

Lesson 07 — Workstation Hardening for AI Workflows

Your workstation is now an AI-integrated platform whether you planned it or not. This lesson is how to inventory what's on it, reason about the blast radius of each piece, and keep the footprint small.

Audience: Devs + Security + Operators

AI OpSec

Hardening

Notes: Inventory first, harden second. You cannot secure what you have not enumerated.

What you will learn

- **Inventory AI tooling** — how to enumerate every AI-touching component on your workstation, including the ones people forget
 - **Evaluate blast radius** — a structured way to reason about reads, writes, network, exec, and telemetry for each tool
 - **Choose a minimal footprint** — prefer local, explicit-approval, inspectable, telemetry-off-by-default tools over the opposite
-

The AI tooling inventory

Five categories. Each is routinely missed in a casual audit.

- **IDE extensions** — Copilot, Cursor's AI pane, JetBrains AI, Continue, Cody
- **CLI agents** — Claude Code, Codex CLI, aider, shell-GPT-style wrappers
- **MCP servers** — filesystem, git, browser, database, Slack, anything you connected to an agent
- **Local model runners** — Ollama, LM Studio, llama.cpp, text-generation-webui
- **Browser extensions** — summarizers, rewriters, sidebar assistants, "explain this page" add-ons

If you skip any category you have skipped the whole audit.

IDE extensions

- **What they read on autocomplete** — the whole open buffer, often the nearby files, sometimes the whole workspace index
- **What they send** — typically the full local context window, not just the line you're editing
- **How to audit** — read the extension manifest, check declared permissions, run a network inspector against the editor while autocomplete fires, confirm destinations

The surprising part is not "the extension sends data" — it's how much data, how often, and to how many endpoints.

CLI agents

- **Exec scope** — a CLI agent can run commands in your shell. Every command is a potential side-effect
- **Filesystem scope** — reads and writes are scoped to the current working directory at best, and to the whole home directory at worst
- **Tool scope** — which tools the agent can call, with what arguments, with what approval model

A CLI agent with shell access is, in effect, a scripted version of you. Treat it with the trust level you would give a new contractor, not a library.

MCP servers

- An MCP server is effectively a **local daemon with a toolbelt** — a long-running process that exposes tools to any MCP-speaking client
- Each tool is a permission surface: filesystem tools can read and write, shell tools can exec, network tools can egress
- The aggregate permission is the union of every tool the server exposes, not the intersection
- You are trusting the server's author, the transport, and every client that can talk to it

Audit the server the same way you would audit a service you run in production, because operationally that's what it is.

Local model runners

- **Generally safer for data** — the prompt stays on your machine, the inference happens locally, no provider log is created
- **Watch the auto-update endpoint** — the runner itself phones home, even when inference doesn't
- **Watch telemetry** — usage pings, crash reports, model-download analytics
- **Watch model weights provenance** — where the weights came from, whether they were signed, whether the registry is trusted

Local runners are the least bad category on the data axis. They are not zero risk — the risk just moves from "prompt leakage" to "supply chain."

Browser extensions

The highest-footprint class.

A browser extension with `<all_urls>` permission can read every tab, every form, every page — your bank, your email, your internal admin panels, your password manager's autofill targets, every prompt you type into any other AI.

- "Read and change all your data on all websites" is the default for most AI extensions
- The extension sees more of your life than your OS-level malware scanner does
- Extensions update silently through the browser's store — yesterday's safe publisher can ship today's malware

Assume an installed AI extension sees everything you see in the browser, forever.

The blast-radius table

Every tool gets one row. Columns are the only things you need to reason about.

Tool	Reads	Writes	Net	Exec	Telemetry	Verdict
Cursor (IDE)	Whole repo buffers	Files in workspace	AI provider APIs	Yes (agent mode)	On by default	Restrict

If you cannot fill every column for a tool, you have not audited it.

How to audit a tool before install

- **Publisher** — is it a known organization, a known individual, or an anonymous account with three installs?
- **Manifest** — read it. What permissions does it declare? What hosts does it list?
- **Permissions requested** — compare the requested set to the minimum needed for the advertised feature
- **Network calls** — run it in a sandbox with a proxy, watch where it actually connects
- **Update channel** — how does it update, who controls the channel, can you pin a version?

Five minutes of this before install saves the 10 hours of incident response after.

Rules to remember

- You cannot harden what you haven't inventoried. List every AI-related component first.
- Every tool has a blast radius: reads, writes, network, exec, telemetry. Know it before you install.
- Prefer explicit-approval per action over silent autonomy.
- Auto-update is a supply-chain surface, not a feature. Pin versions where you can.
- Browser extensions are the highest-footprint class. Assume they see everything in every tab.

