

Sandboxing and Least Privilege for AI Agents

For devs, security, and operators · ~14 min read

An autonomous AI agent is a strange thing to run on your machine. It is not a library you called, it is not a service someone else operates, and it is not a script whose steps you reviewed. It is a non-deterministic loop that decides, on every iteration, what to do next — and then it reaches into a toolbox you gave it and does that thing on your filesystem, your network, and your state. Taking that process seriously as an operational risk is the whole job of agent sandboxing. This lesson walks through the five layers you put around an agent before you let it run, and then points at a working reference implementation you can copy.

The threat model

Every autonomous agent decomposes into three components, and each component is a distinct attack vector. The first component is **the model** — the LLM that decides what to do on every step. The model is non-deterministic, prompt-injectable, and its behavior changes without warning between versions. An attacker who controls any input the model reads — a file in the workspace, a web page the agent fetches, an email in an inbox the agent reviews, even the output of another tool — can attempt to steer the agent. This is not a theoretical risk. Prompt injection is a working, documented, routinely-successful attack technique.

The second component is **the toolbox** — the code that actually runs when the model decides it wants to. Shell execution, filesystem reads and writes, network calls, browser automation, code execution. The toolbox is the mechanism by which the model's output becomes side effects on your machine. A compromised model plus an unrestricted toolbox is a remote code execution primitive with a natural-language interface.

The third component is **the state** — everything the agent has access to. The working directory, the scratch files, the environment variables, the tokens, the memory of past steps, the history of the conversation. State is both what the attacker wants to steal and the persistence mechanism they need to maintain a foothold. If your agent writes to `~/.aws/credentials` or reads from `~/.ssh/id_rsa`, you have already lost.

Sandbox design is about making each of these three vectors safe to lose. Assume the model will be injected. Assume the toolbelt will be called with hostile arguments. Assume the state is visible to whoever controls the model. Design for that world, not for the world where the agent is well-behaved.

Why sandboxing and not just "trust the model"

The alternative to sandboxing is some version of "the model won't do that because we told it not to." Every week someone proposes it and every week the same class of incidents proves it wrong. System prompts leak. Jailbreaks work. Models hallucinate tool arguments. New model versions change behavior in ways no one tested. A security control that depends on a non-deterministic component deciding correctly, every time, under adversarial inputs, is not a control — it is a wish.

A non-deterministic component cannot be a security control.

The layers that can be controls are the ones the kernel enforces. Namespaces. Cgroups. iptables. Capabilities. Mount options. These are deterministic, well-understood, battle-tested, and they do not change their mind based on what the agent's conversation history looks like. When the model is compromised — and you should plan on "when," not "if" — the kernel layers are what stop the compromise from becoming an incident. That is defense in depth. The model is the outermost layer and the least reliable one. Everything below it has to hold on its own.

The five layers

A production-grade agent sandbox has five layers stacked on top of each other. Miss any one and the others degrade from "defense in depth" to "defense in optimism."

Layer 1: containerization. Run the agent inside a container — Docker, Podman, or equivalent. A container gives you a separate PID namespace, mount namespace, and network namespace; an isolated view of the filesystem; and, if you enable it, a user namespace that remaps container UIDs to an unprivileged range on the host. Run the agent process as a non-root user inside the container. Set `security_opt: - no-new-privileges:true` so a setuid binary cannot be used to gain privileges later.

Containerization is not a security silver bullet — container escapes exist, kernel bugs happen — but it is the substrate on which every other layer rests, and without it the other layers have nowhere to stand.

Layer 2: network isolation. Put the agent on a Docker network with `internal: true`. This removes the default NAT masquerade rule, so the container has no route out of the host and cannot reach the internet. The default for an agent should be "cannot make outbound network connections." If the agent genuinely needs outbound — say, to call a specific API — you put an allowlist proxy in front of it and give the agent access only to the proxy, which only permits specific destinations. You never give an agent raw internet.

```
networks:
  isolated:
    driver: bridge
    internal: true           # <-- blocks all outbound from the container
    driver_opts:
      com.docker.network.bridge.enable_ip_masquerade: "false"
```

That snippet, taken directly from `ctf-labs-env/docker-compose.yml`, turns off the entire egress attack surface. A prompt-injected agent on this network cannot exfiltrate data, cannot beacon, cannot fetch a second-stage payload. The model can decide to do any of those things and the kernel will refuse.

Layer 3: filesystem scoping. The agent's view of the filesystem should be the smallest possible surface. Mount host paths read-only (`:ro`) whenever the agent only needs to read. Use a single named volume for writable state, mounted at one specific path. Never bind-mount `/`, `/home`, or the user's dotfiles directory — not for convenience, not "just this once," not for debugging. The right question is not "what should I exclude?" but "what is the minimum I must include?" and then mount only that.

Layer 4: capability drops. Linux capabilities break "root" into fine-grained privileges — the ability to change ownership, bind low ports, load kernel modules, override

discretionary access control, and dozens of others. A normal container starts with a handful of capabilities the runtime considers safe. A sandboxed agent should start with none, and add back only what the workload actually needs.

```
cap_drop:
  - ALL
cap_add:
  - CHOWN
  - SETUID
  - SETGID
  - DAC_OVERRIDE
```

That is the literal capability list used by `ctf-labs-env`, and each added cap has a specific justification: the image needs to transition between users at startup (`SETUID` / `SETGID`), chown the home directory to the runtime user (`CHOWN`), and override DAC where the init scripts manipulate files owned by root (`DAC_OVERRIDE`). No network admin, no kernel module loading, no raw socket creation, no process tracing. If you cannot justify a capability in a single sentence, it does not go on the `cap_add` list.

Layer 5: loopback-only port publishing. Docker's default port publishing binds to `0.0.0.0`, which means every host on your LAN — your home network, the coffee shop Wi-Fi, the corporate VPN — can reach the published port. For an agent that is a disaster: the web UI you exposed "just for local testing" is now reachable by anyone who shares a network segment with you. The fix is one character: prefix the published port with `127.0.0.1:`. Docker's DNAT rule still sends local traffic through, so the agent is reachable from the host it runs on, but no one else. This is the difference between "local tool" and "public service" and it is worth being paranoid about.

What the `ctf-labs-env` pattern gives you

The `ctf-labs-env/docker-compose.yml` file in this repository is a concrete reference implementation of all five layers. A single service, bound to loopback-only port `127.0.0.1:6080:6080`, attached to an internal-only bridge network, hardened with `no-new-privileges:true`, started with all capabilities dropped and a minimal add-back list, and with a single named volume (`lab-home`) mounted at `/home/headless/work` for writable state. Everything else in the container is read-only because nothing else is mounted writable. You can start the container, break into it as if you were a prompt-

injected agent, and fail to reach the internet, fail to escape the filesystem, and fail to be reached from the LAN.

Read `ctf-labs-env/docker-compose.yml` before starting the lab. It is short, intentional, and the annotated model for how to wrap any agent you build.

That file is also the direct target of this lesson's lab — you will bring the sandbox up, verify each of the five layers from the outside and from the inside, and then extend it with a scoped writable volume in a way that does not break the other layers.

Writable state without escape

The hardest part of an agent sandbox is usually not "no network, no caps, no writable filesystem." It is "the agent needs to write *somewhere* to do its job." A coding agent needs to create files. A research agent needs to save notes. A test runner needs scratch space. The pattern that threads the needle is a single named volume, mounted at one specific path, and nothing else writable. The volume is isolated from the host filesystem — it lives in Docker's volume storage, not a host bind mount — which means an attacker who writes hostile content into it cannot traverse out to the host's home directory. The path inside the container is narrow, so the agent's tool permissions can be scoped to it without ambiguity. And because the rest of the filesystem is read-only or absent, the attacker has exactly one place to persist data, and your monitoring and cleanup can focus entirely on that place.

When the agent needs more structure — say, separate workspaces per task — you use more named volumes, not bind mounts. "One volume per trust boundary" is a good default. The wrong instinct is to mount a host directory "because it's easier to see the files." That host directory is a jailbreak target the moment the model is compromised.

Rules to remember

- "I trust the model" is not a security control.
- Default deny, explicit allow: nothing reaches out unless you added it.
- Containerize the agent process. Filesystem, network, caps — all scoped.

- Use an internal-only Docker bridge for agents; inbound DNAT still works on a loopback-bound port.
- Drop all capabilities, then add back only what the agent demonstrably needs.
- Writable state goes into a named volume at a single path. Everything else is read-only.