The Hitchhiker's Guide to Building an Encrypted Filesystem in Rust

BEGINNING: It all started after I began learning **Rust** and wanted an interesting learning project to keep me motivated. Initially, I had some ideas, then consulted **ChatGPT**, which suggested apps like a **Todo list** :) I pushed it to more interesting and challenging realms, leading to suggestions like a **distributed filesystem**, **password manager**, **proxy**, **network traffic monitor**... Now these all sound interesting, but maybe some are a bit too complicated for a learning project, like the distributed filesystem.

IDEA: My project idea originated from having a work directory with projects information, including some private data (not credentials, which I keep in **Proton Pass**. I synced this directory with **Resilio** across multiple devices but considered using **Google Drive** or **Dropbox**, but hey, there is private info in there, not ideal for them to have access to it. So a solution like **encrypted directories**, keeping the **privacy**, was appealing. So I decided to build one. I thought to myself this would be a great learning experience after all. **And it was indeed**.

From a learning project it evolved into something more and soon will be releasing stable version with many interesting features. You can view the project `https://github.com/radumarias/rencfs`.

FUSE: I used it before and I could use it to expose the filesystem to the OS to access it from **File Manager** and **terminal**. I looked for *FUSE* implementations in Rust and found **fuser**, and later migrating to **fuse3** which is **async**. I began with its examples.

IN-MEMORY-FS: I started wth a simple **in-memory** FS using *FUSE*, where I learned more about **smart pointers** like **Box**, **Rc**, **RefCell**, **Arc** and **lifetimes**. *Aargh... lifetimes*, would say many, one of the *most complicated concept* in Rust, after the **Borrow-Checker**. They are quite complicated, at first but after you fight them for a while, you bury the hatchet and are easier to live with. After you understand how and why the compiler lets you do things, you understand that's the correct way to do it and it saves you from a lot of problems, and you appreciate it. After all, these are the promises of Rust, **memory safety**, **no data race** and reduces **race conditions**. And indeed it lives to its premise. You need to come from other languages where you had all sort of problems to really appreciate what Rust is offering you.

STRUCTURE: I started with a simple one that keeps the files in *inode structure*, each metadata is stored in **inodes** dir in a file with *inode*'s name and in *contents* directory we have files with *inode's* name with the actual content of the file.

MULTI-NODE: We need to run in multi node, as the folder will be synced over several devices the app could run in **parallel** or even **offline**. We need to generate unique **inode**s for new files. Solution is to assign a **instance_id** as a *random id* to each **device** (or safer to set by command arg) and generate as **instance_id | inode_seq**, where **inode_seq** is a sequence/counter for each device.

SECURITY: We do the same for **nonce**, **instance_id | nonce_seq**. The **sequence**s we keep in **data_dir** in a per instance folder. To resolve problem where user *restores a backup* and hence would **reuse nonces** and reuses *inodes* (which ends up in catastrophic failure) we keep sequences in **keyring** too and use **max(keyring, data_dir)**. **Limits**: if the **instance_id** is **u8**, the max *inode* (**u64**) it's reduced to **2^{56} - 3**. It's -3 and not -1 because inode 0 is not used, and 1 is reserved for root dir, so we're left with value **72,057,594,037,927,933**. And max data to encrypt **2^{56} - 1 * 256 KB**, which is **1.845×10^{19} petabytes**.

Using **ring** for **encryption**, will extend to **RustCrypto** too, which is **pure Rust**. First time we generate a random **encryption key** and encrypt that with another key **derived from user's password** using **argon2**. We use only **AEAD** ciphers, **ChaCha20Poly1305** and **Aes256Gcm**. **Credentials** are kept in mem with **secrecy**, **mlock**ed when used, **mprotect**ed when not read and **zeroize**d on *drop*. **Hashing** is made with **blake3** and **rand_chacha** for random numbers.

DATA-PRIVACY: We aim to offer **truly privacy** and for that we need to make sure we **hide** all **metadata**, **content**, **file name**, **file size**, ***time** fields, **files count**, **directory structure** and that all of these are encrypted. Filename and content are easier to hide, we just encrypt then and **pad filenames** to fix size and we're fine. But files size, files count, *time fields, directory structure are not trivial. For that we **split** the file in **chunks** and each is like an **item** in a **LinkedList** on disk with **next** pointer kept encrypted inside **chunk file content**. This hides the actual files count, but to hide it even more we add dummy nodes a the beginning with random data. Also we add dummy random data to each chunk at the beginning (as it's easier to skip) so we hide even more the file size. All these hides file size, files count and *times fields. This creates a problem, how to get to the root chunk files (nodes) without an attacker being able to to the same, given our code is publicly available on GitHub. For that we keep an **index** file with all **root chunk files** (inodes actually). What's remaining is **directory structure** in the sense of the directories inside another directory. For this we do similarly, we create dummy folders with random names so we hide how many actual directories are there and we keep all these in the index file.

FILE-INTEGRITY: *"There's The Great Wall, and then there's this: an okay WAL."*. **WAL**(Write-ahead logging) is a very common technique used in *DBs* world for writing transactions to ensure file integrity. I'm using **okaywal**.

SEEK: To support **fast seeks** we encrypt file in **blocks** of **256KB**. When we need to **seek on read** we translate from *plaintext* **offset** to *ciphertext* **block_index**, and **decrypt** that block. We actually **impl Seek** on the same **Read struct**. For **seek on write** it's a bit more complicated, we need to act as reader too. First we need to *decrypt* the block then write to it and when at the end of the block **encrypt** the block and *write* it to disk. Because Rust doesn't have **method overwriting** the code is not as clean as for reader where, we only extend.

WRITES-IN-PARALLEL: Using **RwLock** we allow reading and writing in parallel and we resolve conflicts with **WAL**. Particularly useful for torrent apps which writes different chunks in parallel, but also for DBs.

STACK: Fully **async** upon **tokio**, **fuse3**, **ring** for encryption, **argon2** for **KDF** (deriving key used to encrypt master encryption key from user's password), **blake3** for hashing, **rand_chacha** for random generators, **secrets** for keeping pass and encryption keys safe in memory, **mlock** on use, **mprotect** when not read and **zeroize** on **drop**. To mitigate **cold boot attack** we keep encryption keys in memory only while being used, and on idle **zeroize** and **drop**, **password** saved in OS **keyring** using **keyring**, **tracing**.