Linux filesystem, permissions and agent safety

DIY SMART CODE
Thomas — @DIYSMARTCODE

Linux Filesystem Explained Through a Real AI Coding Project — coming soon

https://www.youtube.com/@DIYSMARTCODE

ONE TREE // MANY JOBS

The filesystem is a map

Every directory has one defined job. Once you know the jobs, you can predict where any file lives and whether you are allowed to touch it.

USER WORK

`/home`	Personal directories, one per user. All your documents, projects and settings live under your own folder here.
`/home/thomas`	Your personal directory in this sandbox. Typing `cd` with no argument brings you here, and `~` is shorthand for this path.
`/root`	The home directory of the administrator account (the user called root). Despite the name it is not `/`, and normal users cannot enter it.

CONFIG

`/etc`	System-wide settings, stored as plain text files owned by root. Programs read their configuration from here when they start. A change here affects every user on the machine.
`/etc/hosts`	A text file that maps computer names to network addresses. The system checks it before asking a DNS server.
`/etc/ssh`	Settings for the SSH service, which allows remote logins to this machine. Misconfiguring it can lock you out or let others in, so read it but do not edit it casually.

SOFTWARE

`/usr`	Installed software: the programs themselves and the files they need. Your package manager (`apt`, `dnf`, ...) puts things here — you rarely edit it by hand.
`/usr/bin`	The actual program files behind the commands you type. When you run `ls` or `cat`, the file being executed is in this directory.
`/usr/lib`	Code libraries that installed programs share, so the same code is not copied into every program.
`/usr/share`	Non-program files that installed software needs: icons, manual pages, translations.

`/usr/local`	Software you compiled or installed manually, kept apart so the package manager never overwrites it during an update.
`/opt`	Third-party applications that ship as one self-contained folder (for example `/opt/google/chrome`) instead of spreading their files across `/usr`.

CHANGING DATA

`/var`	Files that the system writes to and that grow over time: logs, caches, databases. When a disk mysteriously fills up, the culprit is usually somewhere in here.
`/var/log`	Log files: the record of what programs and the system have been doing. The first place to look when something fails — but logs can also contain passwords and tokens that got printed by accident.
`/var/cache`	Downloaded or computed data that programs keep to save time, and can recreate if it is deleted.
`/var/lib`	Working data that services must not lose between restarts: database contents, the package manager's record of what is installed.
`/tmp`	Temporary files. The system deletes them at reboot or after a few days, so never store anything here that you want to keep.
`/srv`	Files this machine serves to other computers, such as a website's files on a web server. On a desktop it is usually empty.

REAL-TIME STATE

`/dev`	Your hardware, represented as files: disks, keyboards, screens. Programs read and write these files to talk to devices. `/dev/null` is the famous one — anything written to it is discarded.
`/proc`	Live information about running programs and the kernel, presented as files. `cat /proc/uptime` shows how long the system has been running. None of it exists on disk — it is generated the moment you read it.
`/run`	Small bookkeeping files that services create while running: their process ids, sockets and locks. Emptied at every boot.
`/sys`	Detailed information about your hardware and its drivers, arranged as a browsable tree of files. Like `/proc`, generated by the kernel rather than stored.

CONNECTED BRANCHES

`/`	The root: the single starting point of the whole filesystem. Linux has no drive letters — every disk, USB stick and network share appears as a directory somewhere under `/`.
`/mnt + /media`	Where extra storage appears in the tree. `/media` is used by the desktop when you plug in a USB drive; `/mnt` is for storage you attach manually with the `mount` command.
`/boot`	The files needed to start the operating system, including the kernel itself. Breaking something here can make the machine unbootable, so look but do not touch.

- On most current distributions `/bin`, `/sbin` and `/lib` are no longer real directories — they are symbolic links (pointers) to `/usr/bin`, `/usr/sbin` and `/usr/lib`. They exist only so that old scripts with old paths keep working.
- Run `ls -l /bin` to check: if you see an arrow like `/bin → usr/bin`, your system has merged them. Not every distribution has, so verify rather than assume.
- A path starting with `/` is absolute: it names the same file no matter where you are. Any other path is relative to your current directory, where `.` means "this directory" and `..` means "one level up".
- File names are case-sensitive by default: `Deploy.sh` and `deploy.sh` are two different files that can sit side by side. (A filesystem can be configured to ignore case, but do not count on it.)
- Files whose names start with a dot (`.bashrc`, `.env`) are hidden: a plain `ls` skips them. `ls -a` lists them too. Programs keep their per-user settings in these dotfiles.

READ THE PERMISSION // BOUND THE ACTION

Permission is a mechanical rule

Every "Permission denied" has an exact cause: one missing letter, on one specific path. This section shows how to read the permission line and find that letter yourself.

`-`	The first character is the file type: `-` means regular file, `d` directory, `l` symbolic link (a pointer to another path), `c` or `b` a device.
`rw-`	Characters 2-4: what the owner of the file may do. Here `thomas` may read and write, but not execute — each position is r, w, x, with `-` meaning "not allowed".
`r--`	Characters 5-7: what members of the file's group may do. Here anyone in the `dev` group may read the file, and nothing else.
`r--`	Characters 8-10: what everybody else on the machine may do. Here: read only.
`thomas`	The owner — a single user, normally whoever created the file. The owner is the only one who can change its permissions.
`dev`	The group — a named set of users. Groups let several people share access to a file without opening it to everyone.
`r`	On a file: you may read its contents. On a directory: you may list the file names inside it.
`w`	On a file: you may change its contents. On a directory: you may create, rename or delete files inside it.
`x`	On a file: you may run it as a program. On a directory it means something different: you may enter it and reach the things inside. Without x on a directory, even `cd` into it fails. This double meaning surprises everyone once.
`4 2 1`	How the numbers work: `r` counts 4, `w` counts 2, `x` counts 1. Add them up per class — `rw-` is 4+2=6, `r-x` is 4+1=5. Three digits = owner, group, others.
`644`	Owner: read+write (6). Group and others: read only (4). The normal setting for an ordinary file.

`600`	Owner: read+write. Everyone else: nothing (0). Use for anything secret — keys, tokens, `.env` files.
`755`	Owner: read+write+execute (7). Everyone else: read+execute (5). The normal setting for scripts and for directories.
`700`	Owner: everything. Everyone else: nothing. A fully private script or directory.
`chmod 644 f`	Numeric form: replaces all three classes at once with exactly the permissions the number spells out.
`chmod u+x f`	Symbolic form: changes only what you name. This adds (`+`) execute (`x`) for the owner (`u` for user) and leaves group and others untouched.
`chmod go-w f`	Removes (`-`) write permission from group (`g`) and others (`o`).
`chmod a+r f`	Grants read to all three classes at once — `a` means "all".
`chmod -R 755 .`	`-R` means recursive: apply to this directory and everything below it. Powerful, and usually more than you meant — prefer naming the one file.

- Only one of the three classes applies to you, and it is picked by first match wins: if you own the file you are checked against the owner letters; otherwise, if you are in its group, the group letters; otherwise the "others" letters. There is no falling through to the next class. Strange consequence: own a file with mode `044` and you cannot read it, even though group and others can — you matched "owner", and owner says no.
- When a path is denied, the system checks your right to pass through each directory (the x bit) before it even looks up the name you asked for. So `ls /root/secret` fails with "Permission denied" on `/root` without revealing whether `secret` exists at all. The error refuses to leak information.
- A directory with `r--` (read but no execute) lets you list the names inside it and nothing more: `ls dir` works, but `cd dir` and `cat dir/file` both fail, because using anything inside requires passing through — the x bit.
- Deleting a file needs write permission on its directory, not on the file. A file name is an entry in the directory, and deleting edits the directory. So you can delete a file you cannot read — and fail to delete a file you own, if the directory refuses.
- Only the owner of a file may `chmod` it. Having read or write access is not enough, and this sandbox has no `sudo` to override that — a root-owned file simply stays as it is.
- `644` and `755` are conventions, not rules. They fit ordinary files and scripts; for anything secret, reach for `600` or `700` instead.

`chmod -R 777 .`

The classic panic move when a permission error appears — and almost always wrong. It gives every account on the machine full read, write and execute on everything below you; it does not fix files owned by root (you cannot chmod those anyway); and it leaves the wide-open permissions behind long after the original problem is forgotten. Instead: read the error, find the one path and the one missing letter, and change only that.

EVERY COMMAND // THIS SANDBOX

What you can type

Every command this sandbox understands, generated from its own command table — so this list is always exactly what works in the terminal.

Not supported: pipes (`|`), `sudo`, and redirecting into a program; trying one prints an honest error instead of pretending.

`cat FILE...`	**print file contents** `-n number` Print the whole content of a file.
`cd [DIRECTORY]`	**change the working directory** Change directory. `cd ..` goes one level up, `cd ~` goes home, `cd -` returns to the previous directory.
`chmod [-R] MODE PATH...`	**change file permissions** `-R recursive` Change permissions, numerically (`chmod 644 file`) or symbolically (`chmod u+x file`). You must own the file.
`clear`	**clear the scrollback** Clear the scrollback. History is kept.
`cp [-r] SOURCE DEST`	**copy files and directories** `-r recursive · -R recursive-upper` Copy a file. `-r` copies a directory and everything inside it.
`echo [-n] TEXT...`	**print arguments** `-n no-newline · -e escapes` Print its arguments. With `>` or `>>` it writes them into a file instead.
`exit`	**end the session** End the session. Nothing is lost — you can restart it.
`file PATH...`	**identify what a path is** Guess what kind of thing a path is: regular file, directory, symbolic link, or device.
`head [-n COUNT] FILE...`	**print the first lines of a file** `-n N lines` Print the first lines of a file (10 by default, `-n` to change).
`help [COMMAND]`	**list available commands** List every command this sandbox understands.
`history`	**list commands typed this session** List the commands typed this session.
`lab reset|hint|next|back|goto CHAPTER|video|sheet [SECTION]`	**sandbox controls (not a Linux command)** Sandbox controls: `lab reset` rebuilds the filesystem, `lab hint` shows the current step's answer.
`ls [-a|-A] [-l] [-d] [-R] [-h] [-1] [FILE...]`	**list directory contents** `-1 one-per-line · -a all · -A almost-all · -l long · -d directory · -R recursive · -h human-readable` List directory contents. `-a` shows dot-prefixed names, `-l` shows the long form with owner and permissions.
`man COMMAND`	**show a short description of a command** Show a short description of a command. These are sandbox blurbs, not the real manual pages.
`mkdir [-p] DIRECTORY...`	**create a directory** `-p parents` Create a directory. `-p` creates missing parents and does not complain if it already exists.
`mv SOURCE DEST`	**move or rename** Move or rename. Moving within the same filesystem changes a name, not the data.

`pwd`	**print the working directory** Print the working directory: the absolute path of where you are standing in the tree.
`rm [-r] [-f] PATH...`	**remove files and directories** `-r recursive · -R recursive-upper · -f force · -d dir` Remove files. `-r` recurses into directories, `-f` stays quiet about things that are not there.
`stat PATH...`	**show full metadata for a path** Show the full metadata of a path: type, size, permissions in both forms, owner, group, modification time.
`su [USER]`	**switch user (not available here)**
`sudo COMMAND`	**run a command as another user (not available here)** Run a command as another user. Not available here on purpose: the boundary is the lesson.
`tail [-n COUNT] FILE...`	**print the last lines of a file** `-n N lines` Print the last lines of a file (10 by default, `-n` to change).
`touch FILE...`	**create a file, or update its timestamp** Create an empty file, or update the modification time of an existing one.
`tree [-a] [-L DEPTH] [DIRECTORY]`	**draw the directory structure** `-a all · -L N level` Draw the directory structure as a tree. Depth is capped in this sandbox.
`whoami`	**print the current user** Print the current username.

KNOW THE PATH // CONTROL THE AGENT

Scope the access, not the ambition

Giving a coding agent access to your project does not mean giving it your whole machine. The question is always the same one permissions ask: which paths, and read or write?

SAFE BY DEFAULT // READ + WRITE

`~/projects/...`	Your project folder. The right workspace for an agent: it only touches this one project, and every edit shows up in a git diff you can review.
`.git/`	The version history inside each project. This is what makes agent edits safe — any change can be reviewed, and reverted with one command. Without version control there is no undo.
`/tmp`	Temporary scratch space. Fine for an agent to write test output and intermediate files here, because nothing in it is meant to be kept.

REVIEW CAREFULLY // READ, THEN REDACT

`~/.claude/`	Claude Code's own settings: `CLAUDE.md`, `settings.json`, `skills/`. Be careful letting an agent edit these — a change here alters how the agent behaves in every project, not just the current one.
`~/.codex/`	The same, for Codex: `config.toml` and `AGENTS.md`. An edit here also reaches beyond the current project.

`~/.agents/skills/`	Skills shared by several AI tools. One edit changes the behaviour of every tool that reads this folder.
`~/.env`	Environment files hold real secrets: API keys, database passwords. If an agent needs to know what variables exist, give it `.env.example` — the same variable names with the values left blank.
`/var/log`	Logs are the best debugging source, but they often contain leaked secrets, addresses and user data. Let an agent read them, then have it quote only the relevant, redacted lines — never paste whole logs into a prompt.

HIGH RISK // LOOK, DO NOT TOUCH

`/etc`	System-wide configuration. Reading it to diagnose a problem is fine; writing to it changes the machine for every user and every service.
`/dev`	Device files. Writing to one sends data straight to hardware — the wrong write to a disk device destroys the disk's contents.
`/proc`	Live kernel and process state. Mostly read-only information, but a few entries accept writes that reconfigure the running system.
`/sys`	Hardware and driver settings. Some files here change device behaviour the moment they are written.
`~/.ssh`	Your SSH private keys — full access to every server they unlock. No agent task requires writing here, and key contents must never end up inside a prompt.

- Before granting an agent elevated (admin/root) privileges, make it answer three questions: what exact file needs to change? Why does that change require elevation? Is there a project-level alternative that does not?
- If the agent cannot name the exact file it needs to change, do not grant elevation yet. A vague request for admin rights is a request for the whole machine.
- One permission error means the agent hit one boundary. That is normal and informative — it is not a reason to run the agent as administrator.
- Prefer changes you can see and undo: an edit inside the project shows up in a diff and reverts with git. A system-level change does neither.
- Read-only access solves most of it. What an agent usually needs from `/etc`, `/proc` or `/var/log` is information for a diagnosis — it can read without being allowed to write.