

way of the hax0r
n00b to console punk

Chapter 13

Cross the OS Boundary

Almost nobody works on one operating system. A Windows laptop deploys to a Linux server, a Mac shares a project with a Windows teammate, and a class hands out a script that has to run on all three. Each boundary crossing has the same handful of traps: paths that only one side understands, files that look the same and are not, archives that unpack differently, and commands that share a name but not their options. This chapter collects those traps and the tools that handle them.

The concepts in this chapter are:

- **WSL**, a real Linux inside Windows, and how it sits next to the native shells;
- **path conversion** between `C:\Users\student` and `/mnt/c/Users/student`;
- sharing files between environments without corrupting them;
- **encodings** and **line endings** at the boundary, and how Git treats them;
- **archives**: tar, zip, and PowerShell's Compress-Archive;
- **checksums**, which prove that a file arrived unchanged;
- **BSD versus GNU** command-line tools, the difference between macOS and Linux.

The examples create archives and checksums in the practice directory. Nothing needs administrator rights; installing WSL does, and its instructions say so.

WSL: Linux Inside Windows

The **Windows Subsystem for Linux** runs a Linux distribution inside Windows with its own filesystem, package manager, and shell. It is an addition to Windows, not a replacement for PowerShell or Command Prompt, and it is the easiest way to follow the UNIX track of this book on a Windows machine while keeping the Windows track available in the next tab. [1]

Install it from an elevated PowerShell, then reboot:

```
$ wsl --install
```

That installs the current WSL version and Ubuntu by default. `wsl --list --online` shows other distributions, and `wsl --install -d Debian` picks one. Afterward, Windows Terminal offers a profile for the distribution, and `wsl` from any Windows shell opens its default shell. [2]

Three commands are enough to move between the two worlds:

```
$ wsl --list --verbose
$ wsl
$ wsl ls -la
```

The first shows installed distributions and whether they are running. The second opens a Linux shell in the current directory, translated to its Linux path. The third runs one Linux command from PowerShell or Command Prompt and returns, so that `wsl grep -r pattern .` gives a Windows shell the UNIX tools. [3]

From inside WSL, the reverse also works: `explorer.exe .` opens the current Linux directory in File Explorer, `code .` opens it in VS Code with its WSL extension, and `notepad.exe file.txt` runs a Windows program on a Linux file. [4]

Two Filesystems, Two Kinds of Path

WSL keeps Linux files in a Linux filesystem and shows the Windows drives under `/mnt`:

Where you are	Your Windows home	Your Linux home
In Windows	<code>C:\Users\student</code>	<code>\\wsl\$\Ubuntu\home\student</code>
In WSL	<code>/mnt/c/Users/student</code>	<code>/home/student</code>

`wslpath` converts between the two spellings:

```
$ wslpath 'C:\Users\student\Documents'
/mnt/c/Users/student/Documents
$ wslpath -w /home/student/project
\\wsl$\Ubuntu\home\student\project
```

Quote a Windows path in Bash, or the backslashes are escape characters, which Chapter 2 explained. Git Bash, the other UNIX-style environment on Windows, has the same idea with `cygpath` and mounts drives as `/c/Users/student` instead. [5], [6]

Where a file lives decides which tools are fast and which permissions apply. Linux tools working on `/mnt/c` cross a translation layer and run slower; Windows tools reaching into `\\wsl$` do the same in the other direction. Keep a project on the side that does most of the work with it: a Linux build in the Linux filesystem, a Windows application's files on `C:`. Permissions from Chapter 8 follow the filesystem too: files under `/mnt/c` show Windows permissions approximated as `rwx` for everyone, and `chmod` there does nothing unless the drive is mounted with metadata. [5]



Trap: Editing a file inside the Linux filesystem with a Windows program that does not know about WSL, by browsing to it through some other route than `\\wsl$`, can corrupt the Linux filesystem. Use the `\\wsl$` path or `explorer.exe .` from inside WSL, which route through the supported interface.

Line Endings and Encodings at the Boundary

Chapter 5 explained LF and CRLF endings and UTF-8 versus other encodings. The boundary is where they bite, because a file written on one side is read on the other.

- A shell script edited in a Windows editor and run in WSL fails with `$'\r': command not found` or `bad interpreter`. Convert with `tr -d '\r', dos2unix`, or the editor's line-ending switch.
- A file written by Windows PowerShell 5.1 with `>` is UTF-16, and `grep` in WSL finds nothing in it. Pass `-Encoding utf8`, or write it from PowerShell 7.
- A UTF-8 file with a BOM confuses some UNIX tools and a few Windows ones; save without a BOM unless a specific program asks for it.

Git can convert endings on the way in and out. The `core.autocrlf` setting and a `.gitattributes` file with `* text=auto` make the repository store LF and let each side check out what it prefers. Chapter 14 introduces Git; when you get there, set this once per project so that teammates on different systems do not commit every line of every file. [7]

Editors settle most of it. VS Code shows the encoding and line ending in its status bar, and an `.editorconfig` file in the project fixes both for everyone who opens it.

Archives

An **archive** packs a directory tree into one file so that it can be copied, downloaded, or attached as a unit. Two formats matter: **tar** on UNIX, usually compressed with `gzip` to `.tar.gz` or `.tgz`, and **zip** everywhere.

Bash/zsh:

```
$ tar -czf project.tar.gz project
$ tar -tzf project.tar.gz
project/
project/README.md
project/src/
project/src/main.c
$ tar -xzf project.tar.gz -C out
```

The options are create, list, or extract (`-c`, `-t`, `-x`), through `gzip` (`-z`), naming the file (`-f`). Always list an archive before extracting it: a tar file that unpacks a hundred files into the current directory instead of a subdirectory is a common annoyance. `-C` extracts into a directory you choose. [8]

```
$ zip -r project.zip project
$ unzip -l project.zip
$ unzip project.zip -d out
```

`zip -r` recurses, `unzip -l` lists, and `-d` picks the destination. [9]

PowerShell:

```
$ Compress-Archive -Path project -DestinationPath project.zip
$ Expand-Archive -Path project.zip -DestinationPath out
```

These cmdlets handle zip only. Windows 10 and later also ship `tar.exe`, which is BSD tar and understands both formats, so `tar -xzf project.tar.gz` works in PowerShell and Command Prompt too. [10], [11]

Archives cross the boundary with their contents' line endings and permissions. A zip made on Windows carries no execute bits, so a script extracted on Linux needs `chmod +x` again. tar preserves UNIX permissions and ownership, which is why source code ships as `.tar.gz`.

Checksums

A **checksum**, more precisely a **cryptographic hash**, is a fixed-length fingerprint computed from a file's bytes. Change one byte and the fingerprint changes completely. Publishers post the hash next to a download so that you can confirm the file you received is the one they built. SHA-256 is the usual algorithm today; MD5 and SHA-1 still appear in older instructions and are fine for detecting accidental corruption, not for security. [12]

Bash/zsh:

```
$ sha256sum project.tar.gz
7af027f849b8daded806... project.tar.gz
$ sha256sum project.tar.gz > SHA256SUMS
$ sha256sum -c SHA256SUMS
project.tar.gz: OK
```

`sha256sum` prints the hash and the filename; `-c` reads such a file and checks every entry. macOS has no `sha256sum` by default; `shasum -a 256` produces the same output. [13]

PowerShell:

```
$ Get-FileHash project.zip
$ (Get-FileHash project.zip).Hash -eq 'A1B2...'
```

`Get-FileHash` defaults to SHA-256 and returns an object; compare its `Hash` property with the published value. Command Prompt uses `certutil -hashfile project.zip SHA256`. [14], [15]

A matching hash proves the file is the one the publisher hashed. It does not prove the publisher is trustworthy; that is what signatures are for, which is beyond this book.

BSD and GNU: The Same Names, Different Options

macOS ships the **BSD** versions of the classic UNIX tools, and Linux ships the **GNU** versions. They agree on the everyday options in this book and disagree at the edges, which is where tutorials written on one system fail on the other.

Task	GNU (Linux)	BSD (macOS)
Edit a file in place with <code>sed</code>	<code>sed -i 's/a/b/' f</code>	<code>sed -i '' 's/a/b/' f</code>

Task	GNU (Linux)	BSD (macOS)
Yesterday's date	<code>date -d yesterday</code>	<code>date -v-1d</code>
Resolve a path to its absolute form	<code>readlink -f path</code>	<code>realpath path</code> (macOS 13 and later)
File size in bytes with <code>stat</code>	<code>stat -c %s f</code>	<code>stat -f %z f</code>
Colored <code>ls</code>	<code>ls --color</code>	<code>ls -G</code>
Long options such as <code>--all</code>	Usually accepted	Often absent

Two habits handle the difference. Read the local manual page with `man` rather than trusting a tutorial's options, and prefer the short, POSIX options that both accept when writing a script that must run on both. Homebrew's `coreutils` package installs GNU tools on macOS under names starting with `g`, such as `gsed` and `gdate`, for when you need one. [16], [17]

Windows adds a third variant: the `tar`, `curl`, `find`, and `sort` that ship with Windows are not GNU tools either, and `find` and `sort` in Command Prompt are unrelated programs with the same names, which Chapter 6 warned about.

Key Points

- WSL runs a Linux distribution beside the native Windows shells; `wsl` enters it, `wsl` command runs one command, and `explorer.exe` looks back.
- Windows drives appear under `/mnt` in WSL and the Linux filesystem appears at `\\wsl$` in Windows; `wslpath` converts between them.
- Keep a project on the side that works on it most, and edit the Linux filesystem only through supported routes.
- Line endings and encodings break at the boundary; convert scripts to LF, write UTF-8 without a BOM, and use `.gitattributes` in shared repositories.
- `tar -czf`, `tar -xzf`, `zip -r`, `unzip`, and `Compress-Archive` pack and unpack; list an archive before extracting it.
- `sha256sum`, `shasum -a 256`, and `Get-FileHash` verify that a file is the one the publisher hashed.
- macOS ships BSD tools and Linux ships GNU tools; check `man` for the options that differ.

Level Up

n00bs finish the chapter here. You can move between Windows and Linux on one machine, unpack and verify a download, and recognize a BSD-versus-GNU problem. Hackers make their tools work everywhere.

Read the WSL documentation on `wsl.conf` and `.wslconfig`, and change the default user, memory limit, or mount options. Then explore **WSLg** and run a Linux graphical program.

Investigate **containers** as another way across the boundary: run a Linux shell with Docker on any host, and explain how that differs from WSL and from a virtual machine.

Take a script you wrote in Chapter 11 and make it run unchanged on Linux, macOS, and WSL. Use `ShellCheck`, replace GNU-only options, and test each system. Then read the POSIX specification for one utility you used and compare it with both manual pages.

Learn `gpg --verify` and Sigstore’s `cosign`, and explain what a signature proves that a checksum does not.

Look at how package managers and language ecosystems handle line endings and encodings for you: Git’s `.gitattributes`, `EditorConfig`, and Python’s `newline` parameter. Where do they disagree?

Exercises

1. **What does this do?** Predict what `wsl --list --verbose` shows on a machine with one distribution that is not running, then explain what “running” means for a WSL distribution.
2. **What does this do?** Predict the output of `tar -tzf project.tar.gz | head -1` for an archive made with `tar -czf project.tar.gz project/`, and explain what the trailing slash changes, if anything.
3. **What is the problem?** A script works in WSL but `bash script.sh` from `/mnt/c/Users/student/Desktop` prints `$(\r): command not found`. Explain and fix.
4. **What is the problem?** `sed -i 's/foo/bar/' config.txt` on a Mac reports “undefined label” or creates a file named `config.txts`. Explain what BSD `sed` expected.
5. **Why is it like this?** A zip file made on Windows loses the execute bit on scripts, but a tar file made on Linux keeps it. Explain what each format stores.
6. **Why is it like this?** A checksum that matches does not prove a download is safe. Explain what it does prove, and what else you would need.
7. **How do you** convert a Windows path in your clipboard into a path you can `cd` to inside WSL?
8. **How do you** verify a downloaded file against a published SHA-256 value on Linux, on macOS, and in PowerShell?

References

- [1] Microsoft, “What is the windows subsystem for linux?” Available: <https://learn.microsoft.com/en-us/windows/wsl/about>
- [2] Microsoft, “How to install linux on windows with WSL.” Available: <https://learn.microsoft.com/en-us/windows/wsl/install>
- [3] Microsoft, “Basic commands for WSL.” Available: <https://learn.microsoft.com/en-us/windows/wsl/basic-commands>
- [4] Microsoft, “Working across windows and linux file systems.” Available: <https://learn.microsoft.com/en-us/windows/wsl/filesystems>
- [5] Microsoft, “File permissions for WSL.” Available: <https://learn.microsoft.com/en-us/windows/wsl/file-permissions>
- [6] Git for Windows Project, “Git for windows.” Available: <https://gitforwindows.org/>
- [7] Git Project, “Gitattributes: Defining attributes per path.” Available: <https://git-scm.com/docs/gitattributes>
- [8] GNU Project, “GNU tar manual.” Available: <https://www.gnu.org/software/tar/manual/tar.html>
- [9] Info-ZIP, “Info-ZIP zip(1) and unzip(1) manual pages.” Available: <https://infozip.sourceforge.net/>
- [10] Microsoft, “Compress-archive.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.archive/compress-archive>
- [11] Microsoft, “Tar and curl come to windows.” Available: <https://devblogs.microsoft.com/commandline/tar-and-curl-come-to-windows/>
- [12] National Institute of Standards and Technology, “Hash functions (NIST computer security resource center).” Available: <https://csrc.nist.gov/projects/hash-functions>
- [13] GNU Project, “GNU coreutils manual: sha2 utilities: Print or check SHA-2 digests.” Available: https://www.gnu.org/software/coreutils/manual/html_node/sha2-utilities.html
- [14] Microsoft, “Get-FileHash.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-filehash>
- [15] Microsoft, “Windows commands: certutil.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/certutil>
- [16] Apple, “macOS manual pages (xcode command line tools).” Available: <https://developer.apple.com/library/archive/documentation/Darwin/Reference/ManPages/>
- [17] Homebrew, “Homebrew formula: coreutils.” Available: <https://formulae.brew.sh/formula/coreutils>

