Linux Basics

Notes

Security conversations go wrong when the system stays abstract.

> *“Something is misconfigured.”*
> *“The server is risky.”*
> *“That machine needs to be secured.”*

Those statements do not tell a team what exists on the host, where to look for evidence, or what would need to change.

The goal is to make Linux concrete: named machines running real programs that read and write real files and interact through well-defined boundaries. Once those facts are visible, investigations stop being guesswork.

What an operating system is

An operating system is the layer that turns hardware into a controlled, shared machine.

In practice, it provides and enforces:

OS responsibility	**What it controls in practice**
Scheduling	Which program runs next, for how long, and when it is preempted
Memory	What memory a program can use, and isolation between programs
Storage	Filesystems, files, and how data persists across restarts
Devices	Access to disks, terminals, clocks, and other hardware
Processes	Starting, stopping, signalling, and tracking program execution
Security boundaries	Where “allowed” and “not allowed” are enforced by the system

In security work, the operating system matters because it is the place where intent can be enforced—or where drift quietly defeats it.

The main parts of Linux

Linux becomes understandable when the major pieces are named.

Kernel

The kernel is the boundary.

It is responsible for:

- CPU scheduling and memory management
- filesystems and device drivers
- system calls (the privileged API user-space programs invoke)
- core isolation and permission checks

Most security failures are not “in the kernel.” They arise when user-space programs misuse the kernel’s capabilities.

User space

User space is where ordinary programs run: shells, services, scripts, and tools.

It includes:

- service managers (commonly systemd) that start and supervise long-lived processes
- schedulers (cron, timers) that trigger execution
- libraries and runtimes that applications depend on
- the programs that implement the organization’s actual workload

Filesystems and placement

Linux is opinionated about where things tend to live.

You do not need to memorize a directory taxonomy. You do need the rule:
Placement is evidence.

When a binary, config, secret, log, or dataset is not where you expect it to be, that mismatch is often the story.

Live system views

Linux exposes system state as inspectable “files.”

- /proc shows live process and system state
- /sys shows kernel and device state

These matter because observation is not guesswork: tools are reading the same underlying state the kernel maintains.

The core idea

Linux is not a blur. It is an inventory.

A useful Linux description is made of facts that the OS itself can confirm:

Linux fact	What it means in practice
Host	Which specific machine does the observation apply to
Process	What is actually executing on the system, and whether it persists over time
Path	What files and directories the process reads from and writes to
Interface	How execution is triggered and how components interact, starting with local boundaries (service managers, schedulers, IPC, files)

If a claim cannot be anchored to those facts, it cannot be reliably investigated, fixed, or verified.

Two invariant questions

Before interpretation, before severity, before fixes:

1. What is running?
2. What does it touch?

“What it touches” means both:

- paths (files and directories) and
- interfaces (how it is invoked and how it interacts with other components)

Host

A host is the unit of place.

Linux work fails when people talk in roles instead of machines.

- Weak: “the reporting gateway.”
- Strong: rpt-gw-03

A host anchor makes work repeatable: you can revisit it, collect evidence again, and prove that a change affected reality.

Process

A process is the unit of runtime truth.

When someone says “the service is running,” they mean: a process exists and persists over time.

A process can be described without tool syntax. The minimum facts are:

- what it is (name/executable)
- whether it persists (long-lived vs short-lived)
- how it starts (service manager, scheduler, interactive session)
- what identity it runs as (details deepen later)

Service vs one-off command

A recurring mistake is treating activity as presence.

- One-off command: runs, does work, exits.
- Service: persists; creates durable behaviour; often restarts after failure.

If a claim depends on ongoing behaviour, there should be persistence somewhere: a long-lived process, a recurring scheduler, or a stable interface.

Path

At this layer, configuration and behaviour are represented as files.

- Configuration is a path.
- Logs are paths.
- Secrets at rest are paths.
- Output and intermediate state are paths.

Linux becomes tractable when you can say:

- which file the process reads to decide what to do
- which directories it writes to while doing it
- where evidence of its behaviour accumulates

Placement is evidence

Linux systems have conventions. You do not need to memorize them to use them.

You only need the rule: location carries meaning.

Unusual placement often indicates:

- drift from previous deployments
- troubleshooting artifacts that never got removed
- a process running outside the intended lifecycle
- multiple competing “sources of truth” for configuration

Interface

An interface is the unit of interaction.

Even before networking, Linux systems have interfaces that matter:

- service managers starting and restarting processes
- schedulers triggering execution (cron, timers)
- local IPC surfaces (Unix domain sockets, pipes, shared files)
- file-based ingestion points (a directory another process consumes)
- device and terminal interaction (who can run what, and from where)

An interface is only useful when it names the boundary and the trigger.

“Reachable” is not only a network concept. It also means: which local identities can invoke a service, write into a watched directory, connect to a socket, or influence a process through its inputs.

Security issues at the OS layer

Most OS-level security failures are not the result of exotic exploits. They are predictable outcomes of weak boundaries and accumulated drift.

Over-broad identity and privilege

- services run as overly powerful identities
- privileged helpers exist without tight constraints
- components have more permissions than they need

Weak isolation between components

- processes can read or write data, but they should not
- shared directories and sockets allow unintended influence
- multiple services share the same identity, blurring accountability

Secrets handled as ordinary files

- credentials in readable configs or env files
- secrets leaking into logs, temp output, crash artifacts
- “temporary” copies of keys or tokens created during troubleshooting

Unintended interfaces

- processes can be triggered in ways the team does not track

- debug or admin surfaces left enabled
- schedulers or ingestion points used as accidental control planes

Drift and lifecycle failures

- old services persist after migrations
- duplicate configs diverge
- startup scripts and timers accumulate “just for now” behaviour

Observability gaps

- logs exist, but do not support reconstruction
- no reliable inventory of what is running and what it touches
- inability to prove that a fix changed runtime behaviour

Examples

Config File Leaks Database Access Parameters

A reporting gateway service runs on a Linux host and generates PDFs by querying a client reports database. Operations staff sometimes log into the host to inspect the service during incidents. To simplify troubleshooting, the configuration file containing database connection parameters is made readable to all local users and never reset

Client reports database

→ Routine operational access to rpt-gw-03

→ Database parameters live in a config file readable beyond the service identity

→ Unauthorized local users or processes can extract access parameters

and read client report data outside intended workflows.

This is a confidentiality issue because the reporting service continues to function, and no data is changed or destroyed. The failure is that protected information becomes readable by identities that should not have access to it. The observable damage is unauthorized access to private client report data.

Legacy Process Creates an Undocumented Execution Path

A reporting host is expected to run a single gateway service managed through the standard service lifecycle. During a previous migration, an older report generator is left running to compare outputs and was never removed. Both the old and new processes read the same templates and write reports to the same output directories.

Integrity of generated client reports

→ Normal process startup and supervision on rpt-gw-03

→ A legacy report generator persists, reads the same templates, and writes to the same report output locations without current validation

→ Reports can be generated or altered through an undocumented runtime path that the team does not track.

This is an integrity issue because outputs may still appear correct even when produced through an uncontrolled path. The observable damage is that reports can be altered or regenerated while still appearing legitimate.

Scheduled Cleanup Deletes Active Working State

A cleanup task runs regularly on the reporting host to remove temporary files. After a configuration change, the reporting service begins using a directory that had previously been marked for deletion. The cleanup task continues to run on schedule.

Ability to generate client reports on demand

→ Scheduled cleanup timer on the reporting host

→ Cleanup script deletes a directory now used by the reporting process during generation

→ Report generation intermittently fails or produces incomplete outputs during typical operation.

This is an availability issue because the primary failure is the inability to generate reports when needed. The observable damage is repeated, predictable failure to generate reports under routine load.

Persistent Helper Process Reads Secrets From a Nonstandard Path

During an incident, a diagnostic helper process is added to observe the reporting service and collect the state. To keep it working across restarts, it is added to the startup sequence. To avoid breaking the helper, a copy of service credentials is placed in a nonstandard directory that the helper can read. Both remain after the incident is resolved.

Confidentiality of service credentials on reporting hosts

→ Routine local access on rpt-gw-03

→ Persistent helper process reads credential material from a nonstandard path left behind during troubleshooting

→ Credentials can be copied locally and reused to access reporting services or databases.

This is primarily a confidentiality issue because sensitive credentials become accessible to an unauthorized runtime component. It also creates secondary integrity and availability risk because containment typically requires credential rotation and service disruption. The observable damage is unauthorized credential access and the operational interruption that follows remediation.

What to notice

- Linux becomes actionable when it is described in terms of hosts, processes, paths, and interfaces.
- A baseline is created from “usual processes” and “usual locations” for a host role.
- Deviations are evidence, not intuition.
- OS-level security issues are usually boundary failures and lifecycle drift, not mystery attacks.