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 stay motivated. Initially, I had some ideas and consulted **ChatGPT**, which suggested common apps, like a **Todo list** :) However, 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 some are maybe 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 **KeePassXC**. 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. This would be a great learning experience after all. **And it was indeed**.

From a learning project it evolved into something more and soon ready for a stable release 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 **inmemory** 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 concepts* in Rust, after the **Borrow-Checker**. They are quite complicated, at first but after you fight with them for a while, then you bury the hatchet and they are easier to live with. After you understand how and why compiler lets you do things, you understand that's the correct way to do them 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 **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 *random id* to each **device** (or set by command arg) and generate as **instance_id |inode_seq**.

SECURITY: Same we do 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 an backup* and hence would **reuse nonces** and *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 **288230376151711743** and max *nonce* to **7.923E+16 PB** (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 use **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.

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 by *DBs* to write transactions to ensure fie integrity. I'm using **okaywal** for that.

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 **encrypt** 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 **locks** 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.

DATA-PRIVACY: All **metadata are encrypted** and file *chunks* have **random names** and form a **linked list** on disk with *next* pointer kept encrypted in file. Like that we ensure full privacy.

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** for tracing and logs. In future, support for **macOS**, **Windows** and **mobile**.