage,
    ...
};
```

This is called by VFS to notify the driver that VFS intends to write data into given page. It is filesystem driver's responsibility to ensure that write will work. Driver will allocate a page, lock it, and return to VFS.

Address-space Operations

```
const struct address_space_operations hackerfs_aops = {
    .write_begin = hackerfs_write_begin,
    .write_end  = hackerfs_write_end,
    .writepage  = hackerfs_writepage,
    ...
};
```

This call notifies that data has been copied in page by VFS, and now those contents can be committed to filesystem. Post commit, page is unlocked, and size is updated.

Address-space Operations

```
const struct address_space_operations hackerfs_aops = {
    .write_begin = hackerfs_write_begin,
    .write_end   = hackerfs_write_end,
    .writepage   = hackerfs_writepage,
    ...
};
```

This call is made by virtual memory system (not VFS), when it needs to flush a memory page from its cache. It is expected that page will be committed to physical storage, and associated storage will be reclaimed.

Stackable Filesystem Driver

It is largely similar to full driver, except that many operations will be handed over to some other driver via VFS layer to deal with. The stacked driver either can hand over calls as-is, or can do some of the stuff on its own (like encryption/decryption etc.). Hand over is done by calling functions like:

- `vfs_create`
- `vfs_read`
- `vfs_write`
- `etc. etc.`

क्या मतलब इतना सारा कर्नल प्रोग्रामिंग करना पड़ेगा?

6 Writing A Rogue Filesystem Driver

Filesystem In Userspace

Instead of having separate operations for inodes, directories, files, address-space etc., we only need to implement file I/O related operations and supply corresponding pointers in fuse_operations.

Filesystem In Userspace

```
struct fuse_operations {
    int (*mkdir) (const char *, mode_t);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t,
       struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,
        struct fuse_file_info *);
    ...
};
```

Filesystem In Userspace

Just like kernel mode, the FUSE based driver can either have a filesystem on top of a block device, or it can be a stackable filesystem on top of another filesystem.

Most of the FUSE based projects work as stackable filesystem as it keeps implementation easy (a lot of hard work can be outsourced to another filesystem driver to deal with).

Once the functionalities are implemented, we can pass the pointers to FUSE, and let it deal with rest of the stuff.

Filesystem In Userspace

```
static struct fuse_operations operations = {
    .getattr = do_getattr,
    .readdir = do_readdir,
    .read = do_read,
    ...
};
int main( int argc, char *argv[] ) {
    return fuse_main( argc, argv, &operations, NULL);
}
```

Filesystem In Userspace

Just like kernel mode, the FUSE based driver can either have a filesystem on top of a block device, or it can be a stackable filesystem on top of another filesystem.

Most of the FUSE based projects work as stackable filesystem as it keeps implementation easy (a lot of hard work can be outsourced to another filesystem driver to deal with).

Once the functionalities are implemented, we can pass the pointers to FUSE, and let it deal with rest of the stuff.

Going Rouge: Kernel-mode Driver

To plant a backdoor, we need to find some interesting places where we can keep an eye of file paths and inodes both. We also have to create a mapping between file path and corresponding inode, as all actual I/O activities happen on inodes.

When a file is opened, open() and creat() system calls can be used. Both of these eventually end up calling inode_create() and inode_lookup() from corresponding filesystem driver (via VFS layer).

Going Rouge: Kernel-mode Driver

inode_create() has the following function signature:

```
int (*create) ( struct mnt_idmap *idmap,
                struct inode *directory_inode,
                struct dentry *fs_dentry,
                umode_t mode,
                bool excl);
```

Going Rouge: Kernel-mode Driver

inode_create() has the following function signature:

```
int (*create) ( struct mnt_idmap *idmap,
                struct inode *directory_inode,
                struct dentry *fs_dentry,
                umode_t mode,
                bool excl);
```

File name can be found in fs_dentry->d_name.

Going Rouge: Kernel-mode Driver

inode_lookup() has the different function signature:

```
struct dentry * (*lookup) ( struct inode *fs_dir_inode,
                            struct dentry *fs_dentry,
                            unsigned int flags);
```

Going Rouge: Kernel-mode Driver

inode_lookup() has the different function signature:

```
struct dentry * (*lookup) ( struct inode *fs_dir_inode,
                            struct dentry *fs_dentry,
                            unsigned int flags);
```

File name can be found in fs_dentry->d_name.

Going Rouge: Kernel-mode Driver

We can create a mapping between filename and corresponding inode as shown below:

- Allocate a memory buffer with sufficient length using kmalloc()/vmalloc() etc.
- Get the full file path using dentry_path_raw()
- Get the inode (in inode_create() it will be returned value, otherwise inode of existing file can be captured from corresponding dentry struct).
- Acquire mutex lock on the data structure where you will be storing the mapping (mutex_lock())
- Store the pair (filepath, inode) if it does not exist already.
- Release the lock (mutex_unlock()).

Going Rouge: Kernel-mode Driver

Depending upon how deep we want to go, we can also put rogue code in implementations for:

- read(): return bogus data to usermode
- write(): alter the data being written
- readdir(): hide directories / show phantom directories
- stat(): spoof file metadata etc.

Going Rouge: User-mode Driver

For FUSE based drivers, our task becomes far easier, because FUSE operations are like system call counter-parts.

Since here operations are going to happen on file descriptors, we will need to create a mapping between file path and corresponding file descriptor.

Going Rouge: User-mode Driver

Creating a mapping between file path and file descriptor:

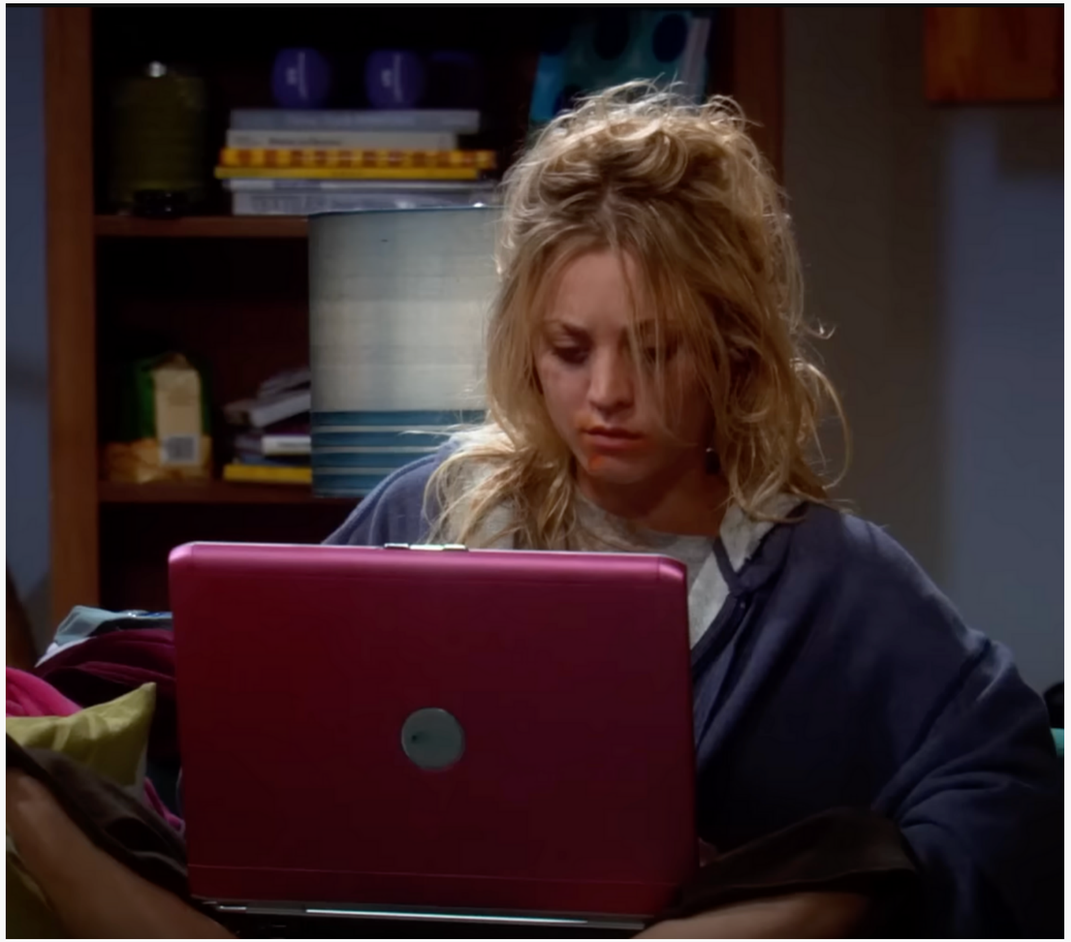
- Copy file path from parameters to open() implementation in FUSE driver.
- Copy the file descriptor at end of the function. This is the value that is to be returned from this function.
- Acquire mutex lock on the data structure where you will be storing the mapping
- Store the pair (file path, file descriptor) if it does not exist already.
- Release the lock.

Going Rouge: User-mode Driver

Similar to kernel drivers, other FUSE operations can be backdoored for similar effects:

- open(): refuse to open certain files for certain processes
- read(): return bogus data to usermode
- write(): alter the data being written
- readdir(): hide directories / show phantom directories
- stat(): spoof file metadata etc.

ड्राइवर लिखते लिखते प्रोग्रामर की हालत



7. ATTACKING THE USER-SPACE

Attacking The User-space

Since we now know more than one place to backdoor a filesystem driver, we can have a lot of fun.

- Evasion from static scans
- Evasion from scan on startup (file)
- Derailing process file dump / copy
- Rootkit like things (hiding files/directories)
- Fooling various rootkit detection techniques
- etc. etc.

Evading From File Scans

Basic idea is to silently “swap” the file for user-space for all unknown processes.

- Get PID of the process which is trying to access the file
  - From kernel driver, you can get PID from context of current process
  - For FUSE based driver, you can get PID from fuse_context structure
- Is the process associated with PID is known?
- If yes, process the request as usual.
- If no, return inode (in case of kernel mode driver), or file descriptor (for FUSE based driver) for other file.

Evading From File Scans

End results:

- When it is time for malware to start after boot (because persistence), filesystem driver supplies correct file.
  - Malware process starts just fine.
  - /proc/<malware pid>/exe points to the malware file path

Evading From File Scans

End results:

- When there is something else which tries to access same file, a “fake file” is used instead.
  - On-disk scanning scans this fake file, and finds nothing.
  - Security tool which wants to scan executable, ends up scanning the fake file instead.
  - Security tool which scans executables for all running processes also ends up scanning the fake file.
    - Because /proc/<malware PID>/exe is nothing more than a symlink.
    - And the actual file for above symlink has been changed.

Covering Tracks

Just like any other kernle level rootkit, we can hide files and directories here.

- Alter return values from inode_create / inode_lookup etc.
- Alter return values from readdir() etc.

Basically, instead of hooking corresponding system calls, and putting our filtering logic there, we put the same logic inside filesystem driver itself.

Fooling Rootkit Detection

There are basically three methods of rootkit detection:

- Detecting tampering in system call table
- Detecting hooks in kernel symbols
- Detecting discrepancies in filesystem

Fooling Rootkit Detection

Detecting tampering in system call table

- No entry in system call table should be out of known good memory address range. Anything pointing outside it (e.g. somewhere in another kernel module) is suspicious.

Fooling Rootkit Detection

Detecting tampering in kernel symbols

- Check beginning and end of desired kernel function. If there are unconditional jumps to somewhere out of the kernel image (or specific kernel module, if symbol is coming from a module); it is most probably hooked.

Fooling Rootkit Detection

Detecting discrepancies in filesystem:

- Iterate over filesystem contents (files and directories) using the normal way (glibc calls or using system calls). Make a list of this data.
- Iterate over contents of same filesystem, but using your own driver. Make a list of this data.
- Iterate over contents of same filesystem, but doing so by reading the block device directly. Make a list of this data.

Fooling Rootkit Detection

- If there are no rootkits, all three lists will match.
- If there is a mismatch (specially if first list contains less items), maybe a rootkit is present to hide files/directories.
  - If first list contains more items, most probably your code is wrong.
  - Does not play nice with stackable filesystems, because those are expected to change how filesystem looks like to userspace.

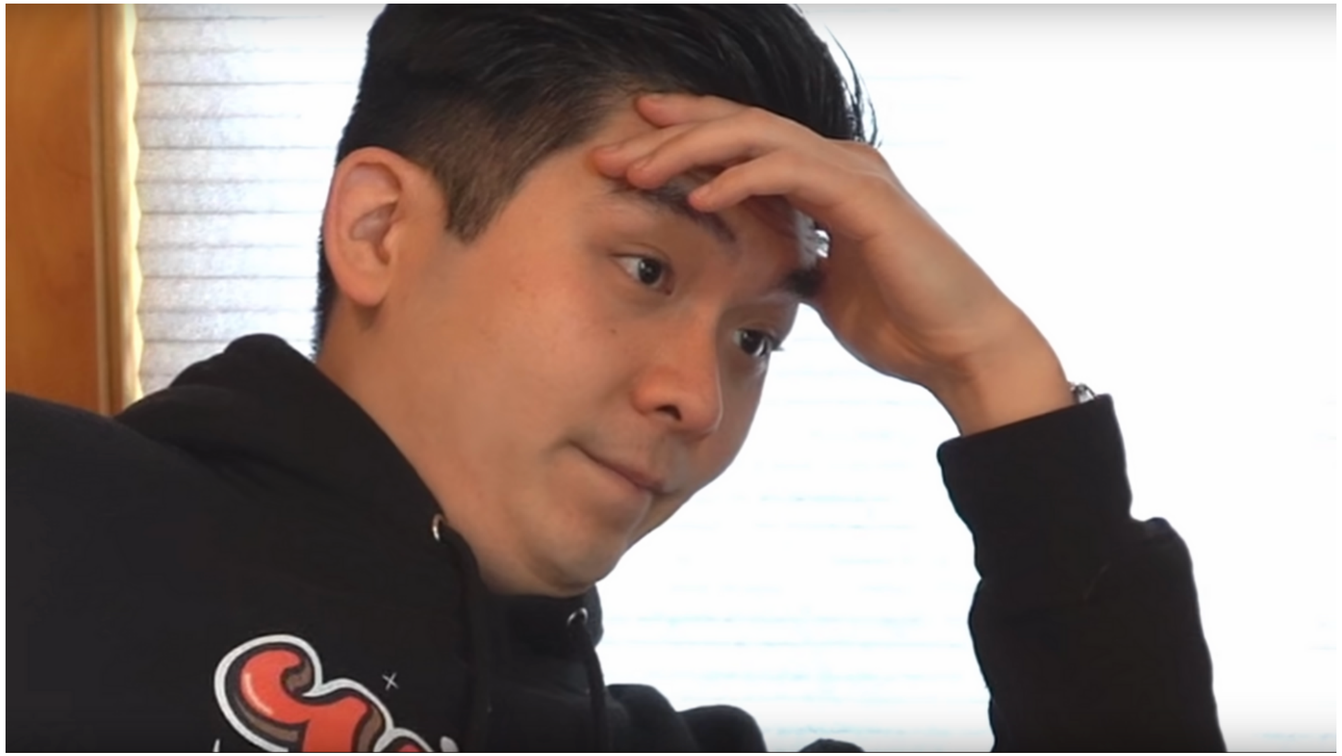
Fooling Rootkit Detection

- Since we are planting a backdoor in a filesystem driver, we are not hooking anything.
  - Which means, hook detection is not going to reveal anything.
- Instead of hiding malware file on filesystem, we are simply swapping it out with another non-malicious file on the same filesystem.
  - Which means, no matter how you iterate over filesystem contents, you will get the same set of files and directories.

Fooling Rootkit Detection

- For hiding a malware binary on filesystem, we can write that file and all associated metadata on unallocated space in the filesystem.
  - Maybe encrypt whole thing so that it looks like random gibberish.
- Since only rouge filesystem driver knows where to find it, and how to read it back, using legitimate filesystem driver will not reveal anything extra.
- Similarly, parsing filesystem structures by reading block device directly will also not reveal anything extra.

हिमालय पर जाने की योजना बनाता SOC एनालिस्ट



8. LIMITATIONS

Limitations

- Kernel level programming:
  - Is finicky (because kernel API and ABI both are unstable)
  - Consumes a lot of time
  - Requires way more effort than anything else
- Kernel modules are tightly coupled with their target kernel versions
  - Module compiled for one kernel version/build will not work with another version/build
  - Separate build for each targeted version/build is necessary

Limitations

- Loading a kernel module requires elevated privileges (typically root).
- In some cases, some other configuration modifications may be necessary (e.g. if we do not want to nuke the original driver)
- FUSE based filesystems are not standalone. These cannot be used in very early stages of boot (deal breaker if we want to change driver for root partition).

9. DETECTING & PREVENTING ROGUE FILESYSTEMS

Detections

- Monitor the following processes
  - **modprobe**
  - **insmod**
  - rmmod
  - lsmod
  - modinfo

Detections

- Monitor the following system calls:
  - **create_module**
  - **init_module**
  - delete_module
  - query_module
  - **finit_module**

Detections

- Monitor the following paths:
  - /dev/fuse

Preventions

- If you do not really need to load/unload kernel modules, consider disabling that “feature” completely.
- If you need that ability, use secure boot, and setup your own cryptographic signing infrastructure to sign kernel image and modules.
  - Block kernel modules which are not signed with correct key.
- Block access to /dev/fuse, except for specific processes which you use (and know that FUSE is required for whatever you do)

10. EXTERNAL REFERENCES / RESOURCES

10 External References / Resources

- https://lwn.net/Articles/849538/
- https://lwn.net/Articles/756625/
- https://lwn.net/Articles/932079/
- https://lwn.net/Articles/937433/
- https://www.kernel.org/doc/html/next/filesystems/vfs.html
- https://static.lwn.net/kerneldoc/filesystems/api-summary.html
- https://www.kernel.org/doc/html/next/filesystems/fuse.html

धन्यवाद

THANK YOU