MSYS2 & the two-homes problem

Key takeaways

1. On Windows, Unix tools see two homes: POSIX $HOME and native $USERPROFILE.
2. Eliminating the two-homes split beats managing it when the platform allows.
3. A loud missing config beats a tool silently reading the wrong home.

Windows paths only look Unix-like

No Windows experience assumed

- Windows path conventions look superficially like Unix but differ in load-bearing ways. Getting them wrong gives you a config that ‘works’ but with the wrong settings.
- When you run Unix-style command-line tools on Windows, TWO home directories exist at once. Different tools read different ones.
- This deck builds the picture from scratch — what POSIX is, what MSYS2 is, why there are two homes — then walks through real bugs the split caused and their fixes.

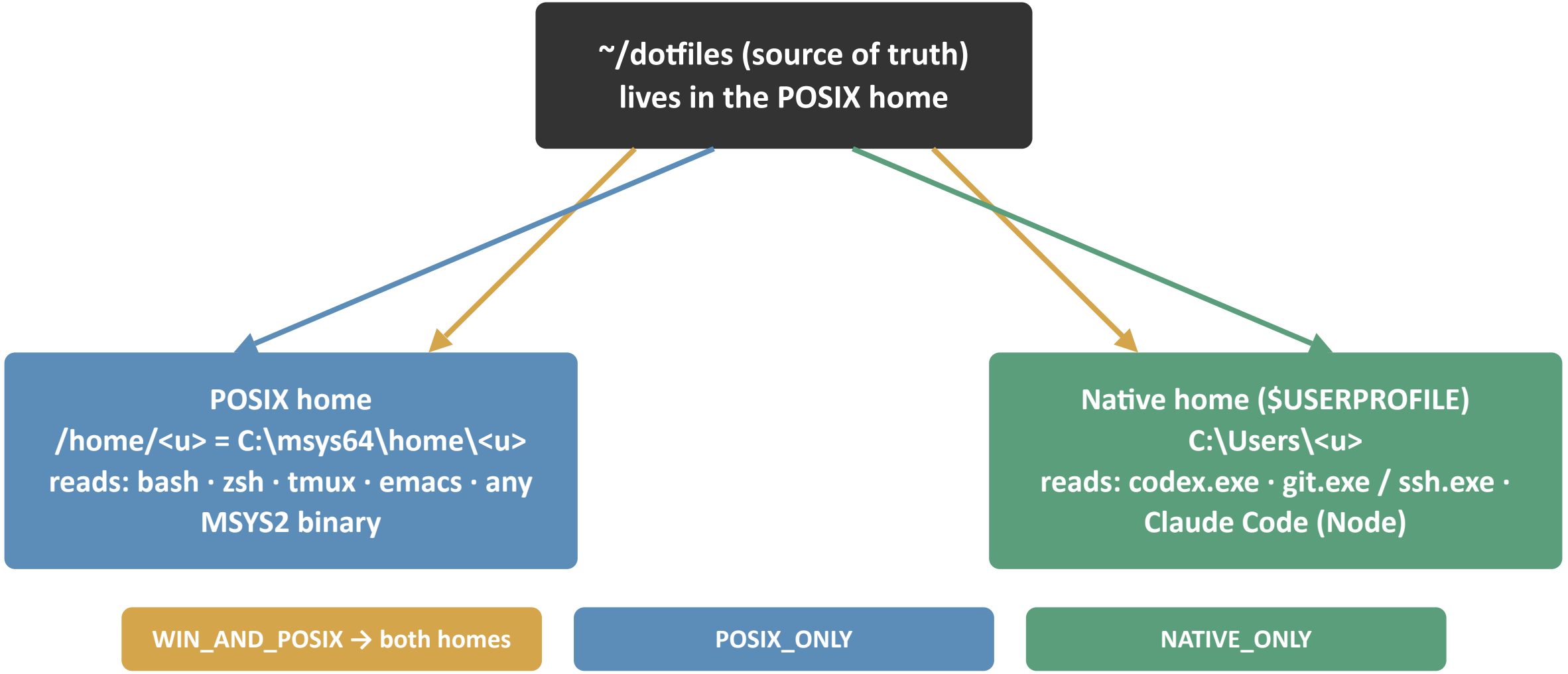
Jargon, defined once

Terms used throughout this deck

- Operating-system families
  - POSIX: Unix-like systems (Linux, macOS, BSD). Forward slashes, /home/<u>, $HOME.
  - Windows native: backslashes, drive letters (C:\Users\you), $USERPROFILE for home.
- Running Unix tools on Windows
  - POSIX-on-Windows runtime: a layer giving Unix tools bash, /home/<u>/, /usr/bin on Windows.
  - MSYS2 / Cygwin / Git Bash: three such runtimes; a user can have all three at once.
- Path translation
  - $HOME: your home dir — /home/<u> (Linux), /Users/<u> (macOS), C:\Users\<u> (Windows).
  - cygpath: translates between Unix (/c/Users/you) and Windows (C:\Users\you) path forms.

Two homes, one source of truth

./links routes each config into the home whose tools read it



Why Windows is harder than ‘Linux with .exe’

Three runtimes, two homes, three names for one file

- Three POSIX-on-Windows runtimes that look alike
  - Cygwin — under C:\cygwin64\, home /home/<u>. Oldest; full POSIX emulation.
  - MSYS2 — under C:\msys64\, home /home/<u>. Cygwin descendant + MinGW-w64 compilers.
  - Git Bash — ships with Git for Windows; uses $USERPROFILE as $HOME, no /home/<u>.
  - All three answer to `bash` — a path can mean one thing in one, nothing in another.
- One file, multiple legal names
  - Same bytes, three string forms: C:\…, /c/…, /home/<u>/…
  - Naive string compare calls two names for one file ‘different’.
  - cygpath translates between forms — a canonical lift before comparing.

Classify every config by who reads it

./links puts each tracked file only in the homes whose tools read it

Group	Linked into	Examples
WIN_AND_POSIX	both homes	.gitconfig (native git.exe + POSIX git), Claude Code state, ccstatusline
NATIVE_ONLY	native home only	codex/AGENTS.md — codex.exe never reads the POSIX home
POSIX_ONLY	POSIX home only	.bashrc, .shellcommon, .tmux.conf, .emacs.el, .ssh/config, .gdbinit

Reader-based classification rule

Classify by consumers, not by source

- Classify by who READS the destination, not who wrote the source — the source always lives in ~/dotfiles.
- A missing POSIX_ONLY dotfile fails loudly when a shell starts in the wrong home; a silently wrong config is harder to notice.
- On macOS/Linux there is one $HOME, so the split is a no-op. Since the 2026-07 single-home migration, Windows can be boring too.

Three rules for paths INSIDE tracked configs

Not just where the file lives — what it contains

- No ~/… paths that name user working trees
  - ~ expands to whatever the running program calls $HOME — different per runtime.
  - ~/git-papers in a config codex.exe reads → C:\Users\witchel\git-papers, the wrong tree.
  - Fix: write an OS-native absolute path, or substitute at ./links time.
- No absolute paths from another host
  - Cross-machine commits smuggle in /Users/<u>/… or /home/<u>/… — dead weight elsewhere.
  - Strip on review; agents are especially prone to baking in the current-machine prefix.
- Tool-written state is tolerated, not engineered around
  - Tools rewrite their own config (codex trust entries, ssh known_hosts) — the churn lands in git.
  - `git diff` deliberately, keep or revert per intent; templating only if churn is constant.

cygpath -w — picking one canonical form

The one primitive that makes path comparison reliable

- The problem: three legal names for the same file
  - C:\msys64\home\witchel\dotfiles\.bashrc — the Windows form.
  - /c/msys64/home/witchel/dotfiles/.bashrc — MSYS2's /c/ view of drive C:.
  - /home/witchel/dotfiles/.bashrc — the mount-alias form. Same bytes; three strings.
- The fix: normalize to Windows form before comparing
  - `cygpath -w <path>` returns the Windows form — stable across mounts and aliases.
  - Lift both sides at every comparison; pass Windows-form paths across every boundary.
- Companion: cygpath -up for translating PATH
  - A Windows-style PATH in bash breaks: drive-letter colons (C:) fragment every entry.
  - `cygpath -up` converts the whole list (semicolons, C:\…) to POSIX form (colons, /c/…).

Setup — PATH, and how it gets mangled

Background for the bug on the next slide

- PATH, in plain English
  - An environment variable: the directories the shell searches when you type a command.
  - Break it and common commands look missing — even though they are installed.
- PATH is spelled differently on each side
  - POSIX: colon-separated, forward slashes — /usr/bin:/usr/local/bin:/home/u/bin.
  - Windows: semicolon-separated, drive letters — C:\Windows\System32;C:\Program Files\Git\bin.
- Why this bites when Claude Code launches bash
  - Claude Code is a native-Windows Node program — its own PATH is Windows-style.
  - The child bash inherits that PATH and splits on colons only.
  - Drive-letter colons (C:) fragment every entry — bash can’t find git or anything else.

Story — PATH bootstrap from a hostile env

How it manifested

Claude Code's Bash tool flaked on `git status`, `tr`, `dirname` — ~10 patches to `bashenv.sh` (the script that prepares bash's environment) tried to derive the right PATH from scratch. None held.

What was actually happening

Claude handed bash a PATH that mixed Windows-style entries (semicolon-separated, with `C:\…` and drive letters) and POSIX-style entries (colon-separated). Bash splits on colons only — so every drive-letter colon fragmented an entry.

The fix — one conditional

```
case "$PATH" in *\;*)
[ -x /usr/bin/cygpath ] && \
PATH=$(/usr/bin/cygpath
-up "$PATH");;
esac
```

- Reading the fix: any semicolon in PATH → translate the whole list through cygpath -up. It's called by absolute path because the broken PATH can't find it.
- Diagnostic that exposed it: dump PATH on entry to `bash -c`. The mix was obvious — every prior iteration had touched derivation logic, not the input.
- Principle: don't trust the environment you inherit. A script on a cross-platform boundary normalizes its inputs at the top, before downstream code runs.

source: dotfiles/HISTORY.md @ 2026-05-13; commits 881e11d, a1ceb64, f1bbcfd

Setup — the env that doesn’t survive a nested shell

A second way the Bash tool’s environment bites — subtler than PATH

- Two Win32 vars native Windows tools can’t run without
  - TEMP / TMP: where a native tool writes scratch files — uv unpacks its installer here.
  - USERPROFILE: the Windows home dir. Native-Windows Python’s expanduser reads it, never POSIX $HOME.
- The Bash tool HANDED bash these vars unexported
  - It ran Git for Windows’ bash — a second msys runtime that keeps TMP/TEMP unexported.
  - A native tool spawned DIRECTLY still got them — MSYS2 rebuilds the Win32 env for a native child.
  - But only EXPORTED vars cross into a child bash; unexported ones stop at the first shell.
- So the trap springs only one shell deeper
  - Type `uv …` at the prompt: native child of the top shell → env reconstructed → works.
  - Run a SCRIPT (./check, ./tools, a shebang, `bash -c`): a bash spawned by bash — its grandchildren lose all three.
  - Not the two-homes problem: not WHICH home, but whether the env survives the trip at all.

Story — the env that vanished one shell down

How it manifested

./check and ./tools failed under Claude Code's Bash tool but ran fine when typed interactively. Pyright: "Could not determine home directory." uv's installer: EACCES on C:\Windows\.tmp*.

What was actually happening

The Bash tool left USERPROFILE / TEMP / TMP unexported. ./check is a bash spawned BY that bash — it inherited none. So its native grandchildren got no writable Win32 temp (uv fell back to C:\Windows) and no home (Python reads USERPROFILE).

The fix — re-export at the top

```
h=$(cygpath -w "$HOME")
t=$(cygpath -w
"${TMPDIR:-/tmp}")
export USERPROFILE="$h" \
TEMP="$t" TMP="$t"
```

- The fix — one helper atop check / tools / gemini-ask: cygpath -w lifts $HOME and $TMPDIR to Windows form, then EXPORTs USERPROFILE / TEMP / TMP. No-op off MSYS2.
- Why it hid: depth-dependent. One bash deep, uv is a direct child and MSYS2 rebuilds its env; a script (bash-in-bash) drops it — worked when typed, failed when scripted.
- Same arc as two-homes — first manage the split, then eliminate it. 2026-08: CLAUDE_CODE_GIT_BASH_PATH pins the Bash tool to MSYS2's own bash and bashenv.sh re-exports TMP/TEMP — one runtime end to end. The helper stays for shells outside Claude.

source: dotfiles/HISTORY.md @ 2026-07-11; commits 69693bf, ee9e3a1, ccd8a94; fix 80891b7

Setup — which runtime owns the dotfiles?

Background for the second story

- The situation
  - A user can install both Cygwin AND MSYS2 — each with its own /home/<u>/ and repo clone.
  - Run `./links` from either clone and the script must know which runtime to set up for.
- The naive approach: detect the OS at runtime
  - Ask $OSTYPE / `uname` — but both runtimes coexist, so detection must pick one winner.
  - Inside Cygwin's bash but the script picks MSYS2 → your Cygwin clone installs MSYS2 links.
- The better signal: where is the clone installed?
  - Each clone knows one thing for sure: its own path on disk.
  - C:\cygwin64\…\dotfiles → Cygwin-side; C:\msys64\…\dotfiles → MSYS2-side. Unambiguous.

Story — route by clone, not by runtime

The wrong design

Detect with $OSTYPE or uname. With two runtimes, detection picks a winner and the loser writes to the wrong tree.

Never let uname pick which tree you write.

The right design

Branch on the clone's install path: /c/cygwin64/* is the Cygwin side, /c/msys64/* the MSYS2 side.

Run ./tools once per clone, on each side.

The principle

When both are valid, route on the deterministic signal: where am I installed, not what the env claims.

Install both runtimes before trusting it.

When you add a second instance, check whether the original assumed itself unique: self-identification scales to N, detection only picks a winner.

source: dotfiles/HISTORY.md @ 2026-04-30 (MSYS2 as peer to Cygwin)

Setup — the split caught shells, not tools

Where the loud-failure safety net has a hole

- Recall the safety net (earlier slides)
  - POSIX_ONLY configs are deliberately kept OUT of the native home.
  - A shell starting with HOME=$USERPROFILE finds no .bashrc and fails LOUDLY — you notice at once.
- What it protects: misrouted SHELLS
  - A shell READS existing dotfiles; a missing file is an immediate, visible error.
  - So a shell in the wrong home cannot run silently on stale config.
- What it misses: tools that CREATE paths on demand
  - A native tool resolving ~ to WRITE (a hook, a cache, a settings file) never hits a missing-file guard.
  - It just makes the directory in the native home — a second, shadow copy — and writes there forever.
  - No missing file, no error, no signal. The guard was shaped for reads; writes slip past it.

Story — the shadow dotfiles tree

How it manifested

ccstatusline rendered its BUILT-IN default bar on Windows for over a month — the customized hostname / nord-aurora config looked right on macOS/Linux but had never been read here. Separately: every Claude breadcrumb ever written was missing from the repo.

What was actually happening

./links wrote the symlink into the POSIX home ($HOME = /c/msys64/home/witchel). But ccstatusline runs via native node.exe, whose os.homedir() returns $USERPROFILE = C:\Users\witchel. Two homes, never met — the native side held a stale defaults file.

The pattern — and the real fix

Every native tool resolving ~ minted a shadow C:\Users\witchel\dotfiles and wrote there. The guard caught misrouted shells; it never fired for path-CREATING tools. The fix isn't a better guard — it's one home.

- Why it was invisible: /c/msys64/home/witchel and C:\Users\witchel 'look interchangeable' — same user, both plausibly 'home'. Nothing ever compared them.
- The native home had quietly become the center of gravity anyway: uv tools, claude.exe, codex state, cargo / rustup all already lived there.
- Principle: a safety net shaped for one failure mode (reads) is silent on its mirror image (writes). When a whole bug class traces to two things that should be one, unify them — don't bolt on a second guard.

source: dotfiles/HISTORY.md @ 2026-07-07 (single-home migration) + 2026-06 (ccstatusline)

The fix — one home, like macOS/Linux

Relocate db_home so MSYS2 stops carving out its own /home

- The move (2026-07)
  - db_home: /c/Users/%U in MSYS2's /etc/nsswitch.conf — MSYS2's $HOME now equals $USERPROFILE.
  - The dotfiles checkout and the repo trees (git/, git-papers/, git-class/) moved under $USERPROFILE.
- The result: no divergence left to route around
  - $HOME, $USERPROFILE, and ~ all resolve to /c/Users/<u> for EVERY runtime — native and POSIX alike.
  - WIN_AND_POSIX / NATIVE_ONLY / POSIX_ONLY collapse to one home: a no-op, exactly like macOS/Linux always were.
  - The HOME-rewrite in .bash_profile / .zshrc and its compensating re-cd are deleted — dead weight once there is one home.

The gotcha — db_home: windows was empty

The keyword did not mean what the migration needed

- What failed
  - `db_home: windows` reads the SAM home-folder attribute.
  - For local accounts that attribute is blank, and the profile-dir fallback did not fire.
- The robust form
  - Use the explicit `/c/Users/%U` template.
  - The three ./links groups survive only to service a hypothetical second or legacy home.

Evolution — from two homes to one

Apr–May: build the machinery to manage the split · Jul: eliminate it

Apr 30

MSYS2 joins as a peer; route-by-clone replaces hardcoded /c/cygwin64 paths

May 1–2

Three-script split; LINKS groups (WIN_AND_POSIX / …); cygpath -w canonical lift

May 10–15

Hardening: codex → NATIVE_ONLY, PATH bootstrap, winpty TUI bridge

Jul 7

Single-home migration: db_home /c/Users/%U — two homes collapse to one

Setup — symlinks on Windows vs Unix

Background for the bug on the next slide

- A symlink, in plain English
  - A symlink is a file that points to another file (the ‘target’).
  - Reading or running the symlink reads or runs the target.
  - Example: ~/.bashrc points to ~/dotfiles/.bashrc — edit either name, same bytes.
- Why creating one on Windows is harder than on Unix
  - On macOS/Linux, any user can make one with `ln -s target link` — no permissions needed.
  - On Windows it needs admin rights: the script asks Windows to elevate (the UAC popup) first.
  - Elevation runs in a separate, short-lived window of PowerShell — Microsoft’s native shell — and its output is gone once it closes.
- The relative-vs-absolute path trap (this is what the bug hit)
  - ‘Create a symlink at A pointing to B’ — Windows has to figure out what B means.
  - If B is relative (‘dotfiles/.bashrc’), Windows resolves it against PowerShell’s current dir — NOT the script’s.
  - If that resolved path doesn’t exist, Windows still creates the link — pointing at nothing.

Story — the silent symlink bug

What looked right

The agent changed Windows symlinks to use relative target paths. The script’s output said ‘Done.’ Exit code 0. No errors.

What actually happened

Of 35 symlinks the script claimed to create, zero existed on disk. PowerShell resolved each relative target against its own current directory — the wrong directory — so every link was created pointing at a non-existent file. Windows then silently allowed it. The elevated window closed before anyone saw the errors.

The fix

Convert every path to an absolute Windows path with cygpath -w — the same canonical lift from earlier in this deck — before passing it to PowerShell. Then verify the artifact, not the log.

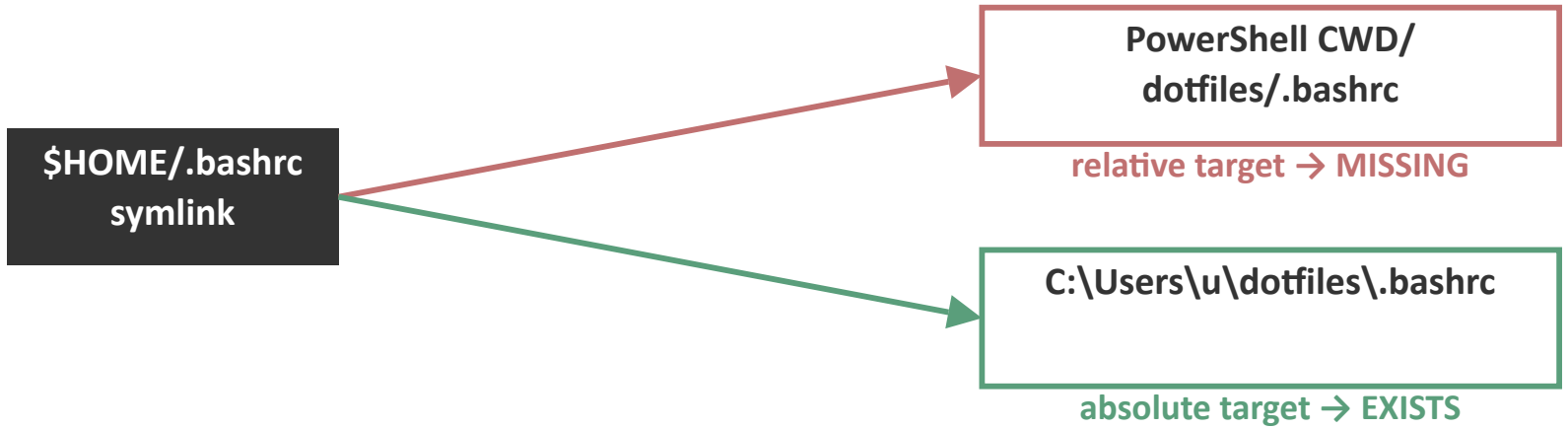
- The commit looked plausible and looked tested. The failure mode only manifested under UAC elevation, which the developer hadn’t reproduced before pushing.
- Two compounding causes: (1) PowerShell’s New-Item resolved the relative target against its own directory, not the script’s; (2) -ErrorAction Continue made it swallow errors instead of stopping.

source: dotfiles/HISTORY.md @ 2026-05-02; commit 1f79587 then its revert

Lesson — the silent symlink bug

1. Exit code 0 only means the script didn't crash — not that the thing it was supposed to do is true on disk. Verify the artifact before declaring success.
2. When a tool runs in a window that closes (elevated PowerShell), its output is gone: halt on first error, or log to a file.
3. Agents are good at making scripts that LOOK right; a plausible diff with a green exit code is not a working change. One verification step beats finding 35 broken symlinks three days later.

The link can exist even when its resolved target does not.



Setup — symlinks are privileged on Windows

The mechanism every ./links slide relied on is itself a trap

- Recall what a symlink is — and why this deck cares
  - The entry POINTS at another path: edit ~/.bashrc, you edit the repo file.
  - That identity is ./links' whole contract. A COPY breaks it silently — edits diverge from day one.
- Windows historically requires elevation to create one
  - Default: only an admin process may create symlinks — which is why ./links batches all its links into ONE elevated PowerShell run (a single UAC prompt).
  - Developer Mode (Settings > For developers) lifts that — but only for tools that pass the new opt-in flag. cmd's mklink does; PowerShell 5.1's New-Item never does.
- MSYS2's default ln -s does not even try
  - With MSYS unset, ln -s COPIES the target and returns 0 — no error, no link.
  - MSYS=winsymlinks:nativestrict demands a real symlink: create one or fail loudly.

Story — re-testing an ‘impossible’

The recorded ‘fact’
The repo’s own landmine note said unprivileged symlinks were impossible: ‘every MSYS=winsymlinks mode copies, nativestrict returns 0 rather than failing; PowerShell demands elevation.’ Written down after real testing — and wrong.

What re-testing showed
With Developer Mode on, `env MSYS=winsymlinks:nativestrict ln -s` makes a REAL symlink — while the prefix form `MSYS=… ln -s` still silently copies, exit 0, every time. The ‘fact’ was invocation shape plus machine state, not the platform.

The fix — set it at birth
./links now sets MSYS=winsymlinks:nativestrict at Windows USER scope: every fresh shell tree is born with it, so ln -s links for real or fails loudly. Older trees keep copying until relaunched.

- The matrix now: tree born with the var → real symlink at any depth; bare or prefix ln -s in an old tree → silent copy; PowerShell 5.1 New-Item → elevation always, so ./links keeps its one UAC batch.
- Why the wrong absolute survived: exit 0 plus a file that LOOKS right — it passes every casual check except test -L. The shadow tree’s silent-wrong-behavior shape, again.
- Principle: a trap note describes an invocation on a machine state, not the platform forever. When the state changes (Developer Mode), re-verify — and record the invocation that works beside the one that fails.

source: dotfiles/HISTORY.md @ 2026-08-07; commits d28bfb9, 16d1f31

Key Takeaways

1. When Unix tools run on Windows, two homes can coexist: POSIX $HOME and native $USERPROFILE.
2. You can manage the split by classifying configs by their readers, but eliminating the split is better when the platform lets you.
3. A loud missing config is easier to debug than a tool silently reading or writing the wrong home.

Takeaways that generalize beyond Windows

1. A Windows path is an equivalence class of names for the same file, not just one string.
2. Normalize PATH, HOME, and path form at the top of a cross-platform script, before downstream code reads them.
3. When you add a second instance of something the original assumed was unique, route by deterministic identity rather than runtime guesses.