

Lab — Sandbox an AI Agent with `ctf-labs-`

`env`

~25 minutes · hands-on

The `ctf-labs-env/` directory in this repository is a reference implementation of every sandbox layer from the theory. In this lab you will bring it up, verify each layer from the outside and from the inside — the same way an attacker would — and then extend it with a scoped writable volume without breaking the other layers.

Prerequisites

- Docker Desktop or Docker Engine installed and running on the host
- The `ctf-labs-env/` directory present in this repository checkout
- Shell access to the host (any terminal will do)
- Completed the [Theory](#) reading so the five layers make sense before you poke at them

All commands assume your current directory is the repository root unless an exercise says otherwise. Paths are relative to that root.

Exercise 1 — Start the sandbox

Bring the reference sandbox up and confirm it is running.

```
cd ctf-labs-env
docker compose up -d
docker compose ps
```

Expected: one service named `lab` in state `running`, with the port mapping shown as `127.0.0.1:6080->6080/tcp`. The loopback prefix on the port mapping

is the Layer 5 protection — note that it is there even though you did nothing special, because the compose file already specified it.

✓ **Self-check:** `docker compose ps` shows `lab` as `running`, and the ports column starts with `127.0.0.1:`, not `0.0.0.0:`.

Exercise 2 — Verify network isolation from inside the container

A prompt-injected agent inside this container should not be able to reach the internet. Step into the container the way that agent would and try to egress.

```
docker exec -it ctf-lab bash -c 'curl -v --max-time 5 https://example.com'
```

Expected: an error along the lines of `Could not resolve host: example.com` or `Network is unreachable`. The container has no DNS and no route, because the bridge network has `internal: true`.

```
docker exec -it ctf-lab bash -c 'ping -c 2 -W 2 8.8.8.8 2>&1 | tail -3'
```

Expected: 100% packet loss. Direct-to-IP does not help either — there is no masquerade rule and no route.

✓ **Self-check:** both commands fail to egress. If either succeeds, the `internal: true` bridge is not in effect and you should re-check the compose file before moving on.

Exercise 3 — Verify loopback-only inbound

From the **host** (not inside the container), confirm that the noVNC port is only listening on loopback.

```
ss -tln | grep 6080 || netstat -an | grep 6080
```

Expected: the port appears bound to `127.0.0.1:6080`, not to `0.0.0.0:6080` or `*:6080`. Only the host itself can reach it; nothing on the LAN, the coffee shop Wi-Fi, or the VPN can.

✓ **Self-check: the listener address starts with** `127.0.0.1`. **If you see** `0.0.0.0` **or** `*`, **the loopback prefix is missing from the** `ports:` **entry and every host on your network can reach the service.**

Exercise 4 — Inspect the capability drop

Use `docker inspect` to print the exact capability configuration the runtime applied to the container. This bypasses any possible disagreement between the compose file and what was actually loaded.

```
docker inspect ctf-lab --format '{{json .HostConfig.CapDrop}} {{json .HostConfig.CapAdd}}
```

Expected output (drawn from the actual `ctf-labs-env/docker-compose.yml`):

```
[ "ALL" ] [ "CHOWN", "SETUID", "SETGID", "DAC_OVERRIDE" ]
```

That is: every Linux capability dropped, then a short, justified add-back list — `CHOWN`, `SETUID`, `SETGID`, `DAC_OVERRIDE`. No network admin, no raw sockets, no module loading, no ptrace. An agent in this container cannot escalate to host-root via a capability primitive because it has no capabilities to work with.

✓ **Self-check: the output matches exactly. If** `CapAdd` **contains anything beyond those four caps, someone added a cap without justification and you should find out why.**

Exercise 5 — Add a scoped writable volume without escape

Real agents need to write state somewhere. The right pattern is a single named path mounted into the container, with everything else still read-only. You will add one bind mount to the compose file, verify the agent can write to it, and then verify it *cannot* escape to anywhere else.

Open `ctf-labs-env/docker-compose.yml` in your editor and add the following entry under the existing `volumes:` key for the `lab` service (leave the existing `lab-home` entry in place):

```
- ./agent-scratch:/home/headless/agent-scratch
```

Then, from `ctf-labs-env/`:

```
mkdir -p agent-scratch
docker compose up -d
docker exec -it ctf-lab bash -c 'touch /home/headless/agent-scratch/hello'
```

Expected: the `hello` file exists in `/home/headless/agent-scratch/`, owned by the container user. The agent has writable state at exactly the path you defined, and nowhere else.

Now try to escape the scoped path. This should fail:

```
docker exec -it ctf-lab bash -c 'touch /etc/new-file 2>&1 || echo "denied"'
```

Expected: `denied (expected)`. The rest of the filesystem is not writable by the container user, so even a fully-prompt-injected agent cannot drop persistence into `/etc`, `/usr`, or any other system path.

Finally, tear it all down and revert the compose edit so the repo stays clean:

```
docker compose down
git checkout ctf-labs-env/docker-compose.yml
rm -rf agent-scratch
```

✓ **Self-check:** the write to `/home/headless/agent-scratch/hello` succeeded, the write to `/etc/new-file` failed, and after teardown `git status` reports a clean working tree under `ctf-labs-env/`.

The teardown step is not optional. Leaving the bind mount in `docker-compose.yml` or the `agent-scratch/` directory on disk pollutes the reference implementation for the next person who reads it — including future-you. Always finish the lab with `git status` showing a clean tree.