CYPHERPUNK SCHOOL — SOVEREIGNTY SERIES

The Command Line, Part 2

Linux CLI 102 — Streams, Pipes & the - Convention

Once you see the command line as plumbing — three hoses, a pipe, and a placeholder for “the stream” — one-liners stop looking like magic incantations

A practical guide to digital sovereignty

Cypherpunk School

cypherpunkschool.com

Sovereignty Series • Foundation: Builds on CLI 101 • June 2026

Contents

1. Once You See Streams, It's Just Plumbing

If *The Command Line, From Zero* got you moving around the system, this short follow-up hands you the idea that makes everything click: **streams**. Once you understand them, the whole command line stops feeling like magic incantations and starts feeling like plumbing — hoses, valves, and a pipe between them. This is the lesson that unlocks redirection, pipes, and one-liners like decrypting a single file out of an encrypted archive without ever writing the rest to disk.

> **What you'll walk away with.** Three streams (`stdin`, `stdout`, `stderr`), the redirection arrows (`> <`), the pipe (`|`), and one quietly powerful convention — the lone `-` that means "use the stream, not a file." That's roughly 90% of command-line power, and it fits on one page.

2. The Three Streams

Every command has three standard "hoses" attached to it:

Stream	#	What it is / analogy
`stdin`	0	where input comes **in** — the drain (it drinks)
`stdout`	1	where normal output goes **out** — the faucet (it pours)
`stderr`	2	where errors go out *separately* — the overflow spout

Errors travel on a *separate* stream so you can capture a command's real results without the error noise mixing in. That separation is the whole reason the next few tricks work cleanly.

3. Redirection — Point a Stream at a File

Redirection aims a stream at a file instead of the screen:

```
command > file      # stdout -> file (overwrite)
command >> file     # stdout -> file (append)
command < file      # file -> stdin (feed input from a file)
command 2> errs.log # stderr -> file
command &> all.log  # stdout AND stderr -> file
```

Read the arrow as "flows into." `>` overwrites; `>>` adds to the end. The `2` in `2>` is just the stream number from the table above — you're saying "send stream 2 (errors) over here."

4. The Pipe | — Connect Two Commands

A pipe wires **the left command's `stdout` into the right command's `stdin`**. Always. Think faucet → drain: the left command's output pours into the pipe, and the right command drinks it through its input.

```
   left command            right command
   produces output         consumes input
   [ stdout ] ====|====> [ stdin ]
       faucet ----------> drain
```

```
cat log.txt | grep ERROR | wc -l    # file -> filter -> count
```

Left writes `stdout`, right reads `stdin`. Every time. You can chain as many as you like — each `|` hands the previous command's output to the next.

> **One catch worth remembering.** `stderr` does *not* go through the pipe — only `stdout` does. So errors still print to your screen even mid-pipeline, which is usually what you want: the results flow on, the warnings stay visible.

5. The `-` Convention — "Use the Standard Stream Here"

Many tools accept a lone `-` wherever they'd normally expect a **filename**. It means *"don't use a file — use the standard stream."* Whether that's `stdin` or `stdout` depends on whether the tool is *reading* or *writing* at that spot:

```
tar xzf -        #  -  = read the archive from STDIN
gpg -o -         #  -  = write the output to STDOUT
curl -o - URL    #  -  = write the download to STDOUT
```

It's context-sensitive on purpose: `-` is just a placeholder for "the pipe / standard stream instead of a named file." Once you see `-` and read it as "the stream," a whole class of one-liners opens up.

6. Capstone — Recover ONE File From an Encrypted Archive

Here's the payoff. You have an encrypted archive and you want a *single* file out of it — without decrypting the whole thing onto disk:

```
gpg -d secrets.tar.gz.gpg | tar xzf - ./dot-env
```

Read it left to right:

1. `gpg -d secrets.tar.gz.gpg` — decrypt; write the plaintext `.tar.gz` bytes to **stdout**.
2. `|` — pipe that `stdout` into tar's `stdin`.

3. `tar x z f -` — e**x**tract, gun**z**ip, archive **f**rom `-` (`stdin`).

4. `./dot-env` — extract *only* this member; ignore everything else in the archive.

> **Nothing but `dot-env` ever touches the disk.** The rest of the plaintext just streams through memory and is discarded. That's selective recovery *and* minimal exposure — no temp files, no decrypted copy left lying around.

Want the file printed to the terminal instead of written to disk? Add a capital `O` ("to stdout"):

```
gpg -d secrets.tar.gz.gpg | tar xzOf - ./dot-env   # contents -> terminal
```

Two different streams, same command: `f -` means "archive comes *from* stdin," and `O` means "send the extracted file's contents *to* stdout."

7. Practice (Safe, Throwaway)

Everything here is reversible and self-cleaning — it builds a throwaway folder and deletes it at the end. `gpg -c` will prompt you for a passphrase; pick anything, you're wiping it in a moment.

```
mkdir cli-reps && cd cli-reps
printf 'token=abc123\n' > .env ; printf 'pubkey\n' > id.pub
tar czf box.tar.gz .env id.pub          # bundle two files
gpg -c box.tar.gz                        # passphrase-encrypt -> box.tar.gz.
   gpg
rm box.tar.gz .env id.pub                # wipe the originals

gpg -d box.tar.gz.gpg | tar tzf -        # LIST contents (t = list)
gpg -d box.tar.gz.gpg | tar xzf - .env   # extract ONLY .env to disk
gpg -d box.tar.gz.gpg | tar xzOf - .env  # print .env to the terminal
cd .. && rm -rf cli-reps                 # clean up
```

Run it line by line and watch what each one does. The three `gpg -d | tar` variants are the same idea with one letter changed: `t` lists, `x` extracts to disk, `xO` prints to the screen.

> **That's the whole model.** `stdin` / `stdout` / `stderr` are the three hoses; `|` joins them; `>` and `<` point them at files; `-` says "use the hose, not a file." Carry that mental picture and the dense one-liners you'll meet in the rest of the series read like plain plumbing.

Continue the track

Part 3 — A Faster Terminal: modern tools (ripgrep, bat, fzf, and friends) that make the command line genuinely pleasant. Read it at

cypherpunkschool.com/guides/cli-103

New here? Start with **The Command Line, From Zero** (CLI 101). For more on self-hosting, privacy, local AI, and digital sovereignty:

cypherpunkschool.com