Linux Basics — Job Readiness Companion

Purpose

Turn an abstract statement about a “risky server” into a checkable inventory of what is running, what it touches, and what boundary failed.

Common artifacts you would see

- Incident notes that reconstruct what ran, what changed, and what was impacted
- Operations runbooks that define baselines and safe verification steps
- Security review comments that ask for concrete host, process, path, and interface details
- Audit findings that cite specific machines, configs, and persistence mechanisms
- Change requests that justify why a process, timer, or file placement must change

---

Junior expectations

- Name the specific host when describing an issue
- Identify the runtime truth as a process or recurring trigger, not a role label
- Identify the key paths the process reads and writes, especially configs, logs, temp state, and secrets at rest
- Identify the interface that triggers or influences execution, including service managers, schedulers, and local ingestion points
- Separate observed facts from assumptions and record what you used as evidence
- Translate a host-level observation into one clear risk path with observable damage

---

Junior guardrails

- Do not describe “the server” or “the service” without naming the host and process
- Do not treat tool output as the conclusion; interpret it as evidence
- Do not assume persistence; prove whether the process persists or is a one-off
- Do not propose fixes before you can state what is running and what it touches
- Do not use vague language (“misconfigured,” “exposed,” “unsafe”) in place of concrete host facts

---

How issues are discussed professionally

In operations and security work, Linux issues are discussed as **host → runtime truth → touch points → trigger → boundary → observable outcome**.

A credible explanation sounds like: “On `app-prod-03`, the `reporting-sync` process runs every 5 minutes via a systemd timer. It reads credentials from `/etc/reporting/creds.json` and writes exports to `/var/tmp/exports/`. The timer is running as `root`, so the process can read secrets and write outside the intended service directory. The observable impact is credentials appearing in process arguments and exported files persisting in a world-readable temp location.”

Not: “The server is misconfigured”

Your job is to make the first statement possible.

Translation patterns you must be able to use

Pattern 1: Unexpected persistence

Structure:

- Host (where you observed it)
- Runtime truth (process name and PID, or recurring unit/job)
- Trigger (systemd, cron, user login, local ingestion)
- Persistence mechanism (unit file, timer, cron entry, rc scripts, autostart)
- Observable damage (what persists, what it changes, what would be noticed)

Example: “On `web-01`, a `python3 /opt/bin/collector.py` process reappears after termination. It is restarted by `systemd` via `collector.service` with `Restart=always`. The unit is enabled and starts at boot. The observable impact is continuous outbound connections and repeated writes to `/var/log/collector.log` that grow disk usage and trigger storage alerts.”

Pattern 2: Unusual placement

Structure:

- Host
- Artifact (file/binary/script/config) and full path

- Ownership and permissions (who controls it)
- Execution path (what runs it and under which identity)
- Boundary failure (why this placement bypasses intended controls)
- Observable damage

Example: “On `batch-02`, an executable named `backup` exists at `/tmp/backup` and is owned by `root` but writable by group `ops`. A cron job runs `/tmp/backup` nightly as `root`. That placement and permission combination bypasses expected review and packaging controls. The observable impact is that changes to the file immediately change privileged behavior, visible as altered backup destinations and unexpected file modifications under `/var/lib/`.”

Pattern 3: Unclear boundaries (what can influence execution)

Structure:

- Host
- Process (what is running)
- Inputs (files, env vars, sockets, directories, config sources)
- Interface (who/what can write those inputs)
- Failure mode (how influence becomes execution or unsafe behavior)
- Observable damage

Example: “On `api-04`, `nginx` loads configuration from `/etc/nginx/conf.d/*.conf`. A deployment script writes configs from a shared NFS mount that is writable by multiple service accounts. That interface allows non-owners to influence routing and upstream targets. The observable impact is traffic being routed to unexpected backends and configuration diffs appearing without matching deployment records.”

How this skill is used on the job

Context 1: Incident reconstruction

- What question you help answer: “What actually ran, when did it start, and what changed?”
- Inputs you work from: process lists, unit files, logs, timestamps, package history
- What your explanation enables: a timeline that identifies the real trigger and the affected files or services

Context 2: Operations baselines and safe verification

- What question you help answer: “What is normal on this host, and how do we verify safely?”

- Inputs you work from: runbooks, configuration management, service definitions, monitoring
- What your explanation enables: repeatable checks that separate drift from expected behavior

Context 3: Security reviews of host posture

- What question you help answer: “What runs with privilege, what does it touch, and what boundary is assumed?”
- Inputs you work from: service configs, file permissions, secrets locations, startup mechanisms
- What your explanation enables: targeted hardening that reduces risk without breaking routine operations

Interview translation

Question: “How do you investigate a Linux issue when you don’t know what’s going on?”

Answer: “I start by picking one machine and getting oriented. I check what’s actually running and what it’s touching, instead of assuming the role label is accurate. I look for the process that seems tied to the problem, then I check what files it reads and writes and how it gets started, like a service, a scheduled job, or something local that triggers it. Once I can point to the host, the process, and what it touches, I can explain what would actually fail in a way a teammate could recognize, and then suggest a change that makes the bad outcome less likely without making everyday work painful.”

Answer pattern:

- Name the host
- Prove runtime truth (process and trigger)
- Identify touch points (paths read/write)
- Identify interfaces and boundaries (who can influence inputs)
- Translate into an observable outcome
- Propose a defensible change and how you would verify it

Common early-career mistakes

- Treating a role label (“web server,” “db host”) as runtime truth instead of verifying running processes
- Listing commands or tools used rather than describing what the evidence implies

- Assuming persistence (“it keeps coming back”) without identifying the restart mechanism
- Proposing remediation before establishing what the process touches and what depends on it
- Using vague labels (“unsafe,” “exposed”) instead of naming the exact host facts and boundaries

What good looks like / If you forget everything else

- You can name the host, the process, and the trigger that starts it
- You can list the key paths the process reads and writes, including configs and secrets
- You can explain who can influence execution through files, directories, or service interfaces
- You separate observed evidence from assumptions and can point to what supports your claim
- Your proposed change is small, testable, and paired with a verification step

If a teammate can repeat your explanation and reach the same conclusion, the work is job-ready.