REBASH PRODUCTION ENGINEERING SERIES

VOLUME 1

LINUX

COURSE BOOK

LEARN.
PRACTICE.
AUTOMATE.
SUCCEED.

From Beginner to Production Engineer

Master Linux. Solve Real Problems.

- Story-Driven Learning
- Hands-on Labs
- Production Focused
- Troubleshoot Like a Pro
- Interview Ready

25 Chapters | **25** Labs | Real Scenarios | Interview Q&A

SHAIK KHADAR BASHA

Cloud Architect | DevOps Trainer | Founder, REBASH Academy

REBASH ACADEMY

LEARN • PRACTICE • BUILD • AUTOMATE

Copyright

Labs and online materials: https://rebash.in

Disclaimer: Commands and examples are for learning on practice systems. Always test changes on non-production hosts. The authors accept no liability for damage arising from use of this material.

British English spelling is used throughout unless quoting product names or commands.

Generated from the REBASH Academy curriculum on 2026-08-02.

About the author

Shaik Khadar Basha

Cloud & Platform Architect · Founder, REBASH Academy

Cloud & Platform Architect | AWS | GCP | Azure | Kubernetes | Terraform | DevSecOps | Platform Engineering | AI Infrastructure

Greater Hyderabad Area, India

LinkedIn · rebash.in

Shaik Khadar Basha is a Cloud and Platform Architect with more than a decade of experience designing, automating, and operating cloud-native platforms. His work spans Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, Kubernetes, Terraform, DevSecOps, platform engineering, and AI infrastructure.

He currently works as Cloud Architect at BreachLock Inc in Hyderabad, leading delivery across engineering teams that build secure, cloud-native applications on Python, React, Kubernetes, AWS, and GCP. Earlier roles at Wipro, Unisys, Orcapod, Nyletech Solutions, and Netskope covered observability, continuous integration and continuous delivery (CI/CD), Infrastructure as Code (IaC), and SaaS security.

His credentials include Red Hat Certified Engineer (RHCE), Red Hat Certified System Administrator (RHCSA), Oracle Cloud Infrastructure Architect Associate, ISO 27001 Cyber Security Expert, and Certified Ethical Hacker (CEH). He has also held AWS Solutions Architect Associate and HashiCorp Terraform Associate certifications. He founded REBASH Academy to teach practical Cloud and DevOps skills through hands-on tutorials and labs.

Education: Master's degree in Computer Science, Jawaharlal Nehru Technological University Kakinada (JNTUK); Bachelor's degree in Mathematics, Acharya Nagarjuna University.

Contents

Module 1 · Fundamentals

Chapter 1. Linux Fundamentals — Distributions and Architecture 19
- Overview 19
- Prerequisites 19
- Learning Objectives 19
- Architecture 19
- Theory 20
- What it is 20
- Why it matters 20
- How it works 20
- Key concepts and comparisons 21
- Common pitfalls 21
- Hands-on Lab 21
- Objective 21
- Prerequisites 19
- Lab environment 21
- Real-world scenario 21
- Step-by-step tasks 22
- Validation steps 22
- Common errors and fixes 23
- Challenge exercise 23
- Learning outcomes 23
- Cleanup 23
- Validation 23
- Code Walkthrough 23
- Security Considerations 23
- Common Mistakes 24
- Best Practices 24
- Troubleshooting 24
- Summary 24
- Interview Questions 24

Chapter 2. Boot Process and Filesystem Hierarchy 26
- Overview 26
- Prerequisites 26
- Learning Objectives 26
- Architecture 26
- Theory 27
- What it is 27
- Why it matters 27
- How it works 27
- Key concepts and comparisons 28
- Common pitfalls 28
- Hands-on Lab 28
- Objective 28
- Prerequisites 26
- Lab environment 28
- Real-world scenario 29
- Step-by-step tasks 29
- Validation steps 30
- Common errors and fixes 30
- Challenge exercise 30
- Learning outcomes 30
- Cleanup 30
- Validation 30

Code Walkthrough 30
Security Considerations 31
Common Mistakes 31
Best Practices 31
Troubleshooting 31
Summary 31
Interview Questions 31

Module 2 · Command Line

Chapter 3. Essential Linux Commands 33
Overview 33
Prerequisites 33
Learning Objectives 33
Architecture 33
Theory 34
What it is 34
Why it matters 34
How it works 34
Key concepts and comparisons 35
Common pitfalls 35
Hands-on Lab 35
Objective 35
Prerequisites 33
Lab environment 35
Real-world scenario 35
Step-by-step tasks 35
Validation steps 37
Common errors and fixes 37
Challenge exercise 37
Learning outcomes 37
Cleanup 37
Validation 37
Code Walkthrough 37
Security Considerations 38
Common Mistakes 38
Best Practices 38
Troubleshooting 38
Summary 38
Interview Questions 38

Module 3 · Filesystem

Chapter 4. Filesystem Paths, Links, Mounts, and Inodes 40
Overview 40
Prerequisites 40
Learning Objectives 40
Architecture 40
Theory 41
What it is 41
Why it matters 41
How it works 41
Key concepts and comparisons 42
Common pitfalls 42
Hands-on Lab 42
Objective 42
Prerequisites 40
Lab environment 42
Real-world scenario 42

Step-by-step tasks 42
Validation steps 44
Common errors and fixes 44
Challenge exercise 44
Learning outcomes 44
Cleanup 44
Validation 44
Code Walkthrough 44
Security Considerations 44
Common Mistakes 45
Best Practices 45
Troubleshooting 45
Summary 45
Interview Questions 45
Chapter 5. Disk Usage and File Attributes 47
Overview 47
Prerequisites 47
Learning Objectives 47
Architecture 47
Theory 48
What it is 48
Why it matters 48
How it works 48
Common pitfalls 49
Hands-on Lab 49
Objective 49
Prerequisites 47
Lab environment 49
Real-world scenario 49
Step-by-step tasks 49
Validation steps 50
Common errors and fixes 50
Challenge exercise 50
Learning outcomes 50
Cleanup 50
Validation 51
Code Walkthrough 51
Security Considerations 51
Common Mistakes 51
Best Practices 51
Troubleshooting 52
Summary 52
Interview Questions 52
Module 4 · Users & Permissions
Chapter 6. Users, Groups, and sudo 54
Overview 54
Prerequisites 54
Learning Objectives 54
Architecture 54
Theory 55
What it is 55
Why it matters 55
How it works 55
Key concepts and comparisons 55
Common pitfalls 56
Hands-on Lab 56

Objective 56
Prerequisites 54
Lab environment 56
Real-world scenario 56
Step-by-step tasks 56
Validation steps 57
Common errors and fixes 58
Challenge exercise 58
Learning outcomes 58
Cleanup 58
Validation 58
Code Walkthrough 58
Security Considerations 59
Common Mistakes 59
Best Practices 59
Troubleshooting 59
Summary 59
Interview Questions 59
Chapter 7. Permissions, ACLs, and Special Bits 61
Overview 61
Prerequisites 61
Learning Objectives 61
Architecture 61
Theory 62
What it is 62
Why it matters 62
How it works 62
Common pitfalls 62
Hands-on Lab 63
Objective 63
Prerequisites 61
Lab environment 63
Real-world scenario 63
Step-by-step tasks 63
Validation steps 64
Common errors and fixes 64
Challenge exercise 64
Learning outcomes 64
Cleanup 64
Validation 65
Code Walkthrough 65
Security Considerations 65
Common Mistakes 65
Best Practices 65
Troubleshooting 66
Summary 66
Interview Questions 66
Module 5 · Text Processing
Chapter 8. Text Processing with grep, sed, and awk 68
Overview 68
Prerequisites 68
Learning Objectives 68
Architecture 68
Theory 69
What it is 69
Why it matters 69

How it works 69
Key concepts and comparisons 70
Common pitfalls 70
Hands-on Lab 70
Objective 70
Prerequisites 68
Lab environment 70
Real-world scenario 70
Step-by-step tasks 70
Validation steps 72
Common errors and fixes 72
Challenge exercise 72
Learning outcomes 72
Cleanup 72
Validation 72
Code Walkthrough 72
Security Considerations 73
Common Mistakes 73
Best Practices 73
Troubleshooting 73
Summary 73
Interview Questions 73

Module 6 · Processes

Chapter 9. Process Management 75
Overview 75
Prerequisites 75
Learning Objectives 75
Architecture 75
Theory 76
What it is 76
Why it matters 76
How it works 76
Key concepts and comparisons 77
Common pitfalls 77
Hands-on Lab 77
Objective 77
Prerequisites 75
Lab environment 77
Real-world scenario 77
Step-by-step tasks 77
Validation steps 78
Common errors and fixes 79
Challenge exercise 79
Learning outcomes 79
Cleanup 79
Validation 79
Code Walkthrough 79
Security Considerations 79
Common Mistakes 80
Best Practices 80
Troubleshooting 80
Summary 80
Interview Questions 80

Module 7 · Services & Boot

Chapter 10. systemd Services and journalctl 82

Overview 82
Prerequisites 82
Learning Objectives 82
Architecture 82
Theory 83
What it is 83
Why it matters 83
How it works 83
Key concepts and comparisons 84
Common pitfalls 84
Hands-on Lab 84
Objective 84
Prerequisites 82
Lab environment 84
Real-world scenario 84
Step-by-step tasks 84
Validation steps 86
Common errors and fixes 86
Challenge exercise 86
Learning outcomes 86
Cleanup 86
Validation 86
Code Walkthrough 87
Security Considerations 87
Common Mistakes 87
Best Practices 87
Troubleshooting 87
Summary 88
Interview Questions 88
Chapter 11. systemd Targets, Timers, and Boot 89
Overview 89
Prerequisites 89
Learning Objectives 89
Architecture 89
Theory 90
What it is 90
Why it matters 90
How it works 90
Key concepts and comparisons 91
Common pitfalls 91
Hands-on Lab 91
Objective 91
Prerequisites 89
Lab environment 91
Real-world scenario 91
Step-by-step tasks 91
Validation steps 93
Common errors and fixes 93
Challenge exercise 93
Learning outcomes 93
Cleanup 93
Validation 93
Code Walkthrough 93
Security Considerations 94
Common Mistakes 94
Best Practices 94

Troubleshooting 94
Summary 94
Interview Questions 94

Module 8 · Storage

Chapter 12. Disks, Partitions, and Filesystems 96
Overview 96
Prerequisites 96
Learning Objectives 96
Architecture 96
Theory 96
What it is 96
Why it matters 97
How it works 97
Common pitfalls 97
Hands-on Lab 97
Objective 97
Prerequisites 96
Lab environment 97
Real-world scenario 97
Step-by-step tasks 98
Validation steps 98
Common errors and fixes 99
Challenge exercise 99
Learning outcomes 99
Cleanup 99
Validation 99
Code Walkthrough 99
Security Considerations 99
Common Mistakes 100
Best Practices 100
Troubleshooting 100
Summary 100
Interview Questions 100
Chapter 13. LVM, Swap, and Disk Monitoring 102
Overview 102
Prerequisites 102
Learning Objectives 102
Architecture 102
Theory 103
What it is 103
Why it matters 103
How it works 103
Common pitfalls 103
Hands-on Lab 103
Objective 103
Prerequisites 102
Lab environment 103
Real-world scenario 104
Step-by-step tasks 104
Validation steps 105
Common errors and fixes 105
Challenge exercise 105
Learning outcomes 105
Cleanup 105
Validation 105
Code Walkthrough 105

Security Considerations 105
Common Mistakes 106
Best Practices 106
Troubleshooting 106
Summary 106
Interview Questions 106

Module 9 · Networking

Chapter 14. Linux Networking Tools 108
Overview 108
Prerequisites 108
Learning Objectives 108
Architecture 108
Theory 109
What it is 109
Why it matters 109
How it works 109
Key concepts and comparisons 110
Common pitfalls 110
Hands-on Lab 110
Objective 110
Prerequisites 108
Lab environment 110
Real-world scenario 110
Step-by-step tasks 111
Validation steps 111
Common errors and fixes 112
Challenge exercise 112
Learning outcomes 112
Cleanup 112
Validation 112
Code Walkthrough 112
Security Considerations 112
Common Mistakes 113
Best Practices 113
Troubleshooting 113
Summary 113
Interview Questions 113
Chapter 15. SSH and Remote Access 115
Overview 115
Prerequisites 115
Learning Objectives 115
Architecture 115
Theory 115
What it is 115
Why it matters 116
How it works 116
Key concepts and comparisons 116
Common pitfalls 116
Hands-on Lab 116
Objective 117
Prerequisites 115
Lab environment 117
Real-world scenario 117
Step-by-step tasks 117
Validation steps 118
Common errors and fixes 119

Challenge exercise 119
Learning outcomes 119
Cleanup 119
Validation 119
Code Walkthrough 120
Security Considerations 120
Common Mistakes 120
Best Practices 120
Troubleshooting 120
Summary 121
Interview Questions 121

Module 10 · Packages

Chapter 16. Package Management 122
Overview 122
Prerequisites 122
Learning Objectives 122
Architecture 122
Theory 123
What it is 123
Why it matters 123
How it works 123
Key concepts and comparisons 123
Common pitfalls 123
Hands-on Lab 124
Objective 124
Prerequisites 122
Lab environment 124
Real-world scenario 124
Step-by-step tasks 124
Validation steps 125
Common errors and fixes 125
Challenge exercise 125
Learning outcomes 125
Cleanup 125
Validation 125
Code Walkthrough 126
Security Considerations 126
Common Mistakes 126
Best Practices 126
Troubleshooting 126
Summary 127
Interview Questions 127

Module 11 · Scheduling

Chapter 17. Scheduling with cron, at, and Timers 128
Overview 128
Prerequisites 128
Learning Objectives 128
Architecture 128
Theory 129
What it is 129
Why it matters 129
How it works 129
Key concepts and comparisons 130
Common pitfalls 130
Hands-on Lab 130

Objective 130
Prerequisites 128
Lab environment 130
Real-world scenario 130
Step-by-step tasks 130
Validation steps 132
Common errors and fixes 132
Challenge exercise 132
Learning outcomes 132
Cleanup 132
Validation 133
Code Walkthrough 133
Security Considerations 133
Common Mistakes 133
Best Practices 133
Troubleshooting 134
Summary 134
Interview Questions 134

Module 12 · Logging & Monitoring

Chapter 18. Logging — syslog, journald, and logrotate 136
Overview 136
Prerequisites 136
Learning Objectives 136
Architecture 136
Theory 137
What it is 137
Why it matters 137
How it works 137
Common pitfalls 137
Hands-on Lab 137
Objective 138
Prerequisites 136
Lab environment 138
Real-world scenario 138
Step-by-step tasks 138
Validation steps 139
Common errors and fixes 139
Challenge exercise 139
Learning outcomes 139
Cleanup 139
Validation 139
Code Walkthrough 139
Security Considerations 140
Common Mistakes 140
Best Practices 140
Troubleshooting 140
Summary 140
Interview Questions 140
Chapter 19. Host Monitoring — vmstat, iostat, and sar 142
Overview 142
Prerequisites 142
Learning Objectives 142
Architecture 142
Theory 143
What it is 143
Why it matters 143

How it works 143
Common pitfalls 143
Hands-on Lab 143
Objective 143
Prerequisites 142
Lab environment 143
Real-world scenario 144
Step-by-step tasks 144
Validation steps 144
Common errors and fixes 145
Challenge exercise 145
Learning outcomes 145
Cleanup 145
Validation 145
Code Walkthrough 145
Security Considerations 145
Common Mistakes 145
Best Practices 146
Troubleshooting 146
Summary 146
Interview Questions 146

Module 13 · Security

Chapter 20. SSH Hardening and Firewalls 148
Overview 148
Prerequisites 148
Learning Objectives 148
Architecture 148
Theory 149
What it is 149
Why it matters 149
How it works 149
Key concepts and comparisons 149
Common pitfalls 149
Hands-on Lab 150
Objective 150
Prerequisites 148
Lab environment 150
Real-world scenario 150
Step-by-step tasks 150
Validation steps 152
Common errors and fixes 152
Challenge exercise 152
Learning outcomes 152
Cleanup 152
Validation 153
Code Walkthrough 153
Security Considerations 153
Common Mistakes 153
Best Practices 153
Troubleshooting 154
Summary 154
Interview Questions 154
Chapter 21. SELinux, AppArmor, Fail2Ban, Auditd, and PAM 156
Overview 156
Prerequisites 156
Learning Objectives 156

Architecture 156
Theory 157
What it is 157
Why it matters 157
How it works 157
Key concepts and comparisons 157
Common pitfalls 158
Hands-on Lab 158
Objective 158
Prerequisites 156
Lab environment 158
Real-world scenario 158
Step-by-step tasks 158
Validation steps 160
Common errors and fixes 160
Challenge exercise 160
Learning outcomes 160
Cleanup 160
Validation 161
Code Walkthrough 161
Security Considerations 161
Common Mistakes 161
Best Practices 161
Troubleshooting 162
Summary 162
Interview Questions 162

Module 14 · Containers & Cloud

Chapter 22. Containers — Namespaces, cgroups, OverlayFS, and OCI 164
Overview 164
Prerequisites 164
Learning Objectives 164
Architecture 164
Theory 165
What it is 165
Why it matters 165
How it works 165
Key concepts and comparisons 165
Common pitfalls 166
Hands-on Lab 166
Objective 166
Prerequisites 164
Lab environment 166
Real-world scenario 166
Step-by-step tasks 166
Validation steps 167
Common errors and fixes 168
Challenge exercise 168
Learning outcomes 168
Cleanup 168
Validation 168
Code Walkthrough 168
Security Considerations 168
Common Mistakes 169
Best Practices 169
Troubleshooting 169
Summary 169

Interview Questions 169

Module 15 · Troubleshooting

Chapter 23. Troubleshooting Linux Systems 171
Overview 171
Prerequisites 171
Learning Objectives 171
Architecture 171
Theory 172
What it is 172
Why it matters 172
How it works 172
Common pitfalls 172
Hands-on Lab 172
Objective 172
Prerequisites 171
Lab environment 172
Real-world scenario 173
Step-by-step tasks 173
Validation steps 174
Common errors and fixes 174
Challenge exercise 174
Learning outcomes 174
Cleanup 174
Validation 174
Code Walkthrough 175
Security Considerations 175
Common Mistakes 175
Best Practices 175
Troubleshooting 175
Summary 175
Interview Questions 175

Module 16 · Production Linux

Chapter 24. Production Hardening and Performance 177
Overview 177
Prerequisites 177
Learning Objectives 177
Architecture 177
Theory 178
What it is 178
Why it matters 178
How it works 178
Common pitfalls 178
Hands-on Lab 178
Objective 178
Prerequisites 177
Lab environment 178
Real-world scenario 179
Step-by-step tasks 179
Validation steps 180
Common errors and fixes 180
Challenge exercise 180
Learning outcomes 180
Cleanup 180
Validation 180
Code Walkthrough 180

Security Considerations 180
Common Mistakes 181
Best Practices 181
Troubleshooting 181
Summary 181
Interview Questions 181
Chapter 25. Backup, Disaster Recovery, and Capacity 183
Overview 183
Prerequisites 183
Learning Objectives 183
Architecture 183
Theory 184
What it is 184
Why it matters 184
How it works 184
Common pitfalls 184
Hands-on Lab 184
Objective 184
Prerequisites 183
Lab environment 184
Real-world scenario 185
Step-by-step tasks 185
Validation steps 186
Common errors and fixes 186
Challenge exercise 186
Learning outcomes 186
Cleanup 186
Validation 186
Code Walkthrough 186
Security Considerations 186
Common Mistakes 187
Best Practices 187
Troubleshooting 187
Summary 187
Interview Questions 187

List of figures

Figure 1.1: Linux Fundamentals — Distributions and Architecture 20
Figure 2.1: Boot Process and Filesystem Hierarchy 27
Figure 3.1: Essential Linux Commands 34
Figure 4.1: Filesystem Paths, Links, Mounts, and Inodes 41
Figure 5.1: Disk Usage and File Attributes 48
Figure 6.1: Users, Groups, and sudo 55
Figure 7.1: Permissions, ACLs, and Special Bits 62
Figure 8.1: Text Processing with grep, sed, and awk 69
Figure 9.1: Process Management 76
Figure 10.1: systemd Services and journalctl 83
Figure 11.1: systemd Targets, Timers, and Boot 90
Figure 12.1: Disks, Partitions, and Filesystems 96
Figure 13.1: LVM, Swap, and Disk Monitoring 102
Figure 14.1: Linux Networking Tools 109
Figure 15.1: SSH and Remote Access 115
Figure 16.1: Package Management 122
Figure 17.1: Scheduling with cron, at, and Timers 129
Figure 18.1: Logging 137
Figure 19.1: Host Monitoring 142
Figure 20.1: SSH Hardening and Firewalls 148
Figure 21.1: SELinux, AppArmor, Fail2Ban, Auditd, and PAM 157
Figure 22.1: Containers — Namespaces, cgroups, OverlayFS, and OCI 165
Figure 23.1: Troubleshooting Linux Systems 171
Figure 24.1: Production Hardening and Performance 177
Figure 25.1: Backup, Disaster Recovery, and Capacity 183

Linux Fundamentals — Distributions and Architecture
MODULE 1 · FUNDAMENTALS

Chapter 1. Linux Fundamentals — Distributions and Architecture

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/linux-fundamentals-distributions-and-architecture/

Overview

When you open a cloud virtual machine (VM), a Continuous Integration (CI) runner, or a Kubernetes worker node, you are almost always working on **Linux**. Before you change files, services, or networks, you need a clear picture of what "Linux" means, which **distribution** you are on, and how the **kernel**, **user space**, **shell**, and **terminal** fit together.

Linux started as a **kernel** — the core that manages CPU, memory, disks, and network. A usable server also needs user-space tools: a package manager, a shell, libraries, and services such as `sshd` and `systemd`. A **distribution** (often called a **distro**) packages the kernel with those tools, an installer, and a support policy. Ubuntu and Debian use `apt`. Red Hat Enterprise Linux (RHEL), Rocky Linux, and AlmaLinux use `dnf` / `yum`. Alpine uses `apk` and is common in containers. In this tutorial you will identify your distro, read kernel and user-space facts, and prove the difference between a shell and a terminal.

Cloud teams care about this because images, packages, and support windows differ by family. Mixing Alpine (musl) assumptions with Ubuntu (glibc) tools breaks builds. Treating every host as "just Linux" leads to the wrong package commands, wrong service names, and wrong runbooks. In production you pin known cloud images, document the distro family in inventory, and separate **kernel problems** (rare, often need reboot or a new image) from **user-space problems** (usually fixable with packages, configs, or service restarts).

This is **Tutorial 1** in **Module 1: Linux Fundamentals** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a small evidence pack you can use when onboarding a new VM or explaining Linux architecture in an interview.

Prerequisites

- Basic computer knowledge (files, folders, login)
- A **practice Ubuntu 22.04/24.04 VM** (or similar Linux host) with terminal access
- A normal user account; `sudo` only where the lab says so

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain what the Linux kernel is and how a distribution differs from "just the kernel"
- [ ] Identify your distro family, version, and package manager from `/etc/os-release` and commands
- [ ] Describe the layers: hardware → kernel → user space → shell/applications
- [ ] Distinguish shell, terminal emulator, and TTY/PTY with real commands
- [ ] Capture host identity evidence suitable for a change ticket or onboarding note

Architecture

Linux sits between people or automation and the hardware (or hypervisor). The kernel owns resources. User space runs tools and services. The shell is the program that reads your commands inside a terminal session.

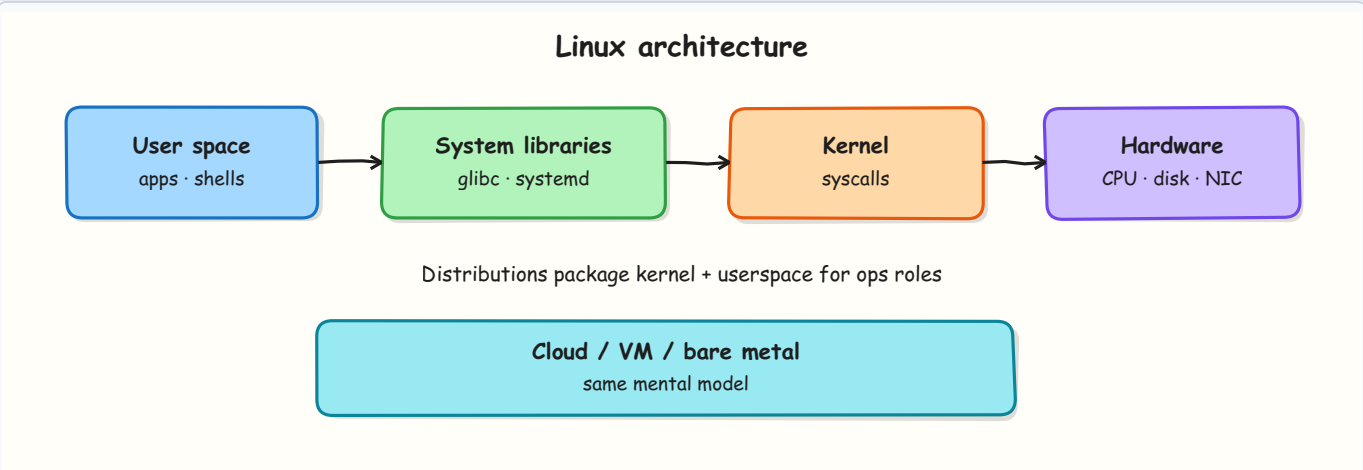

Figure 1.1: Linux Fundamentals — Distributions and Architecture

Theory

What it is

Linux is an open-source **kernel**. It schedules processes, manages memory, talks to devices through drivers, and exposes a system-call interface (`open`, `read`, `fork`, `exec`, `socket`, and many more).

A **distribution** ships:

Piece	Examples
Kernel	Version shown by `uname -r`
Init / services	Usually `systemd` on servers
Package manager	`apt`, `dnf`, `zypper`, `apk`
User-space tools	GNU coreutils, shells, libraries
Release policy	LTS vs rolling; security support window

User space is everything that is not the kernel: daemons, CLI tools, libraries under `/usr`, and your applications. The **shell** (`bash`, `zsh`, `sh`) interprets commands. The **terminal** (or terminal emulator) is the window or SSH session that shows text and sends keystrokes. The kernel connects them with a TTY or PTY device.

```
uname -r
cat /etc/os-release
echo "$SHELL"
tty
```

Why it matters

Cloud VM images, container base images, and CI runners are chosen by **distro family** and **support window**, not by fashion. Wrong package commands waste time. Wrong glibc/musl assumptions break binaries. Kernel version matters for features such as cgroup v2, eBPF, and newer filesystems. When an incident happens, your first question is often: "Is this kernel, systemd/user space, or the app?" That split decides whether you restart a service, rebuild a package, or replace the image.

How it works

1. **Hardware or hypervisor** presents CPU, RAM, disks, and NICs.
2. **Kernel** boots, loads drivers, mounts the root filesystem, and starts PID 1 (usually systemd).
3. **User space** starts services (`sshd`, `cron`, your app).
4. You connect with SSH into a **PTY**; a **shell** process reads your commands and starts child processes.
5. Those processes call the **kernel** through system calls.

```
hostnamectl          # OS pretty name, kernel, architecture
ps -p $$ -o pid,tty,comm,args
ls -l /proc/$$/exe   # which shell binary this session uses
```

Key concepts and comparisons

Layer	What lives here	Typical failure
Hardware / VM	CPU, disk, NIC	Hypervisor, quota, wrong instance type
Kernel	Drivers, memory, scheduling	Panic, OOM killer, driver bug
User space	systemd, sshd, packages	Bad config, missing package, crashed daemon
Shell / app	bash, Python, nginx	Script error, app bug

Family	Examples	Package tool	Common Cloud use
Debian	Debian, Ubuntu	`apt`	Popular cloud images, docs, CI
RHEL-like	RHEL, Rocky, Alma	`dnf` / `yum`	Enterprise, OpenShift
SUSE	SLES, openSUSE	`zypper`	Enterprise niches
Minimal	Alpine, Amazon Linux	`apk` / `dnf`	Containers, AWS-tuned hosts

Concept	Role
Terminal emulator	UI that shows text (GNOME Terminal, Windows Terminal, `tmux`)
Shell	Program that interprets commands
TTY / PTY	Kernel device for the login session

Common pitfalls

- Saying “Linux” when you mean a full distro — `apt` vs `dnf` vs `apk` are not the same.
- Choosing a distro for popularity instead of support window, image availability, and company standard.
- Debugging only in a graphical terminal and assuming cron or CI has the same environment variables.
- Treating every user-space failure as a reboot problem — many daemon issues do not need a reboot.
- Mixing Alpine (musl) containers with Ubuntu/RHEL (glibc) binaries without testing.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, identify the distribution and architecture layers, prove shell versus terminal facts, and save an identity evidence pack under `~/rebash-linux/lab01`.

Prerequisites

- Ubuntu 22.04/24.04 (or Debian) practice VM
- Packages already present: `bash`, `coreutils`, `procps`, `hostname` tooling (`hostnamectl` via `systemd`)
- Do **not** run destructive changes on a shared production server

Lab environment

Workspace: `~/rebash-linux/lab01`

```
mkdir -p ~/rebash-linux/lab01 && cd ~/rebash-linux/lab01
set -euo pipefail
whoami | tee lab-user.txt
uname -s | tee kernel-name.txt
test "$(uname -s)" = "Linux"
```

Expected output: `lab-user.txt` and `kernel-name.txt` exist; `kernel-name.txt` contains `Linux`.

Real-world scenario

Your team received a new Ubuntu cloud image for application servers. Before the first deploy, platform asks you to confirm the OS family, kernel version, architecture, default shell, and package manager — and to keep proof for the onboarding ticket. You collect facts only; you do not change packages yet.

Step-by-step tasks

Task 1 – Identify distribution and package family

```
cd ~/rebash-linux/lab01
set -euo pipefail

cat /etc/os-release | tee os-release.txt
. /etc/os-release
printf 'ID=%s\nVERSION_ID=%s\nID_LIKE=%s\n' "${ID}" "${VERSION_ID}" "${ID_LIKE:-}" | tee distro-summary.txt

if command -v apt-get >/dev/null; then
  echo "package_family=apt" | tee package-family.txt
elif command -v dnf >/dev/null; then
  echo "package_family=dnf" | tee package-family.txt
elif command -v apk >/dev/null; then
  echo "package_family=apk" | tee package-family.txt
else
  echo "package_family=unknown" | tee package-family.txt
fi

grep -E '^(ID|VERSION_ID|ID_LIKE)=' os-release.txt
test -s package-family.txt
```

Expected output: `os-release.txt` shows Ubuntu (or your distro); `package-family.txt` says `apt` on Ubuntu; `distro-summary.txt` has `ID` and `VERSION_ID`.

Task 2 – Map kernel versus user space

```
cd ~/rebash-linux/lab01
set -euo pipefail

uname -a | tee uname.txt
uname -r | tee kernel-release.txt
hostnamectl 2>/dev/null | tee hostnamectl.txt || true

# Kernel-exported facts (user space reading kernel interfaces)
head -n 5 /proc/version | tee proc-version.txt
grep -E '^(MemTotal|MemFree):' /proc/meminfo | tee meminfo-snippet.txt
nproc | tee nproc.txt

# User-space binaries that are not the kernel
command -v bash | tee bash-path.txt
command -v systemctl | tee systemctl-path.txt || echo 'no-systemctl' | tee systemctl-path.txt
ls -l "$(command -v bash)" | tee bash-ls.txt

test -s kernel-release.txt
test -s proc-version.txt
```

Expected output: `kernel-release.txt` shows a version like `6.x.x-…`; `bash-path.txt` points under `/usr` or `/bin`; `proc-version.txt` mentions Linux.

Task 3 – Prove shell versus terminal, then pack evidence

```
cd ~/rebash-linux/lab01
set -euo pipefail

echo "SHELL=$SHELL" | tee shell-env.txt
ps -p $$ -o pid=,tty=,comm=,args= | tee shell-process.txt
tty | tee tty.txt
ls -l /proc/$$/exe | tee shell-exe.txt

# Same host, different “views”: login name vs process vs device
id -un | tee id-user.txt
printf 'pid=%s tty=%s\n' "$$" "$(tty)" | tee session-map.txt

tar -czf linux-identity-evidence.tgz \
  lab-user.txt kernel-name.txt os-release.txt distro-summary.txt package-family.txt \
  uname.txt kernel-release.txt hostnamectl.txt proc-version.txt meminfo-snippet.txt nproc.txt \
  bash-path.txt systemctl-path.txt bash-ls.txt \
  shell-env.txt shell-process.txt tty.txt shell-exe.txt id-user.txt session-map.txt
ls -l linux-identity-evidence.tgz | tee evidence-ls.txt
test -s linux-identity-evidence.tgz
```

Expected output: `tty.txt` shows a pts or tty device; `shell-process.txt` shows your shell (often `bash`); evidence archive is not empty.

Validation steps

- [ ] `ID=` in `os-release.txt` matches the VM you expected
- [ ] `package-family.txt` matches the distro family

- [ ] `kernel-release.txt` and `bash-path.txt` both exist (kernel fact vs user-space binary)
- [ ] `tty.txt` and `shell-env.txt` are different kinds of facts (device vs program)
- [ ] `linux-identity-evidence.tgz` exists under `~/rebash-linux/lab01`

Common errors and fixes

Error	Cause	Fix
`hostnamectl: command not found`	Minimal image without systemd tools	Use `cat /etc/os-release` and `uname -a`; skip hostnamectl
Empty `/etc/os-release`	Unusual or container-minimal image	Check `/usr/lib/os-release` or image docs
`tty` says `not a tty`	Non-interactive script context	Run tasks in an interactive SSH/terminal session
Wrong package family guess	Custom image	Inspect `command -v apt-get dnf apk` manually

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write an executable script `~/rebash-linux/lab01/host-identity.sh` that prints four labelled lines: `distro=`, `kernel=`, `package_family=`, and `shell=`, using `/etc/os-release`, `uname -r`, package-manager detection, and `$SHELL`. Run it and save output to `host-identity-out.txt`. Keep the script as your stretch artefact.

```
# After you create the script:
chmod +x ~/rebash-linux/lab01/host-identity.sh
~/rebash-linux/lab01/host-identity.sh | tee ~/rebash-linux/lab01/host-identity-out.txt
```

Learning outcomes

- Identified distro ID, version, and package family from real files
- Separated kernel facts (`uname`, `/proc`) from user-space binaries
- Mapped shell process, `$SHELL`, and TTY device
- Built an evidence archive for onboarding or tickets

Cleanup

```
cd ~/rebash-linux/lab01
# Keep the evidence archive and challenge script if you want them for your notes.
# To remove lab text files only:
# rm -f *.txt
# rm -f linux-identity-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab01/` with evidence files
- [ ] You can explain kernel vs distribution vs user space in plain words
- [ ] You can explain shell vs terminal vs TTY
- [ ] You know why cloud teams pin a distro image instead of “latest random Linux”

Code Walkthrough

In real servers, Linux identity checks for **distributions and architecture** usually follow this order:

1. **Confirm the OS** — `/etc/os-release`, `hostnamectl`
2. **Confirm the kernel** — `uname -r`, note architecture (`uname -m`)
3. **Confirm the package family** — `apt` / `dnf` / `apk` before you install anything
4. **Confirm the session** — shell path, TTY, whether you are in SSH, `tmux`, or CI
5. **Capture evidence** — save outputs for tickets and handovers

Later modules assume you can answer “what am I on?” in under a minute.

Security Considerations

- Treat SSH and sudo on the host as privileged — know who can log in

- Do not paste secrets into shell history, tickets, or screenshots
- Prefer known, patched cloud images over unmanaged “golden” USB installs
- Record kernel and package versions when you report vulnerabilities
- Separate human admin access from application service accounts (covered in Module 4)

Common Mistakes

Calling every host ‘Linux’ without naming the distro

Package commands and paths differ. **Fix:** always read `/etc/os-release` and record `ID` + `VERSION_ID` in inventory.

Assuming the terminal program is the shell

Closing a window is not the same as understanding `bash` vs `sh`. **Fix:** check `echo $SHELL`, `ps -p $$`, and `/proc/$$/exe`.

Rebooting for every failure

User-space service failures often need `systemctl` and logs, not a reboot. **Fix:** decide kernel vs user space before you reboot.

Using Alpine container habits on Ubuntu VMs without thinking

musl vs glibc and busybox vs GNU tools behave differently. **Fix:** match base image family to the binary and docs you ship.

Best Practices

- Pin cloud image IDs (AMI / gallery image version) in Infrastructure as Code (IaC)
- Document distro family and package manager in the team runbook
- Prefer Long Term Support (LTS) server images for production fleets
- Keep a small “first five commands” list for new hosts: `os-release`, `uname -a`, `df -h`, `ip -br a`, `systemctl --failed`
- Test containers and VMs on the same family you use in production

Troubleshooting

Symptom	Likely cause	Fix
`apt-get: command not found`	RHEL-like or Alpine host	Use `dnf` or `apk`; confirm `/etc/os-release`
Binary “not found” but file exists	Wrong arch or musl/glibc mismatch	Check `uname -m` and `ldd` / image family
`hostnamectl` missing	Minimal/container image	Use `cat /etc/os-release` and `uname`
Script works in GUI terminal, fails in cron	Different shell or environment	Use absolute paths; set shebang; do not rely on interactive `$PATH`
Unclear if issue is kernel	Panic, hard lock, driver errors	Check `journalctl -k` / console; plan image or kernel update

Summary

Linux for Cloud and DevOps means a **kernel plus a distribution’s user space**. Know your distro family, package manager, kernel version, and the difference between shell and terminal — then keep proof. Next, learn how the system starts and where files live in Boot Process and Filesystem Hierarchy.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between the Linux kernel and a Linux distribution?

Answer

The **kernel** is the core that manages CPU, memory, devices, and system calls. A **distribution** packages that kernel with user-space tools, a package manager, an init system, and a support/release policy. When people say "Ubuntu server", they mean the full distribution, not only the kernel.

2. A teammate says "install with yum" on an Ubuntu cloud VM. What do you check, and what do you run instead?

Answer

Check `/etc/os-release` (and optionally `hostnamectl`). On Ubuntu/Debian the package tool is **`apt`** (`apt-get` / `apt`). `yum` / `dnf` belong to RHEL-like systems. Using the wrong tool is a signal that inventory or assumptions about the image are wrong.

3. How do shell, terminal emulator, and TTY/PTY differ?

Answer

The **terminal emulator** (or SSH client view) displays text and captures keys. The **shell** (`bash`, `zsh`, …) is the program that parses commands. **TTY/PTY** is the kernel device that connects that session to the shell process. You can see them with `tty`, `echo $SHELL`, and `ps -p $$`.

4. Why do Cloud and platform teams pin a specific image version instead of always taking "latest Ubuntu"?

Answer

Pinning keeps kernels, packages, and behaviour **reproducible** across the fleet. "Latest" can change under you and break automation. Teams record image IDs in IaC, test upgrades, then roll forward on purpose.

5. Give one production symptom that points to the kernel and one that points to user space.

Answer

Kernel: panic, hard lock-up, driver failure, sudden OOM killer behaviour tied to memory management. **User space:** `sshd` or nginx crash loop, bad unit file, missing package, wrong config — often fixed with packages/config/ `systemctl` without replacing the kernel. Always gather evidence before you reboot.

6. Why can a binary that runs on Ubuntu fail inside an Alpine container?

Answer

Ubuntu uses **glibc**; many Alpine images use **musl** and different library paths. Dynamically linked binaries and some tooling assumptions do not transfer. Teams either rebuild for the target image family or use a matching base image.

7. What facts would you put in an onboarding ticket for a new Linux VM?

Answer

At minimum: distro `ID` and `VERSION_ID`, kernel release (`uname -r`), architecture (`uname -m`), package family, hostname, and default admin access model (who has sudo). Attach command output (`os-release`, `uname -a`, `hostnamectl`) so the next engineer does not guess.

Boot Process and Filesystem Hierarchy
MODULE 1 · FUNDAMENTALS

Chapter 2. Boot Process and Filesystem Hierarchy

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/boot-process-and-filesystem-hierarchy/

Overview

When a cloud virtual machine (VM) fails to come up, hangs before Secure Shell (SSH), or boots into emergency mode, you need two maps: the **boot process** (how the machine starts) and the **Filesystem Hierarchy Standard (FHS)** (where important files live under `/`).

The **boot process** is the ordered path from power-on to a usable multi-user system: firmware (BIOS or UEFI), bootloader (usually GRUB), kernel plus **initial RAM filesystem (initramfs)**, then PID 1 (**systemd**), which reaches a default **target** such as `multi-user.target`. On many cloud images, **cloud-init** runs next to apply instance metadata, SSH keys, and hostname. The **FHS** is the conventional layout of directories so packages and operators know where configuration (`/etc`), variable data (`/var`), and runtime state (`/run`) belong. In this tutorial you will inspect boot timing with `systemd-analyze`, check the default target, and verify critical FHS paths on a practice Ubuntu VM.

Wrong layout looks like an application bug: logs filling `/` instead of `/var`, or apps writing under `/tmp` that disappear after reboot. Boot failures often sit in one stage — firmware disk selection, bad `initramfs`, broken `/etc/fstab`, or a failed unit blocking the default target. In production you separate OS disk from data volumes, persist mounts correctly, and practise rescue/emergency targets before you need them at 03:00.

This is **Tutorial 2** in **Module 1: Linux Fundamentals** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have boot and FHS evidence you can attach to an incident or change ticket.

Prerequisites

- Linux Fundamentals — Distributions and Architecture
- A **practice Ubuntu 22.04/24.04 VM** with `systemd` and sudo
- Comfort with a normal user shell (Tutorial 1)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Describe the boot chain from firmware to the default systemd target
- [ ] Use `systemd-analyze` and `systemctl get-default` to inspect boot behaviour
- [ ] Locate and explain key FHS directories (`/etc`, `/var`, `/usr`, `/home`, `/tmp`, `/run`, `/boot`)
- [ ] Check whether critical paths exist and are mounted as expected
- [ ] Capture boot and layout evidence for troubleshooting

Architecture

Boot stages hand control from firmware to the kernel, then to systemd. After the system is up, the FHS tells you where config, binaries, and variable data live on the mounted root (and other volumes).

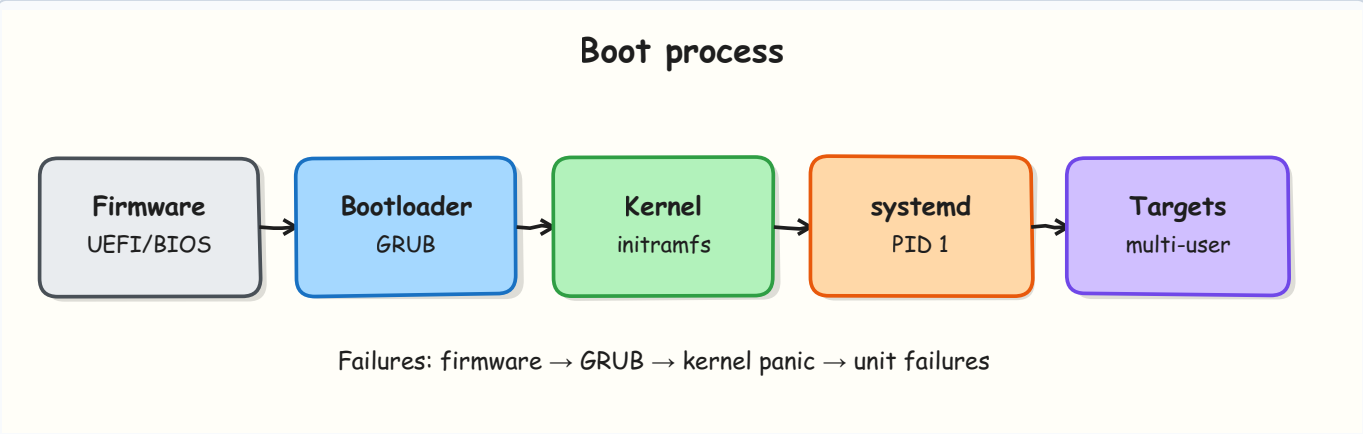

Figure 2.1: Boot Process and Filesystem Hierarchy

Theory

What it is

Boot process stages (typical systemd server):

1. **Firmware** — BIOS/UEFI initialises hardware (or the hypervisor presents virtual firmware) and chooses a boot device.
2. **Bootloader** — GRUB (common on Ubuntu) loads the kernel and initramfs.
3. **Kernel + initramfs** — early drivers and helpers; real root is mounted; PID 1 starts.
4. **systemd** — mounts filesystems, starts units in dependency order, reaches the default **target**.
5. **cloud-init** (cloud images) — applies metadata, users, SSH keys, and first-boot jobs.

FHS is the agreed directory layout under `/`. You do not need to memorise every path, but you must know the landmarks operators use every day.

```
systemctl get-default
systemd-analyze
ls -ld /etc /var /usr /home /tmp /run /boot
```

Why it matters

A VM that never accepts SSH may still be "booting" in firmware, stuck in initramfs, or waiting in emergency mode because `/etc/fstab` points at a missing disk. FHS mistakes cause capacity and data-loss incidents: writing state into `/tmp`, filling `/` with logs, or putting application data on the small OS disk. Cloud volumes must be mounted at the right path and listed so they return after reboot.

How it works

Firmware hands off to the bootloader. The bootloader loads kernel + initramfs. The kernel starts systemd. systemd reads units, mounts from `fstab` or `.mount` units, and enters the default target (`multi-user.target` on most servers; `graphical.target` on desktops). **Rescue** and **emergency** targets start fewer services so you can repair. Inspect timing with `systemd-analyze blame` and `systemd-analyze critical-chain`. Inspect mounts with `findmnt` and `/proc/mounts`.

```
systemd-analyze blame | head
findmnt -o TARGET,SOURCE,FSTYPE,OPTIONS /
ls /lib/systemd/system/multi-user.target
```

Key concepts and comparisons

Stage	Role	What breaks
Firmware	Boot device selection	Wrong disk, Secure Boot issues
Bootloader	Kernel + initramfs	Bad GRUB config, missing kernel
Kernel / initramfs	Early root, drivers	Missing module, bad root UUID
systemd target	Multi-user services	Failed unit, bad fstab
cloud-init	Instance identity	Metadata/network delay

Path	Purpose
`/boot`	Kernel, initramfs, bootloader files
`/etc`	Host-specific configuration
`/usr`	Shareable system programs and libraries
`/var`	Variable data (logs, caches, spool)
`/home`	User home directories
`/tmp`	Temporary files (often cleared on reboot)
`/run`	Early runtime data (tmpfs); not for durable storage
`/opt`	Optional add-on application software
`/proc`, `/sys`	Kernel interfaces (pseudo-filesystems)

Target idea	Prefer when
`multi-user.target`	Headless servers, most cloud VMs
`graphical.target`	Desktop with GUI
`rescue.target` / `emergency.target`	Repair with few services

Common pitfalls

- Storing application data under `/tmp` and wondering why it vanished after reboot.
- Filling `/` with logs because `/var` was never on a larger volume.
- Editing `/etc/fstab` with a typo and landing in emergency mode on next boot.
- Assuming cloud data disks auto-mount without `fstab` or a mount unit.
- Confusing “SSH is down” with “boot finished” — check console and `systemd-analyze` when you regain access.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, inspect the boot path and default target, map critical FHS directories and mounts, and save evidence under `~/rebash-linux/lab02`.

Prerequisites

- Ubuntu 22.04/24.04 with systemd
- Admin user with sudo (only where noted)
- VM console access available if you ever test rescue mode (not required for this lab)

Lab environment

Workspace: `~/rebash-linux/lab02`

```
mkdir -p ~/rebash-linux/lab02 && cd ~/rebash-linux/lab02
set -euo pipefail
```

```
test -d /etc && test -d /var
systemctl get-default | tee default-target.txt
```

Expected output: `default-target.txt` contains a target name such as `multi-user.target` or `graphical.target`.

Real-world scenario

A new Ubuntu VM takes a long time before SSH is ready. Platform asks you to measure boot time, list slow units, confirm the default target, and verify that OS landmarks (`/etc`, `/var`, `/boot`) exist and that `/` is mounted as expected — then attach proof to the ticket.

Step-by-step tasks

Task 1 – Inspect boot and default target

```
cd ~/rebash-linux/lab02
set -euo pipefail

systemctl get-default | tee default-target.txt
systemctl is-system-running | tee system-state.txt || true
systemd-analyze | tee analyze.txt
systemd-analyze blame | head -n 15 | tee blame-top.txt
systemd-analyze critical-chain 2>/dev/null | tee critical-chain.txt || true

# Recent boot messages (may need privileges on some setups)
journalctl -b -o short-iso --no-pager 2>/dev/null | head -n 40 | tee journal-boot-head.txt || \
  echo 'journal-unavailable' | tee journal-boot-head.txt

grep -E 'Startup finished|multi-user|graphical' analyze.txt || test -s analyze.txt
```

Expected output: `analyze.txt` shows total startup time; `blame-top.txt` lists units; `default-target.txt` is non-empty.

Task 2 – Map FHS landmarks and root mount

```
cd ~/rebash-linux/lab02
set -euo pipefail

FHS_PATHS="/boot /etc /usr /var /home /tmp /run /opt /proc /sys"
for p in $FHS_PATHS; do
  if [ -e "$p" ]; then
    printf 'OK\t%s\n' "$p"
  else
    printf 'MISSING\t%s\n' "$p"
  fi
done | tee fhs-check.txt

ls -ld /boot /etc /usr /var /home /tmp /run | tee fhs-ls.txt
findmnt -o TARGET,SOURCE,FSTYPE,OPTIONS / | tee root-mount.txt
findmnt -o TARGET,SOURCE,FSTYPE | tee mounts-all.txt
df -hT / /boot /var 2>/dev/null | tee df-fhs.txt || df -hT / | tee df-fhs.txt

grep -E '^OK\t/(etc|var|usr|boot)$' fhs-check.txt
test -s root-mount.txt
```

Expected output: `fhs-check.txt` shows `OK` for `/etc`, `/var`, `/usr`, `/boot`; `root-mount.txt` shows `/` with a real source and filesystem type.

Task 3 – Link boot files to FHS and pack evidence

```
cd ~/rebash-linux/lab02
set -euo pipefail

ls -l /boot | head -n 30 | tee boot-listing.txt
# Kernel/initramfs names vary; prove /boot is not empty on a normal VM
test "$(find /boot -maxdepth 1 -type f 2>/dev/null | wc -l)" -ge 1 || \
  echo 'WARN: /boot has no regular files (container or special image?)' | tee boot-warn.txt

# fstab is the persistence map for mounts (read-only inspection)
sudo test -r /etc/fstab
sudo cp /etc/fstab ./fstab.copy
wc -l fstab.copy | tee fstab-lines.txt

tar -czf boot-fhs-evidence.tgz \
  default-target.txt system-state.txt analyze.txt blame-top.txt critical-chain.txt \
  journal-boot-head.txt fhs-check.txt fhs-ls.txt root-mount.txt mounts-all.txt \
  df-fhs.txt boot-listing.txt fstab.copy fstab-lines.txt \
  $(test -f boot-warn.txt && echo boot-warn.txt || true)
ls -l boot-fhs-evidence.tgz | tee evidence-ls.txt
test -s boot-fhs-evidence.tgz
```

Expected output: `fstab.copy` exists; `boot-fhs-evidence.tgz` is not empty; `/boot` listing is captured.

Validation steps

- [ ] `systemctl get-default` matches `default-target.txt`
- [ ] `systemd-analyze` produced a startup summary in `analyze.txt`
- [ ] `fhs-check.txt` shows `OK` for `/etc` and `/var`
- [ ] `root-mount.txt` shows filesystem type for `/`
- [ ] `boot-fhs-evidence.tgz` exists under `~/rebash-linux/lab02`

Common errors and fixes

Error	Cause	Fix
`systemd-analyze: command not found`	Non-systemd environment	Use a full Ubuntu VM, not a tiny container without systemd
`Permission denied` reading journal	User not in `adm`/`systemd-journal`	Use `sudo journalctl -b …` or skip journal head
Empty `/boot` listing	Container or unusual image	Note it; FHS checks on `/etc` and `/var` still matter
`findmnt` missing	Minimal image	Install `util-linux` or use `mount | head`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Create an executable script `~/rebash-linux/lab02/fhs-audit.sh` that checks these paths exist: `/etc`, `/var`, `/usr`, `/home`, `/tmp`, `/run`, `/boot`. Print `OK` or `MISSING` per path, exit `1` if any required path is missing, and write the report to `fhs-audit-out.txt`. That script is your stretch artefact (not a markdown runbook).

Learning outcomes

- Measured boot with `systemd-analyze` and recorded the default target
- Verified FHS landmarks and the root mount
- Linked `/boot` and `/etc/fstab` to real boot/mount behaviour
- Packed evidence for a ticket

Cleanup

```
cd ~/rebash-linux/lab02
# Keep boot-fhs-evidence.tgz and fhs-audit.sh if you created them.
# rm -f *.txt fstab.copy boot-fhs-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab02/` with evidence files
- [ ] You can list boot stages from firmware to default target
- [ ] You can explain why `/etc`, `/var`, `/tmp`, and `/run` are different
- [ ] You know that a bad `fstab` entry can block a normal boot

Code Walkthrough

In real incidents, **boot and FHS** checks usually follow this order:

1. **Can you reach the console?** — cloud serial console if SSH is down
2. **Has systemd finished?** — `systemctl is-system-running`, `systemd-analyze`
3. **What is slow or failed?** — `systemd-analyze blame`, `systemctl --failed`
4. **Are disks mounted?** — `findmnt`, `df -hT`, read `/etc/fstab`
5. **Is data on the right volume?** — confirm `/var` and app paths are not silently filling `/`

Practise these on a healthy lab VM so the path is familiar under pressure.

Security Considerations

- Limit who can edit `/etc/fstab` and bootloader config — mistakes cause outages
- Protect `/boot` and bootloader passwords in environments that require them
- Do not store secrets under world-readable FHS paths
- Treat cloud-init and instance metadata as sensitive (SSH keys, user-data)
- Use rescue access procedures that are audited and rarely needed

Common Mistakes

Putting durable data in `/tmp` or `/run`

These locations are temporary. **Fix:** use `/var` or a dedicated data mount for anything that must survive reboot.

Growing logs on the OS root disk

`/var/log` can fill `/` and break the host. **Fix:** separate volume for `/var` when needed; enable logrotate; alert on `df`.

Hand-editing fstab without a test plan

A bad UUID sends the next boot to emergency mode. **Fix:** use UUIDs, keep a console path ready, test on a snapshot VM.

Ignoring cloud-init when SSH keys are wrong

First-boot identity comes from metadata. **Fix:** check cloud-init status/logs after instance launch.

Best Practices

- Prefer UUID (or LABEL) in `fstab`, not unstable device names like `/dev/sdb1`
- Separate OS and data disks on cloud VMs
- Keep `multi-user.target` as default on servers unless you need a GUI
- Record `systemd-analyze` before/after large image changes
- Document rescue/emergency login steps for your cloud provider

Troubleshooting

Symptom	Likely cause	Fix
Emergency mode on boot	Bad `fstab` / missing device	Console: remount `/` rw; fix or comment the line; `systemctl default`
Very slow boot	Slow unit or network wait	`systemd-analyze blame` / `critical-chain`; fix unit deps
SSH never ready	Service failed / cloud-init stuck	Console + `systemctl --failed` + cloud-init logs
`/tmp` empty after reboot	Expected for tmpfs/cleanup	Store durable files elsewhere
`/` full, `/home` fine	Logs or data on root	`du` under `/var`; move mounts; clean safely

Summary

Boot is a chain from firmware to the default systemd target; FHS tells you where config and data should live. Measure with `systemd-analyze`, verify mounts and landmarks, and keep evidence. Next, practise everyday file commands in Essential Linux Commands.

Interview Questions

INTERVIEW PRACTICE

1. List the main stages of a typical Linux server boot with systemd.

Answer

Firmware (BIOS/UEFI) → bootloader (for example GRUB) → kernel + initramfs → systemd (PID 1) → default target such as `multi-user.target`. On cloud images, cloud-init often runs to apply metadata and SSH keys. Interviewers want the order and what each stage is responsible for.

2. What is the difference between `rescue.target` and a normal `multi-user.target`?

Answer

`multi-user.target` is the usual multi-user, non-graphical server state with networking and services. `rescue.target` (and the tighter **emergency** mode) starts far fewer services so you can repair disks, `fstab`, or critical config. You use rescue/emergency when a normal boot cannot finish cleanly.

3. Why must operators care about the Filesystem Hierarchy Standard (FHS)?

Answer

FHS gives predictable locations: config in `/etc`, variable data in `/var`, temporary files in `/tmp`, runtime in `/run`. Apps and packages assume this layout. Putting durable data in `/tmp` or filling `/` with logs causes outages that look like "mystery application failures".

4. A VM boots to emergency mode after you added a data disk. What do you check first?

Answer

Check `/etc/fstab` for a wrong UUID, missing `nofail` / `x-systemd.device-timeout` where appropriate, or a device that is not attached. Use the serial console, remount root read-write if needed, fix or comment the bad line, then continue boot. Prefer UUID over `/dev/sdX` names.

5. How do you find which systemd units slowed down the last boot?

Answer

Run `systemd-analyze` for the total time, `systemd-analyze blame` for per-unit time, and `systemd-analyze critical-chain` for the chain that blocked reaching the default target. That trio is the standard interview answer for boot performance.

6. What belongs in `/run` versus `/var`, and what happens if you confuse them?

Answer

`/run` is early runtime state (often tmpfs) and does not survive reboot. `/var` holds variable data such as logs, caches, and spool files that must persist. Storing durable state in `/run` loses data on reboot; stuffing huge durable trees into the wrong place also fills the wrong filesystem.

7. How does cloud-init fit into the boot story on a new cloud VM?

Answer

After the OS reaches a networked multi-user state, **cloud-init** applies instance user-data and metadata: hostname, users, SSH authorised keys, and optional packages or scripts. If SSH keys are missing or first boot hangs, check cloud-init status and logs, not only `sshd` alone.

Essential Linux Commands
MODULE 2 · COMMAND LINE

Chapter 3. Essential Linux Commands

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/essential-linux-commands/

Overview

Every Secure Shell (SSH) session starts with the same basics: where am I, what files exist, how do I create or move them safely, and how do I read content without breaking a production host. These **essential Linux commands** are the tools you will use every day in Cloud, DevOps, and Site Reliability Engineering (SRE) work.

Navigation commands (`pwd`, `ls`, `cd`) tell you your place in the tree. Mutation commands (`mkdir`, `touch`, `cp`, `mv`, `rm`) create and change files. Viewing commands (`cat`, `less`, `head`, `tail`) show content. Inspection commands (`stat`, `file`, `history`) explain metadata and what you ran before. Redirection (`>`, `>>`) and pipes (`|`) combine small tools into short workflows. In this tutorial you will build a real directory tree, copy and move files, inspect them, and save proof under `~/rebash-linux/lab03`.

Incidents go wrong when people guess paths or run destructive `rm -rf` patterns without checking. Operators who can page logs with `less`, follow growth with `tail -f`, and confirm type with `file` / `stat` move faster and with less risk. The same commands appear inside containers and Continuous Integration (CI) runners, so this fluency transfers across hosts.

This is **Tutorial 3** in **Module 2: Command Line Essentials** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, SRE, and platform engineers. By the end, you will have a small project tree and an evidence archive you can show as CLI practice.

Prerequisites

- Boot Process and Filesystem Hierarchy
- A **practice Ubuntu 22.04/24.04 VM** with a normal user account
- Willingness to create and delete files only under your lab directory

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Navigate with `pwd`, `ls`, and `cd` using absolute and relative paths
- [ ] Create, copy, move, and remove files and directories safely
- [ ] View file content with `cat`, `head`, `tail`, and `less` (when appropriate)
- [ ] Inspect metadata with `stat` and `file`, and use shell `history`
- [ ] Build a small directory tree and prove each operation with saved output

Architecture

The shell keeps a **current working directory**. Commands resolve relative paths against it. File operations create or change directory entries; viewers and inspectors read content and metadata without needing a graphical file manager.

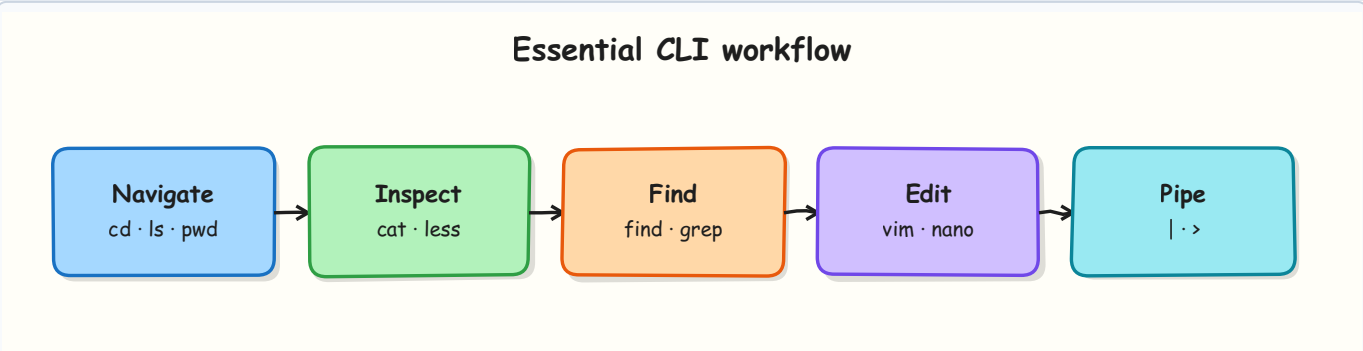

Figure 3.1: Essential Linux Commands

Theory

What it is

Need	Commands
Where am I / list / move	`pwd`, `ls`, `cd`
Create empty file / dirs	`touch`, `mkdir`
Copy / rename-move / delete	`cp`, `mv`, `rm`
View content	`cat`, `less`, `head`, `tail`
Inspect	`stat`, `file`, `history`

```
pwd
ls -la
mkdir -p project/src
touch project/README.md
```

Why it matters

Automation, deploys, and incident response all assume you can navigate and change files without surprises. A wrong `rm` or `mv` on a production path causes outages. Clear use of `cp -a`, careful `rm`, and checking with `ls` / `stat` before destructive steps is professional baseline skill — not “beginner only”.

How it works

1. The shell stores the current directory; `pwd` prints it.
2. `ls` lists names; `-a` includes hidden (dot) files; `-l` long format; `-h` human sizes with `-l`.
3. `cd` changes directory; `cd -` returns to the previous directory; `cd` alone goes to `$HOME`.
4. `mkdir -p` creates parent paths; `touch` creates empty files or updates timestamps.
5. `cp` copies; `cp -a` archives (mode/timestamps/links as documented); `mv` renames or relocates; `rm` unlinks names (`-r` for directories — dangerous).
6. `cat` dumps a whole file; `less` pages; `head` / `tail` show ends (`tail -f` follows).
7. `stat` shows inode metadata; `file` guesses content type; `history` lists past commands.

```
cp -a src dest
mv oldname newname
head -n 20 app.log
stat -c '%n %s %A' app.log
file app.log
```

Key concepts and comparisons

Need	Prefer	Avoid
Whole small file	`cat`	`cat` on multi-gigabyte logs
Browse / search large file	`less`	Dumping huge files to SSH
First or last lines	`head` / `tail`	Opening binary files as text
Preserve mode/timestamps	`cp -a`	Blind `cp` when attributes matter
Delete directory tree	`rm -r` only after `ls`	`rm -rf /` patterns and unquoted globs

Redirection	Meaning
`>`	Overwrite stdout to a file
`>>`	Append stdout
`2>`	Redirect stderr
`|`	Pipe stdout to next command

Common pitfalls

- Running `rm -rf` with a variable that expands empty (deletes unexpected paths).
- Using `cat file | less` instead of `less file`.
- Forgetting that `mv` across filesystems copies-then-deletes (attributes/space matter).
- Editing production files in place with no backup copy.
- Trusting `history` for secrets — never put passwords on the command line.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Build a real mini project tree with copy/move/view/inspect operations, prove each step with files under `~/rebash-linux/lab03`, and package evidence. This lab is **file operations**, not a generic host baseline.

Prerequisites

- Ubuntu 22.04/24.04 practice VM
- Write access to your home directory
- No sudo required for the main tasks

Lab environment

Workspace: `~/rebash-linux/lab03`

```
mkdir -p ~/rebash-linux/lab03 && cd ~/rebash-linux/lab03
set -euo pipefail
pwd | tee pwd-start.txt
rm -rf demo-app
```

Expected output: `pwd-start.txt` ends with `lab03`; clean slate for `demo-app`.

Real-world scenario

You are preparing a small application directory on a jump host: README, config sample, logs folder, and a scripts folder. You must create the layout, populate files, reorganise with `mv` / `cp`, and leave proof for a teammate who will automate the same layout later.

Step-by-step tasks

Task 1 – Create a project tree and files

```
cd ~/rebash-linux/lab03
set -euo pipefail

mkdir -p demo-app/{src,scripts,logs,config}
```

```
touch demo-app/README.md
touch demo-app/src/main.sh demo-app/scripts/run.sh
touch demo-app/config/app.env.sample

cat > demo-app/README.md << 'EOF'
# demo-app
REBASH lab sample application tree.
EOF

cat > demo-app/src/main.sh << 'EOF'
#!/usr/bin/env bash
echo "demo-app start"
EOF

cat > demo-app/logs/app.log << 'EOF'
INFO start
INFO ready
WARN retry
INFO ok
EOF

find demo-app -print | sort | tee tree-created.txt
test -f demo-app/README.md
test -d demo-app/logs
```

Expected output: `tree-created.txt` lists `demo-app` paths including `README.md`, `src/main.sh`, and `logs/app.log`.

Task 2 – Copy, move, view, and inspect

```
cd ~/rebash-linux/lab03
set -euo pipefail

cp -a demo-app/config/app.env.sample demo-app/config/app.env
cp demo-app/logs/app.log demo-app/logs/app.log.bak
mv demo-app/scripts/run.sh demo-app/scripts/start.sh

# Viewing (small files — safe to cat)
cat demo-app/README.md | tee readme-cat.txt
head -n 2 demo-app/logs/app.log | tee log-head.txt
tail -n 2 demo-app/logs/app.log | tee log-tail.txt

# Metadata
stat demo-app/src/main.sh | tee stat-main.txt
stat -c '%n size=%s mode=%A' demo-app/src/main.sh | tee stat-short.txt
file demo-app/README.md demo-app/src/main.sh demo-app/logs/app.log | tee file-types.txt

ls -laR demo-app | tee ls-after-ops.txt
test -f demo-app/scripts/start.sh
test ! -e demo-app/scripts/run.sh
grep -q 'demo-app' readme-cat.txt
```

Expected output: `run.sh` is gone and `start.sh` exists; `file-types.txt` shows text types; `stat-short.txt` has size and mode.

Task 3 – Safe delete proof, history snippet, evidence pack

```
cd ~/rebash-linux/lab03
set -euo pipefail

# Create a disposable file, then remove it deliberately (not rm -rf /)
echo 'temp' > demo-app/logs/temp.scratch
test -f demo-app/logs/temp.scratch
rm -f demo-app/logs/temp.scratch
test ! -e demo-app/logs/temp.scratch
echo 'removed temp.scratch OK' | tee rm-proof.txt

# history may be empty in some non-interactive shells — still capture what we can
history 20 2>/dev/null | tee history-snip.txt || \
  fc -l -20 2>/dev/null | tee history-snip.txt || \
  echo 'history-unavailable-in-this-shell' | tee history-snip.txt

# less is interactive — demonstrate non-interactive page count instead with wc
wc -l demo-app/logs/app.log | tee log-wc.txt

tar -czf cli-fileops-evidence.tgz \
  pwd-start.txt tree-created.txt readme-cat.txt log-head.txt log-tail.txt \
  stat-main.txt stat-short.txt file-types.txt ls-after-ops.txt \
  rm-proof.txt history-snip.txt log-wc.txt demo-app
ls -l cli-fileops-evidence.tgz | tee evidence-ls.txt
test -s cli-fileops-evidence.tgz
```

Expected output: `rm-proof.txt` confirms scratch removal; archive includes the `demo-app` tree and is non-empty.

Validation steps

- [ ] `demo-app/` contains `src`, `scripts`, `logs`, `config`
- [ ] `scripts/start.sh` exists (renamed from `run.sh`)
- [ ] `config/app.env` is a copy of the sample
- [ ] `stat-short.txt` and `file-types.txt` exist
- [ ] `cli-fileops-evidence.tgz` exists under `~/rebash-linux/lab03`

Common errors and fixes

Error	Cause	Fix
`mkdir: cannot create directory`	Parent missing without `-p`	Use `mkdir -p`
`mv: cannot stat`	Wrong path	`ls` the parent; fix spelling
Accidental overwrite with `cp`	Destination existed	Use distinct names; consider `cp -i` while learning
`history` empty	Non-interactive shell	Run in interactive bash; note unavailable in evidence

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write an executable script `~/rebash-linux/lab03/organise-logs.sh` that:

1. Creates `demo-app/logs/archive/` if needed
2. Copies `demo-app/logs/app.log` to `demo-app/logs/archive/app-$(date +%Y%m%d).log`
3. Appends one line `INFO archived-by-organise-logs` to `demo-app/logs/app.log`
4. Prints the archived file path

Run it once and keep the script as your stretch artefact.

Learning outcomes

- Created a real directory tree with `mkdir` and `touch`
- Used `cp`, `mv`, and careful `rm` with proof
- Viewed and inspected files with `cat` / `head` / `tail` / `stat` / `file`
- Packaged a CLI evidence archive

Cleanup

```
cd ~/rebash-linux/lab03
# Keep cli-fileops-evidence.tgz if you want it.
# rm -rf demo-app *.txt cli-fileops-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab03/` with a real `demo-app` tree
- [ ] You can explain when to use `cat` vs `less` vs `tail`
- [ ] You know why `rm -rf` needs extreme care
- [ ] You can inspect a file with `stat` and `file` before changing it

Code Walkthrough

In production, everyday file work usually follows this order:

1. **Orient** — `pwd`, `ls -la`
2. **Read before write** — `head` / `less` / `stat`
3. **Change with copies** — `cp` backup, then edit or `mv`
4. **Prove** — `ls`, `stat`, or a checksum when it matters
5. **Avoid secrets on the CLI** — do not put passwords in `history`

Scripts should use `set -euo pipefail`, quote variables, and prefer `--` before path arguments when names can start with `-`.

Security Considerations

- Never run destructive `rm` patterns from untrusted input
- Do not put tokens or passwords on the command line (visible in `history` and process lists)
- Prefer editing copies of production config, then replace deliberately
- Be careful with shell globs (`*`) in directories that contain unexpected names
- Restrict write access to deploy directories (permissions covered in Module 4)

Common Mistakes

Using `rm -rf` without listing first

Typos delete the wrong tree. **Fix:** `ls` the path, prefer trash/backup on shared hosts, quote variables, avoid root-level globs.

`cat` on huge log files over SSH

Floods your terminal and session. **Fix:** use `less`, `head`, `tail`, or remote grep tools.

Assuming `cp` keeps permissions you care about

Default `cp` may not preserve everything you expect. **Fix:** use `cp -a` (or explicit `--preserve`) when modes/timestamps matter.

Relying on `history` as documentation

History is local and may contain secrets. **Fix:** keep deliberate evidence files or scripts in git — not password-bearing history lines.

Best Practices

- Prefer `mkdir -p` and explicit paths in scripts
- Use `cp -a` for config backups before edits
- Page with `less`; follow with `tail -f` during live incidents
- Check type with `file` before treating a file as text
- Keep lab and production data in clearly named directories

Troubleshooting

Symptom	Likely cause	Fix
`No such file or directory`	Wrong cwd or path	`pwd`; use absolute path
`Permission denied`	Mode/owner	Check `ls -l`; fix later with chmod/chown (Module 4)
`Is a directory` on `cat`	Path is a directory	`ls` it; cat a file inside
`cp` overwrote a file	Destination existed	Restore from backup; use unique names
Binary garbage in terminal	Viewed binary with `cat`	Use `file` first; use `less` / `xxd` carefully

Summary

Essential commands are how you navigate, change, view, and inspect files on every Linux host. Practise safe file operations with proof — not only memorising names. Next, deepen the filesystem model in Filesystem Paths, Links, Mounts, and Inodes.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between `pwd` and `ls`, and when do you need both?

Answer

`pwd` prints the current working directory path. `ls` lists entries in a directory. You need both when a relative path fails: confirm where you are (`pwd`), then see what names exist (`ls`) before you `cd` or run a destructive command.

2. When do you use `cat`, `less`, `head`, and `tail`?

Answer

Use `cat` for small files you want entirely. Use `less` to browse or search large text. Use `head` / `**tail**` for the start or end; `tail -f` follows a growing log during an incident. Avoid `cat` on huge logs over SSH.

3. Why is `rm -rf` dangerous in scripts, and how do you reduce risk?

Answer

A wrong variable or unquoted path can delete far more than intended. Reduce risk with `set -u`, quote variables, assert paths (`[[ -d $dir ]]`), avoid running as root for app cleanup, and never build paths like `"$ROOT/$REL"` when `$REL` can be empty without checks.

4. What does `cp -a` give you that plain `cp` may not?

Answer

`cp -a` (archive) aims to preserve permissions, timestamps, and other attributes (and copies directories recursively). Plain `cp` is fine for simple content copies but may not keep the metadata you need for config backups or deploy artefacts.

5. How do `stat` and `file` help before you edit a "config" file?

Answer

`stat` shows size, mode, owner, and timestamps (inode metadata). `file` guesses whether content is text, binary, or empty. Together they stop you from treating a binary or a directory like a text config, and they document state before a change.

6. A junior engineer runs `mv /var/log/app.log /tmp/` on a production host to "make space". What can go wrong?

Answer

The app may still write to the old path or reopen the file; `/tmp` may be emptied on reboot; you may break log shipping that expects `/var/log`. Better: rotate/compress logs, fix retention, or expand the volume — and coordinate with the service owner.

7. How would you prove in a ticket that you reorganised a directory safely?

Answer

Save `find` / `ls -laR` before and after, show the exact `mv` / `cp` commands, keep a backup copy (`cp -a`), and attach a small archive or listing. Proof is listings and hashes when needed — not "I think it worked".

Filesystem Paths, Links, Mounts, and Inodes
MODULE 3 · FILESYSTEM

Chapter 4. Filesystem Paths, Links, Mounts, and Inodes

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/filesystem-paths-links-mounts-and-inodes/

Overview

Broken symbolic links after a deploy, “disk full” with free space still showing, and bind mounts that hide a directory under the same path are everyday operations problems. They all come from how Linux names files: **paths**, **inodes**, **links**, and **mounts**.

Linux presents storage as a **single tree** starting at `/`. A **path** names a place in that tree. An **inode** holds metadata and points to data blocks. Directory entries map names to inode numbers. A **hard link** is another name for the same inode on one filesystem. A **symbolic link (symlink)** stores a path string and can cross filesystems. A **mount** attaches another filesystem onto a directory (mount point). In this tutorial you will create hard links and symlinks, inspect inode numbers, list mounts with `findmnt`, and save proof under `~/rebash-linux/lab04`.

Scripts that use relative paths fail under cron when the working directory changes. Symlinks that point at versioned release directories are a common deploy pattern — until someone deletes the target. Inode exhaustion can deny new files while `df -h` still shows space. In production you prefer absolute paths in automation, check mounts after attaching cloud disks, and monitor both space and inodes.

This is **Tutorial 4** in **Module 3: Linux Filesystem** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a small “release symlink” layout you can explain in an interview.

Prerequisites

- Essential Linux Commands
- A **practice Ubuntu 22.04/24.04 VM** with write access under your home directory
- Ability to run `findmnt` (package `util-linux`, present on Ubuntu)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain absolute paths, relative paths, and why cron jobs prefer absolute paths
- [ ] Show inode numbers with `ls -i` and relate them to hard links
- [ ] Create and resolve symbolic links with `ln -s` and `readlink -f`
- [ ] List mounts with `findmnt` and explain what a mount point is
- [ ] Build a versioned app directory with a `current` symlink as deploy-style practice

Architecture

Names in directories point to inodes. Hard links share one inode. Symlinks store a path. Mounts graft other filesystems into the tree so disks and pseudo-filesystems appear as ordinary paths.

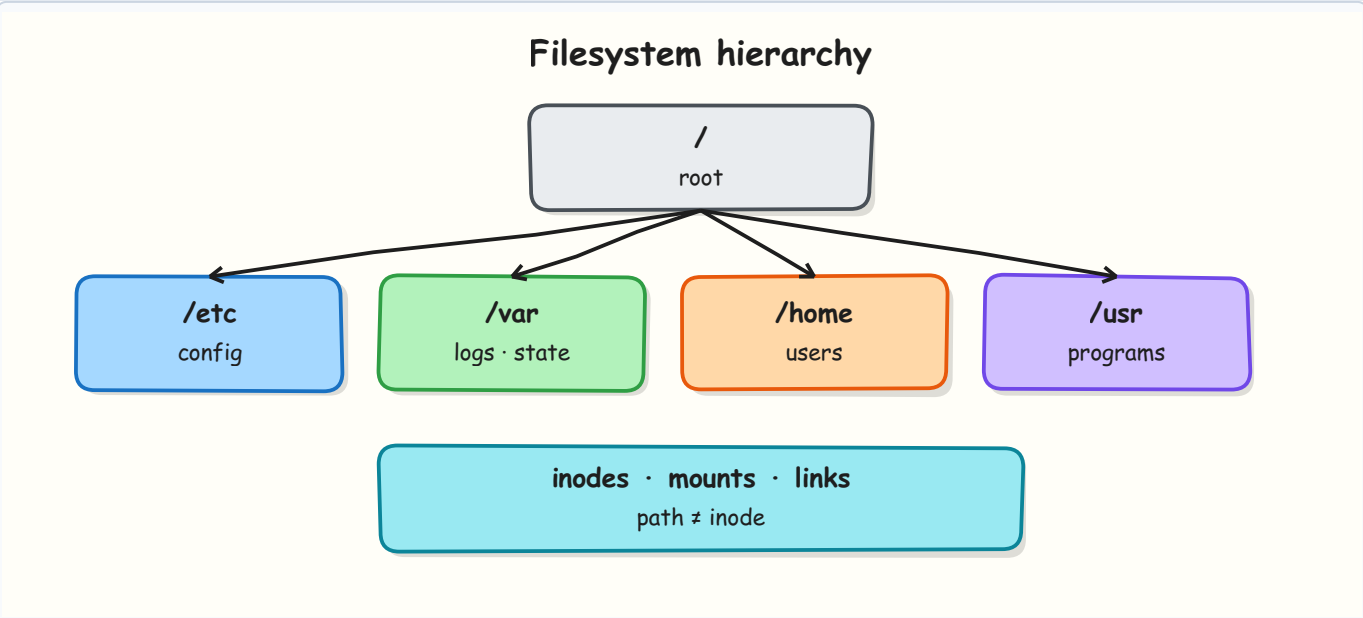

Figure 4.1: Filesystem Paths, Links, Mounts, and Inodes

Theory

What it is

Idea	Meaning
Absolute path	Starts at `/` (example: `/home/ubuntu/app`)
Relative path	Depends on current directory (example: `../logs`)
Inode	Metadata + data pointers; identified by number on a filesystem
Hard link	Extra name for the same inode (same filesystem only)
Symlink	Special file holding a path string
Mount point	Directory where another filesystem is attached

```
ls -li
ln file hardlink
ln -s /etc/hosts hosts.link
readlink -f hosts.link
findmnt /
```

Why it matters

Deploy tools often create `current` → `releases/2026-08-02`. If the symlink breaks, the site goes down even though files still exist. Bind mounts can hide old files under a path until unmount — confusing backups and deletes. Inode exhaustion appears as `No space left on device` while `df -h` looks fine; you need `df -i`. Understanding paths prevents "works in my SSH session, fails in systemd/cron" bugs.

How it works

1. Creating a file allocates an inode and one directory entry (link count 1).
2. `ln` without `-s` adds another name; `ls -i` shows the same inode; data remains until link count is 0 and no process holds it open.
3. `ln -s target name` creates a symlink; if the target is missing, the link is **dangling**.
4. `readlink` / `readlink -f` show or fully resolve the target.
5. `mount` / `findmnt` show how devices and bind mounts attach to directories; persist with `/etc/fstab` or systemd `.mount` units (by UUID preferred).

```
echo hello > a.txt
ln a.txt b.txt
ls -li a.txt b.txt
ln -s a.txt a.sym
findmnt -T .
```

Key concepts and comparisons

Type	Same inode?	Cross filesystem?	Typical use
Hard link	Yes	No	Extra name for a file
Symlink	No (stores path)	Yes	Stable path to a versioned target

Path style	Prefer when	Risk
Absolute	cron, systemd, scripts	Longer; must stay valid if homes move
Relative	Interactive short work	Breaks when cwd changes

Tool	Shows
`ls -i`	Inode numbers
`stat`	Inode, links, size, mode
`findmnt`	Mount table (readable)
`df -i`	Inode capacity per mount

Common pitfalls

- Creating a symlink with a relative target and then moving the link file.
- Using hard links for directories (normally not allowed) or across mounts (fails).
- Assuming deleting one hard-linked name deletes the data immediately (other names remain).
- Forgetting that a mount can hide previous contents of a directory until unmount.
- Ignoring inode limits on filesystems with millions of tiny files.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, prove path types, hard links, symlinks, and mounts with a small versioned “release” layout under `~/rebash-linux/lab04`, and pack evidence.

Prerequisites

- Ubuntu 22.04/24.04
- Commands: `ln`, `readlink`, `findmnt`, `stat`, `df`
- Work only under your home lab path

Lab environment

Workspace: `~/rebash-linux/lab04`

```
mkdir -p ~/rebash-linux/lab04 && cd ~/rebash-linux/lab04
set -euo pipefail
rm -rf pathlab
mkdir -p pathlab && cd pathlab
pwd | tee ../pwd-pathlab.txt
```

Expected output: `pwd-pathlab.txt` ends with `pathlab`.

Real-world scenario

Your team deploys an app by unpacking a release directory and flipping a `current` symlink. You must practise that pattern, prove inode behaviour with a hard link, and document mounts for the lab filesystem so on-call knows how to read `findmnt` during a disk incident.

Step-by-step tasks

Task 1 – Absolute vs relative paths

```
cd ~/rebash-linux/lab04/pathlab
set -euo pipefail
```

```
mkdir -p releases/v1
echo 'v1 app' > releases/v1/app.txt

# Relative path from pathlab
cat releases/v1/app.txt | tee ../read-relative.txt

# Absolute path
ABS="$(pwd)/releases/v1/app.txt"
echo "$ABS" | tee ../abs-path.txt
cat "$ABS" | tee ../read-absolute.txt

# Show that relative fails if cwd is wrong
cd /tmp
if cat releases/v1/app.txt 2>~/rebash-linux/lab04/relative-fail.txt; then
  echo 'ERROR: relative path should fail from /tmp' >&2
  exit 1
fi
cd ~/rebash-linux/lab04/pathlab
grep -Ei 'No such file|cannot' ~/rebash-linux/lab04/relative-fail.txt
test -s ~/rebash-linux/lab04/abs-path.txt
```

Expected output: relative and absolute reads succeed from `pathlab`; from `/tmp` the relative read fails and is recorded in `relative-fail.txt`.

Task 2 – Inodes, hard links, and symlinks

```
cd ~/rebash-linux/lab04/pathlab
set -euo pipefail

echo 'payload' > releases/v1/data.bin
ln releases/v1/data.bin releases/v1/data-hard.bin
ln -s data.bin releases/v1/data.sym
ln -s "$(pwd)/releases/v1" current-release

ls -li releases/v1/data.bin releases/v1/data-hard.bin | tee ../inode-hardlinks.txt
# Same inode number on both hard-linked names
INO1=$(stat -c '%i' releases/v1/data.bin)
INO2=$(stat -c '%i' releases/v1/data-hard.bin)
test "$INO1" = "$INO2"
stat -c 'links=%h inode=%i name=%n' releases/v1/data.bin | tee ../stat-hard.txt

readlink releases/v1/data.sym | tee ../symlink-target.txt
readlink -f current-release | tee ../current-resolved.txt
ls -l current-release | tee ../current-ls.txt

test "$(readlink releases/v1/data.sym)" = "data.bin"
grep -q 'payload' releases/v1/data-hard.bin
```

Expected output: `inode-hardlinks.txt` shows the same inode for both hard links; `stat-hard.txt` shows link count ≥ 2; `current-resolved.txt` points at `…/releases/v1`.

Task 3 – Mounts, inode capacity, evidence pack

```
cd ~/rebash-linux/lab04
set -euo pipefail

findmnt -T "$HOME" | tee findmnt-home.txt
findmnt -o TARGET,SOURCE,FSTYPE,OPTIONS / | tee findmnt-root.txt
df -hT . | tee df-pathlab.txt
df -i . | tee dfi-pathlab.txt

# Dangling symlink demo (safe, inside lab)
ln -sf /no/such/target pathlab/dangling.sym
readlink pathlab/dangling.sym | tee dangling-target.txt
if [ -e pathlab/dangling.sym ]; then
  # symlink exists as a name; target may not
  ls -l pathlab/dangling.sym | tee dangling-ls.txt
fi
test ! -e /no/such/target

tar -czf paths-links-evidence.tgz \
  pwd-pathlab.txt read-relative.txt abs-path.txt read-absolute.txt relative-fail.txt \
  inode-hardlinks.txt stat-hard.txt symlink-target.txt current-resolved.txt current-ls.txt \
  findmnt-home.txt findmnt-root.txt df-pathlab.txt dfi-pathlab.txt \
  dangling-target.txt dangling-ls.txt pathlab
ls -l paths-links-evidence.tgz | tee evidence-ls.txt
test -s paths-links-evidence.tgz
```

Expected output: `findmnt-home.txt` shows the mount that holds your home; `dfi-pathlab.txt` shows inode use; evidence archive is non-empty.

Validation steps

- [ ] Absolute read works; relative read from `/tmp` fails as shown
- [ ] Hard links share one inode (`test` in Task 2 passed)
- [ ] `current-release` resolves with `readlink -f`
- [ ] `findmnt` and `df -i` outputs exist
- [ ] `paths-links-evidence.tgz` exists under `~/rebash-linux/lab04`

Common errors and fixes

Error	Cause	Fix
`ln: failed to create hard link: Invalid cross-device link`	Target on another mount	Use a symlink, or keep both names on one filesystem
Symlink points at wrong place	Relative target + moved link	Prefer absolute targets for system links, or careful relative layout
`findmnt: command not found`	Minimal image	Install `util-linux`
Confused by dangling symlink	Target deleted	Recreate target or fix link with `ln -sfn`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Create `pathlab/releases/v2/app.txt` with content `v2 app`, then atomically repoint `pathlab/current-release` to `releases/v2` using `ln -sfn` (or `ln -s` + `mv -T` pattern). Prove with `readlink -f pathlab/current-release` and save `current-v2.txt`. Keep the v1 and v2 directories as your deploy-style artefact.

Learning outcomes

- Proved why absolute paths matter when cwd changes
- Created hard links and symlinks and inspected inodes
- Listed mounts and inode capacity for the lab path
- Practised a `current` → release directory layout

Cleanup

```
cd ~/rebash-linux/lab04
# rm -rf pathlab *.txt paths-links-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab04/` with evidence files
- [ ] You can explain inode vs filename vs symlink
- [ ] You can explain hard link vs symlink trade-offs
- [ ] You know mounts can hide directory contents and that `df -i` matters

Code Walkthrough

In production, filesystem identity checks usually follow this order:

1. **Resolve the path** — `pwd`, absolute paths in scripts
2. **See what the name is** — `ls -l` (symlink?) + `readlink -f`
3. **See the inode** — `ls -i`, `stat`
4. **See the mount** — `findmnt -T path`, `df -hT`, `df -i`
5. **Change carefully** — `ln -sfn` for flips; never delete the only copy of data blindly

Deploy symlink flips should be atomic (`ln -sfn` or `mv -T` of a prepared link).

Security Considerations

- Symlinks in world-writable directories can be abused (symlink races) — be careful in `/tmp`
- Do not follow untrusted symlinks in privileged scripts without checks

- Mount options (`nosuid`, `nodev`, `noexec`) matter on user-writable volumes
- Hide mounts and bind mounts can confuse audits — document them
- Restrict who can `mount` / edit `fstab`

Common Mistakes

Using relative paths in cron or systemd units

Working directory is not your SSH home. **Fix:** use absolute paths; set `WorkingDirectory=` deliberately in units.

Deleting a symlink target and leaving a dangling link

Services keep failing with "No such file". **Fix:** update the symlink when you remove old releases; monitor with deploy checks.

Assuming `df -h` is enough

Inodes can be exhausted. **Fix:** always check `df -i` in capacity incidents.

Creating hard links across mounts

The kernel rejects cross-device hard links. **Fix:** use symlinks for cross-filesystem references.

Best Practices

- Prefer absolute paths in automation
- Use versioned directories + `current` symlink for releases
- Document bind mounts and extra disks in the runbook
- Monitor disk space **and** inodes
- Clean old releases with a policy so disks and inodes stay healthy

Troubleshooting

Symptom	Likely cause	Fix
`No such file` but `ls` shows a symlink	Dangling symlink	`readlink -f`; restore target or fix link
`Invalid cross-device link`	Hard link across mounts	Use `ln -s`
Files vanished under a directory	Mount covered the path	`findmnt`; unmount or use the correct mount point
Cannot create files; `df -h` OK	Inode exhaustion	`df -i`; delete tiny-file trees; raise limits if appropriate
cron script fails	Relative paths	Switch to absolute paths

Summary

Paths name locations; inodes hold the real file; hard links share inodes; symlinks store paths; mounts attach filesystems into the tree. Practise deploy-style symlinks and always check mounts and inodes during capacity issues. Next, measure usage in Disk Usage and File Attributes.

Interview Questions

INTERVIEW PRACTICE

1. What is an inode, and what does a filename have to do with it?

Answer

An **inode** stores file metadata and pointers to data on a filesystem. A **filename** is a directory entry that points to an inode number. Several names (hard links) can point to the same inode. Deleting a name reduces the link count; data can remain until the last link is gone and no process has the file open.

2. Compare hard links and symbolic links.

Answer

Hard links share the same inode, must stay on one filesystem, and the file survives until all hard names are removed. **Symlinks** store a path string, can cross filesystems, and can dangle if the target is missing. Deploys often use symlinks for a stable `current` name.

3. Why do cron jobs often break with relative paths that worked over SSH?

Answer

Interactive SSH sessions start in a home directory; cron's working directory is usually different (often `/` or a service home). Relative paths resolve against that cwd and fail. Use **absolute paths** (and explicit `cd` only when intentional).

4. How can a directory look empty after you mount a disk on it?

Answer

Mounting attaches a filesystem **over** the mount-point directory. Previous files under that directory are hidden until unmount (they still exist on the underlying filesystem). Operators sometimes "lose" files this way. Check with `findmnt` before and after attaching disks.

5. `df -h` shows free space but creating a file fails with "No space left on device". What else do you check?

Answer

Check **`df -i`** for inode exhaustion. Also check whether you are on the mount you think (`findmnt -T .`), and whether quotas apply. Millions of tiny files can exhaust inodes while byte space remains.

6. How would you flip a release symlink safely in production?

Answer

Prepare `releases/new`, then repoint atomically with **`ln -sfn releases/new current`** (or create a new link name and `mv -T` it over `current`). Verify with `readlink -f`. Keep previous releases until health checks pass so you can roll back by flipping again.

7. When would you choose a hard link over a symlink?

Answer

Choose a **hard link** when you want two names for the same file data on one filesystem without depending on a path string (and you accept that tools must understand link counts). Choose a **symlink** for cross-filesystem pointers, versioned release directories, and optional targets. Hard links to directories are not a normal operator tool.

Disk Usage and File Attributes
MODULE 3 · FILESYSTEM

Chapter 5. Disk Usage and File Attributes

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/disk-usage-and-file-attributes/

Overview

When a server cannot write files, you need two answers quickly: **which filesystem is full**, and **which directory tree used the space**. `df` answers the first (free space and free inodes per mount). `du` answers the second (space used under a path). **File attributes** are the metadata you see with `ls -l` and `stat`: mode, owner, size, and timestamps.

On cloud virtual machines (VMs), container hosts, and Continuous Integration (CI) runners, disks fill from logs, package caches, and image layers. Sometimes `df` shows full while `du` on `/` looks smaller — often a deleted file is still held open by a process, or another mount (for example `/var`) is the one that filled. In this tutorial you will create a sample tree, measure it with `du`, compare mounts with `df`, inspect attributes with `stat`, and save proof under `~/rebash-linux/lab05`.

In production, Site Reliability Engineering (SRE) runbooks almost always start with `df -h`, `df -i`, then `du` on the hot mount. Attribute checks separate "disk full" from "permission denied" or "wrong owner". Extended flags such as `chattr +i` (immutable) can protect critical configs — and can also block deploys if someone forgets to clear them.

This is **Tutorial 5** in **Module 3: Linux Filesystem** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, SREs, and platform engineers.

Prerequisites

- Paths, Links, Mounts, and Inodes
- A **practice Ubuntu 22.04/24.04 VM** (or similar) with write access under your home directory
- Optional: `sudo` if you want to inspect deleted-open files system-wide later

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain the difference between `df` (per mount) and `du` (per directory tree)
- [ ] Check both block space and inode capacity with `df -h` and `df -i`
- [ ] Find large directories with `du` and sort human-readable sizes
- [ ] Read mode, owner, size, and timestamps with `stat`
- [ ] Pack evidence under `~/rebash-linux/lab05` for a capacity ticket

Architecture

Disk capacity is per **mount**. File attributes live on the **inode**. Tools report different layers of the same storage story.

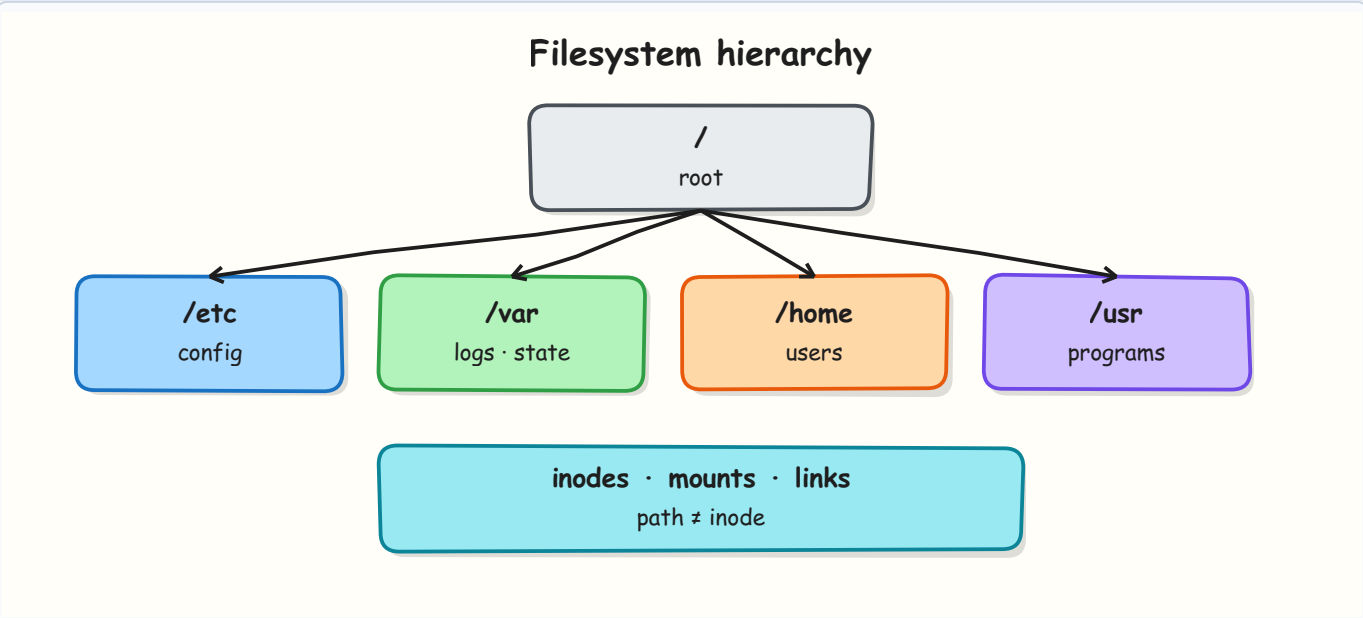

Figure 5.1: Disk Usage and File Attributes

Theory

What it is

File attributes are metadata: permissions, owner, group, size, timestamps (atime/mtime/ctime), inode number, and device. Extended attributes (xattrs) and filesystem flags (`chattr` / `lsattr` on ext4) add extra behaviour such as immutability.

Disk usage has two views:

Tool	Answers
`df`	How much free space / free inodes does this **mount** have?
`du`	How much space does this **directory tree** use?
`stat` / `ls -l`	Who owns it, what mode, what size and times?

```
df -hT
df -i
du -sh ~
stat /etc/passwd
```

Why it matters

Disk-full incidents are among the most common production pages. Container hosts fill `/var/lib/docker` or `/var/lib/containerd`. Log directories grow without rotation. Millions of small files can exhaust **inodes** while `df -h` still shows free blocks. Getting attributes wrong causes `Permission denied` that looks like a storage problem until you check `stat` and `namei -l`.

How it works

1. **Find the full mount** — `df -hT` and `df -i`. Note the **Mounted on** column, not only `/`.
2. **Find the hot directory** — `du -h --max-depth=1 /path | sort -h` (may need `sudo` outside your home).
3. **Inspect a file** — `stat -c '%n %A %U %G %s' file` for scripts; full `stat` for humans.
4. **When `df` and `du` disagree** — look for other mounts and for deleted-but-open files (`sudo lsof +L1` when available).

Symptom	Likely cause
`df` full, `du` on `/` smaller	Other mount full, or deleted-open files
`df -h` free, create fails	Inode exhaustion (`df -i`)
Permission denied	Mode/owner/ACL — not “disk full”

Common pitfalls

- Checking only `/` while `/var` or a data volume is full.
- Ignoring inodes — many tiny files exhaust them first.
- Restarting nothing when `du` and `df` disagree; the process holding a deleted file must exit or close it.
- Leaving `chattr +i` on files and wondering why deploys cannot overwrite them.
- Sorting `du` wrong — use `sort -h` with human-readable sizes.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, build a sample directory tree, measure it with `du`, compare mounts with `df`, inspect attributes with `stat`, and save an evidence pack under `~/rebash-linux/lab05`.

Prerequisites

- Ubuntu 22.04/24.04 (or Debian) with `df`, `du`, `stat` (from `coreutils` / `util-linux`)
- Write access under `$HOME`

Lab environment

Workspace: `~/rebash-linux/lab05`

```
mkdir -p ~/rebash-linux/lab05 && cd ~/rebash-linux/lab05
set -euo pipefail
whoami | tee lab-user.txt
df -hT . | tee df-workspace.txt
```

Expected output: `lab-user.txt` and `df-workspace.txt` exist; workspace is on a real mount.

Real-world scenario

On-call reports “disk almost full” on a practice app VM. Before you expand the volume, you must prove **which mount** is tight, **which folder** grew, and whether a sample file’s attributes look normal. You create a controlled sample tree, measure it, and attach command output to the ticket.

Step-by-step tasks

Task 1 – Build a sample tree and measure with `du`

```
cd ~/rebash-linux/lab05
set -euo pipefail

rm -rf sample-tree
mkdir -p sample-tree/{logs,cache,data}

dd if=/dev/zero of=sample-tree/logs/app.log bs=1M count=8 status=none
dd if=/dev/zero of=sample-tree/cache/pkg.bin bs=1M count=4 status=none
printf 'config=ok\n' > sample-tree/data/app.conf
for i in $(seq 1 50); do
  printf 'x' > "sample-tree/cache/tiny-$i.dat"
done

du -sh sample-tree | tee du-total.txt
du -h --max-depth=1 sample-tree | sort -h | tee du-depth1.txt
du -ah sample-tree | sort -h | tail -n 8 | tee du-largest.txt

test "$(du -sb sample-tree/logs | awk '{print $1}')" -gt "$(du -sb sample-tree/data | awk '{print $1}')"
```

Expected output: `du-total.txt` shows roughly 12M+ for the tree; `du-depth1.txt` lists `logs`, `cache`, and `data`; `logs` is larger than `data`.

Task 2 – Mount capacity with `df` (blocks and inodes)

```
cd ~/rebash-linux/lab05
set -euo pipefail

df -hT | tee df-hT.txt
df -i | tee df-i.txt
df -hT . / | tee df-here-and-root.txt
findmnt -T . | tee findmnt-here.txt || true
```

```
find sample-tree -printf '.' | wc -c | tee inode-ish-count.txt
test -s df-hT.txt
test -s df-i.txt
```

Expected output: `df-hT.txt` shows filesystem type and free space; `df-i.txt` shows inode use; workspace mount appears in `df-here-and-root.txt`.

Task 3 – File attributes with `stat` and evidence pack

```
cd ~/rebash-linux/lab05
set -euo pipefail

stat sample-tree/data/app.conf | tee stat-app.conf.txt
stat -c 'name=%n mode=%A owner=%U group=%G size=%s inode=%i' \
  sample-tree/data/app.conf sample-tree/logs/app.log | tee stat-summary.txt
ls -l sample-tree/data/app.conf | tee ls-app.conf.txt

stat -c '%s' sample-tree/logs/app.log | tee log-bytes.txt
test "$(cat log-bytes.txt)" -eq $((8 * 1024 * 1024))

tar -czf disk-usage-evidence.tgz \
  lab-user.txt df-workspace.txt \
  du-total.txt du-depth1.txt du-largest.txt \
  df-hT.txt df-i.txt df-here-and-root.txt \
  stat-app.conf.txt stat-summary.txt ls-app.conf.txt log-bytes.txt
ls -l disk-usage-evidence.tgz | tee evidence-ls.txt
```

Expected output: `stat-summary.txt` shows mode/owner/size; `log-bytes.txt` is `8388608`; evidence archive is not empty.

Validation steps

- [ ] `du -sh sample-tree` reports a size consistent with the `dd` files (~12M+)
- [ ] `df -hT` and `df -i` both produced output files
- [ ] `stat` shows owner and mode for `app.conf`
- [ ] `disk-usage-evidence.tgz` exists under `~/rebash-linux/lab05`

Common errors and fixes

Error	Cause	Fix
`du: cannot read directory`	Permission on system paths	Stay under `$HOME` for this lab; use `sudo` only when needed
`sort: invalid option` for `-h`	Very old `sort`	Install coreutils, or sort by `du -k` numeric column
`df` looks fine but writes fail	Wrong mount or inode full	Check `df -i` and `findmnt -T path`
Sizes look “wrong” after delete	File still open	Find holders with `lsof` / restart the process

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `~/rebash-linux/lab05/capacity-scan.sh` that: (1) prints `df -hT` for `.`, (2) prints the three largest entries under `sample-tree` using `du`, and (3) exits `0` only if free space on `.` is above 100M (parse `df` carefully). Save a sample run to `capacity-scan.out`.

Learning outcomes

- Separated mount capacity (`df`) from tree usage (`du`)
- Checked inode capacity
- Inspected file attributes with `stat`
- Saved ticket-ready evidence

Cleanup

```
cd ~/rebash-linux/lab05
set -euo pipefail
rm -rf sample-tree
# Keep evidence if you want it:
# rm -f disk-usage-evidence.tgz *.txt capacity-scan.sh capacity-scan.out
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab05/` with evidence files
- [ ] You can explain when `df` and `du` disagree
- [ ] You know to check `df -i` on "no space left" errors
- [ ] You can read mode/owner/size from `stat` without guessing

Code Walkthrough

In production capacity work for **Disk Usage and File Attributes**:

1. **Identify the mount** — `df -hT`, `df -i`, `findmnt`
2. **Find the growth** — `du` on that mount (not always on `/`)
3. **Confirm attributes** — `stat` / `ls -l` before blaming "disk" for permission errors
4. **Check deleted-open files** when numbers disagree
5. **Fix forward** — rotate logs, prune caches, expand volume, then re-measure

Automate the boring checks; keep humans for "is this safe to delete?"

Security Considerations

- Do not run destructive `rm -rf` on production paths from a capacity ticket without a second check
- Restrict who can read application log trees that may contain secrets
- Treat `sudo lsof` output carefully — it can expose paths with credentials
- Immutable flags (`chattr +i`) are a security control; document them
- Prefer least privilege: measure under the app user's directories when possible

Common Mistakes

Checking only the root filesystem

`/var` or a data disk can be full while `/` looks fine. **Fix:** read the Mounted on column from `df -hT` for the path that failed.

Ignoring inodes

Creates fail with "No space left" even when `df -h` shows free megabytes. **Fix:** always run `df -i`.

Deleting large files that are still open

Space does not return until the process closes the file. **Fix:** find the process (`lsof`), restart or reopen logs, then confirm with `df`.

Using `du` without knowing the mount

Summarising `/` includes other mounts and confuses the picture. **Fix:** `du` the specific mount path from `findmnt` / `df`.

Best Practices

- Alert on filesystem use **and** inode use for busy mounts
- Separate OS and data volumes on cloud VMs
- Keep log rotation and container prune jobs on a schedule
- Record `df` / `du` output in incident tickets before and after cleanup
- Document any `chattr` usage in configuration management

Troubleshooting

Symptom	Likely cause	Fix
`No space left on device`	Blocks or inodes full	`df -hT`, `df -i`, then `du`
`df` full, `du` smaller	Deleted-open or other mount	`lsof +L1`, check all mounts
Permission denied on write	Mode/owner/ACL	`stat`, `namei -l`, not `df`
Deploy cannot overwrite config	Immutable attribute	`lsattr`; clear with `chattr -i` if intended
`du` very slow on huge trees	Millions of files	Limit depth; exclude mounts; use offline reporting

Summary

`df` tells you about **mounts**; `du` tells you about **directories**; `stat` tells you about **one file**. Use all three before you expand a disk or delete data. Next, learn identity controls in Users, Groups, and sudo.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between `df` and `du`, and when would you trust each?

Answer

`df` reports free space and inodes for a **mounted filesystem**. `du` reports how much space a **directory tree** uses. Trust `df` for "can I write on this mount?" and `du` for "which folder grew?". When they disagree, check other mounts and deleted-but-open files.

2. A create fails with "No space left on device" but `df -h` shows free space. What do you check next?

Answer

Run `df -i`. The filesystem may be out of **inodes** (common with many tiny files). Also confirm you are looking at the correct mount for the path (`findmnt -T path`).

3. Why can deleting a large log file not free space immediately?

Answer

If a process still has the file open, the kernel keeps the space allocated until the last file descriptor closes. Use tools such as `lsof +L1` (with appropriate privilege), restart or reopen the logging process, then re-check `df`.

4. How do you find the largest directories under `/var` in an interview-style answer?

Answer

Prefer something like `sudo du -h --max-depth=1 /var | sort -h`, then drill into the largest child. Mention that you first confirm `/var` is its own mount or part of `/` with `df` / `findmnt`, so you clean the right place.

5. Which file attributes does `stat` show that matter in a "permission denied" ticket?

Answer

Mode (permissions), owner, group, and whether the path components are traversable (execute on directories). Size and timestamps help for "who changed it when", but mode/owner/group (plus ACL/MAC on hardened hosts) decide access.

6. How would you prove in a change ticket that a cleanup worked?

Answer

Attach **before and after** `df -hT` (and `df -i` if relevant) for the affected mount, plus a `du` summary of the cleaned tree. Show the commands and timestamps. Capacity work without before/after numbers is incomplete.

7. When might you use `chattr +i`, and what is the operational risk?

Answer

Use immutability for critical files that must not change accidentally (for example a golden config). The risk is forgotten flags blocking deploys or emergency fixes. Always document the flag and know how to clear it (`chattr -i`) during an incident.

Users, Groups, and sudo
MODULE 4 · USERS & PERMISSIONS

Chapter 6. Users, Groups, and sudo

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/users-groups-and-sudo/

Overview

When you log in to a Linux server, the system must know three things: **who you are**, **which team groups you belong to**, and **which admin commands you are allowed to run**. These three ideas are **users**, **groups**, and **sudo**.

A **user** is an account with a name, a number called a user ID (UID), a home folder, and a shell (the program that reads your commands). A **group** is a team name with a group ID (GID). Groups help many people share the same folder access without giving "open to everyone" permission. **sudo** means "superuser do". It lets a normal user run selected commands as root (the full admin user), so you do not stay logged in as root all day. In this tutorial you will create accounts, check them with `id`, and see your rights with `sudo -l`.

File permissions, services, containers, and cloud tools all depend on this identity layer. On cloud virtual machines (VMs), jump servers (bastions), Continuous Integration (CI) build machines, and Kubernetes nodes, wrong users or groups cause failed deployments, `Permission denied` errors, or scripts that have more power than they should. Cloud VM images often add the first login user to a sudo or admin group. In real work you should keep **human login accounts** separate from **service accounts** (accounts for apps, usually with a `nologin` shell), add people to the correct groups with `usermod -aG`, and give only small, clear sudo rules under `/etc/sudoers.d/` instead of one rule that allows everything.

In production, a bad sudo rule can affect the whole server. If you give `NOPASSWD:ALL`, any program running as that user can become root without asking for a password. If you edit `/etc/sudoers` with a normal editor and make a typing mistake, you may lose sudo access. Good practice is: role-based groups, small sudo files checked with `visudo`, and proof using `sudo -l` plus one command that must fail. Big companies may use a central login system such as Lightweight Directory Access Protocol (LDAP), FreeIPA, or a cloud directory. Even then, keep a few local emergency admin accounts for when the central system is down.

This is **Tutorial 6** in **Module 4: Users & Permissions** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a small working setup you can explain in an interview or in a change ticket.

Prerequisites

- Disk Usage and File Attributes
- A **practice Ubuntu 22.04/24.04 VM** (or similar) where you already have `sudo`
- You are ready to create and delete lab users (do **not** run this lab on a shared production server)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain how users, groups, and sudo map to `/etc/passwd`, `/etc/group`, and `/etc/sudoers.d/`
- [ ] Create a normal login user and a system (app) account with the correct shell
- [ ] Add a user to a group with `usermod -aG` without removing other groups
- [ ] Create a limited sudoers file with `visudo` and check it with `sudo -l`
- [ ] Remove lab accounts safely and explain one common production problem with identity

Architecture

Linux identity sits between people or automation and the kernel. Account files decide the UID and GID. sudo decides which privileged commands are allowed. Files and processes then use those identities.

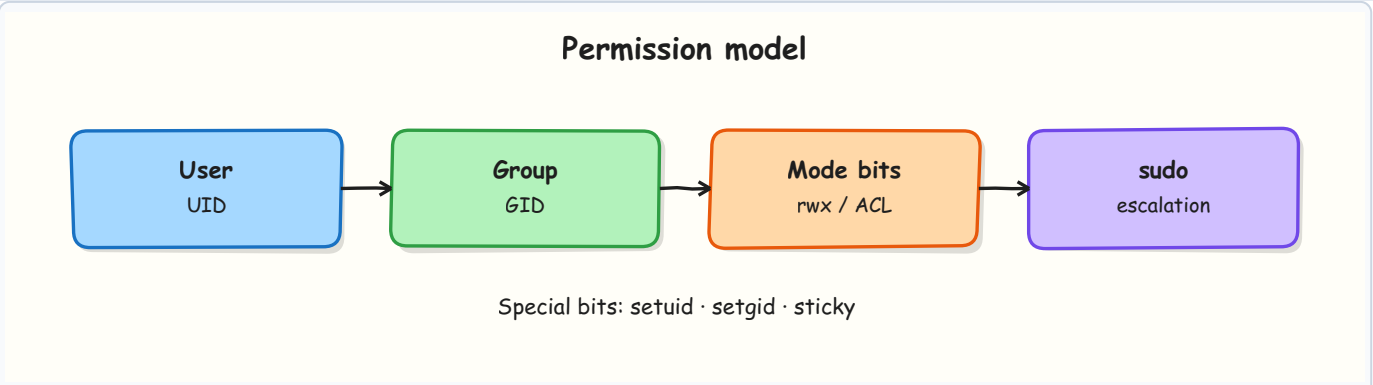

Figure 6.1: Users, Groups, and sudo

Theory

What it is

A **user** has a UID, a home directory, a login shell, and a primary group. A **group** has a GID and a list of members. User details are stored in `/etc/passwd`. Password hashes are stored in `/etc/shadow`. Group membership is stored in `/etc/group`.

sudo lets an allowed user run a command as another user (usually root), based on rules in `/etc/sudoers` and extra files in `/etc/sudoers.d/`. App or service accounts often use `/usr/sbin/nologin` (or `/bin/false`) so they can own files but nobody can log in interactively as that account.

```
id
getent passwd "$USER"
getent group sudo   # Ubuntu; on RHEL-like systems the group is often 'wheel'
```

Why it matters

Who can become root is one of the most important security decisions on a server. If sudo is too open (`ALL=(ALL) NOPASSWD:ALL`), a compromised CI user or app user becomes full root. If a required group is missing, deployment jobs fail when they cannot write to a shared folder. Cloud images usually put the default user in a sudo/admin group. You need to understand that default so you can harden jump servers and check who can become root.

How it works

1. **Check** — `id`, `getent passwd name`, `getent group name`.
2. **Create** — `useradd` / `adduser`, `groupadd`. Use `--system` for app/daemon accounts.
3. **Add to group** — `usermod -aG group user` (**`-a` means add**; `-G` alone can replace the whole secondary group list).
4. **Use sudo** — `sudo -l` shows your rules; `sudo -u otheruser command` runs a command as another user.
5. **Edit sudo rules** — use only `visudo` (or `visudo -f /etc/sudoers.d/file`) so a syntax error does not lock you out.

```
sudo useradd -m -s /bin/bash appuser
sudo usermod -aG sudo appuser          # Ubuntu example — the lab uses a tighter rule
sudo visudo -f /etc/sudoers.d/99-lab   # always check syntax
```

In large companies, LDAP, FreeIPA, or a cloud directory may provide user data to `getent`. Local files still matter for **emergency admin accounts** when the central directory is unavailable.

Key concepts and comparisons

Object	Important fields	Files
User	UID, home, shell, primary GID	`/etc/passwd`, `/etc/shadow`
Group	GID, members	`/etc/group`
sudo rule	Who, as whom, which commands	`/etc/sudoers`, `/etc/sudoers.d/*`

Pattern	Prefer when	Avoid when
Login user + limited sudo	People on jump servers	Putting passwords or secrets in sudoers
System user + `nologin`	Application services	Giving services a normal login shell
Group-based sudo	One team sharing the same allow-list	A unique full-root rule for every person
`NOPASSWD` for one script	Narrow automation only	Server-wide `NOPASSWD:ALL`

Common pitfalls

- Using `usermod -G` **without** `-a`, which can remove the user from other groups.
- Editing `/etc/sudoers` with nano/vim and creating a syntax error (you may lose sudo).
- Giving broad `NOPASSWD` "for CI" on the same server that people use for daily login.
- Running `userdel -r` without checking running processes, cron jobs, and file ownership.
- Assuming UIDs are the same on every server — Network File System (NFS) and shared storage use **numbers**, not names.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, create a team group, one human lab user, one system service account, and a **limited** sudoers file. Then prove group membership and sudo rights, and save command output under `~/rebash-linux/lab06`.

Prerequisites

- Ubuntu 22.04/24.04 (or Debian) with an admin user that already has sudo
- Packages: `sudo`, `passwd` (already present on Ubuntu)
- Take a VM snapshot before you start, if your hypervisor supports it

Lab environment

Workspace: `~/rebash-linux/lab06`

```
mkdir -p ~/rebash-linux/lab06 && cd ~/rebash-linux/lab06
set -euo pipefail
whoami | tee admin-user.txt
id | tee admin-id.txt
test -n "$(command -v sudo)"
sudo -n true 2>/dev/null || sudo -v
```

Expected output: `admin-user.txt` and `admin-id.txt` exist; `sudo` works (you may enter your password once).

Real-world scenario

Your team is setting up a new Ubuntu VM for a small application. Security asks for: (1) a shared group for deployers, (2) a non-login service account that will own app files later, and (3) a human engineer who can restart one named service with sudo — **not** full unrestricted root. You set up the accounts and sudo rules, and keep proof for the change ticket.

Step-by-step tasks

Task 1 – Create group, human user, and service account

Create the accounts first. Add sudo rules only after that.

```
cd ~/rebash-linux/lab06
set -euo pipefail

sudo groupadd rebash-lab || true

# Human engineer (login shell + home)
if ! id rebash-alice >/dev/null 2>&1; then
  sudo useradd -m -s /bin/bash -G rebash-lab rebash-alice
fi

# System service account (no interactive login)
```

```
if ! id rebash-svc >/dev/null 2>&1; then
  sudo useradd --system --home /opt/rebash-lab-svc --shell /usr/sbin/nologin \
    --gid rebash-lab rebash-svc
fi

# Make sure alice is in the lab group (add - do not remove other groups)
sudo usermod -aG rebash-lab rebash-alice

id rebash-alice | tee id-alice.txt
id rebash-svc | tee id-svc.txt
getent group rebash-lab | tee group-rebash-lab.txt

grep -E 'rebash-alice|rebash-svc' /etc/passwd | tee passwd-snippet.txt
```

Expected output: `id-alice.txt` shows group `rebash-lab`; `id-svc.txt` shows a `nologin` shell (or similar) and the lab group; `group-rebash-lab.txt` lists the members (exact format can vary a little).

Task 2 – Limited sudoers file

Allow `rebash-alice` to run **only** `systemctl status` and `systemctl restart` for a sample service name. We check syntax and `sudo -l`. The service does not need to exist yet.

```
cd ~/rebash-linux/lab06
set -euo pipefail

# Write a file, check it with visudo, then install it
TMP="$(mktemp)"
cat > "$TMP" << 'EOF'
# REBASH lab - limited sudo for rebash-alice
Defaults:rebash-alice !requiretty
rebash-alice ALL=(root) NOPASSWD: /bin/systemctl status rebash-lab.service, /bin/systemctl restart rebash-lab.service
EOF

sudo visudo -c -f "$TMP"
sudo install -m 0440 "$TMP" /etc/sudoers.d/99-rebash-lab-alice
rm -f "$TMP"
sudo visudo -c

# Show what alice is allowed to run
sudo -u rebash-alice sudo -l | tee sudo-l-alice.txt
grep -F 'systemctl' sudo-l-alice.txt
```

Expected output: `visudo -c` says the file is OK; `sudo-l-alice.txt` lists the two `systemctl` paths (and not `ALL`).

Task 3 – Negative test and evidence pack

Prove that alice is **not** full root, then pack the proof files.

```
cd ~/rebash-linux/lab06
set -euo pipefail

# This must fail (command is not in the allow-list)
if sudo -u rebash-alice sudo -n /bin/true 2>sudo-denied.txt; then
  echo "ERROR: unrestricted sudo - abort" >&2
  exit 1
fi
grep -Ei 'not allowed|sorry|denied|password' sudo-denied.txt || test -s sudo-denied.txt

tar -czf identity-evidence.tgz \
  admin-user.txt admin-id.txt \
  id-alice.txt id-svc.txt group-rebash-lab.txt passwd-snippet.txt \
  sudo-l-alice.txt sudo-denied.txt
ls -l identity-evidence.tgz | tee evidence-ls.txt
```

Expected output: `/bin/true` with sudo is denied; `identity-evidence.tgz` is not empty.

Validation steps

- [ ] `id rebash-alice` shows group `rebash-lab`
- [ ] `id rebash-svc` uses a non-login shell
- [ ] `/etc/sudoers.d/99-rebash-lab-alice` mode is `0440` (`ls -l`)
- [ ] `sudo -u rebash-alice sudo -l` shows only the intended `systemctl` commands
- [ ] `identity-evidence.tgz` exists under `~/rebash-linux/lab06`

Common errors and fixes

Error	Cause	Fix
`useradd: user already exists`	Lab run again	Safe — the script checks with `id`; continue
`visudo: parse error`	Wrong sudoers syntax	Fix the temp file; never copy a broken file into `/etc/sudoers.d/`
`sudo: a password is required` for alice	NOPASSWD rule missing	Check the drop-in path and run `visudo -c`
`usermod: group 'rebash-lab' does not exist`	`groupadd` was skipped	Run Task 1 from the start
Locked out after bad sudoers	Edited without `visudo`	Use root console / recovery; remove the bad file

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Create user `rebash-bob`, add him to `rebash-lab`, and add a **second** sudoers file that allows **only** `sudo -u rebash-svc /usr/bin/id` (so bob can check the service account without becoming root). Prove with `sudo -u rebash-bob sudo -l` and save `sudo-l-bob.txt`. Remove bob and his sudoers file in Cleanup if you create them.

Learning outcomes

- Created human and system accounts with the right shells
- Used `usermod -aG` safely
- Installed a checked sudoers file and proved both allow and deny
- Saved identity proof suitable for a change ticket

Cleanup

```
cd ~/rebash-linux/lab06
set -euo pipefail

sudo rm -f /etc/sudoers.d/99-rebash-lab-alice
sudo rm -f /etc/sudoers.d/99-rebash-lab-bob 2>/dev/null || true
sudo visudo -c

sudo userdel -r rebash-alice 2>/dev/null || sudo userdel rebash-alice || true
sudo userdel -r rebash-bob 2>/dev/null || sudo userdel rebash-bob || true
sudo userdel rebash-svc 2>/dev/null || true
sudo groupdel rebash-lab 2>/dev/null || true

# Keep the evidence archive if you want it; otherwise:
# rm -f identity-evidence.tgz *.txt
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab06/` with evidence files
- [ ] You can explain UID, GID, sudoers drop-in files, and why `-aG` matters
- [ ] You can explain the risk of server-wide `NOPASSWD:ALL`
- [ ] You know when to use a system user with `nologin` instead of a login shell

Code Walkthrough

In real servers, identity work for **Users, Groups, and sudo** usually follows this order:

1. **Check before you change** — `id`, `getent`, `sudo -l` for the current admin
2. **Prefer small files** — rules under `/etc/sudoers.d/` with mode `0440`; avoid one-off edits to `/etc/sudoers` without review
3. **Check syntax** — `visudo -c` before and after
4. **Prove allow and deny** — `sudo -l` for success, plus one command that must fail
5. **Least privilege** — list exact commands; avoid `ALL`

Later you can create accounts with Ansible or similar tools. People still review the design and keep emergency admin access.

Security Considerations

- Treat sudo access as critical — review it in CI or configuration management
- Never store passwords or API tokens in sudoers or in world-readable home files
- Prefer key-based Secure Shell (SSH) for people; turn off password login on internet-facing servers (covered in later security modules)
- Keep human accounts separate from service accounts (`nologin`)
- Log privileged use (`/var/log/auth.log` or `journalctl`) and limit who can read those logs

Common Mistakes

Using `usermod -G` without `-a`

Secondary groups are replaced, not added. **Fix:** always use `usermod -aG group user`, then check with `id user`.

Editing sudoers with nano/vim directly

One typing mistake can remove your sudo access. **Fix:** use `sudo visudo` or `sudo visudo -f /etc/sudoers.d/…`, and keep a root console open while you change rules.

Giving `NOPASSWD:ALL` for convenience

Any process running as that user becomes root. **Fix:** allow only exact commands; use separate automation users with narrow rules.

Deleting users without checking ownership

Old files can keep the old UID under `/var` or app folders. **Fix:** search carefully with `find / -user name`, fix ownership, then run `userdel`.

Best Practices

- One sudoers file per team or role, managed by configuration management
- System users for services; login users for people
- Name groups by purpose (`deploy`, `logs-read`), not by person name
- Review `sudo -l` output in pull requests when you change jump-server images
- Document emergency admin accounts and test recovery every few months

Troubleshooting

Symptom	Likely cause	Fix
`user is not in the sudoers file`	No rule / wrong user	Check `/etc/sudoers.d/*` and group membership
`sudo: parse error` / sudo fails for all	Broken sudoers syntax	Root console; `visudo -c`; remove the bad file
Group missing in `id` after `usermod`	New group not loaded in this session	Log out and log in again, or use `newgrp`, or reconnect with SSH
Service cannot write files	Wrong UID/GID on folders	Set ownership to the service account
CI job suddenly has root	sudo rule is too open	Narrow the rule; rotate secrets; audit

Summary

Users, groups, and sudo decide **who can do what** on Linux. Create accounts with a clear purpose (login vs service), add groups safely, and give sudo as a short allow-list — then prove both success and failure with command output. Next, learn file modes and Access Control Lists (ACLs) in Permissions, ACLs, and Special Bits.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between a user's primary group and secondary groups, and how do you add a secondary group without removing others?

Answer

The **primary group** is the default group ID (GID) used when the user creates new files. **Secondary groups** give extra shared access (for example a `deploy` group on a shared folder). To add a secondary group without removing others, use `usermod -aG groupname username`, then check with `id username`. If you forget `-a`, the secondary group list can be replaced and access can break.

2. A junior engineer edited `/etc/sudoers` with vim and now `sudo` fails for everyone. How do you recover, and how do you prevent it next time?

Answer

Recover using a **root console**, single-user mode, or the cloud **serial console**. Fix or remove the broken sudoers file, then run `visudo -c`. To prevent this, edit only with `visudo` (or `visudo -f /etc/sudoers.d/…`), prefer small drop-in files, and test on a practice VM first. Interviewers look for a clear recovery plan and a syntax check — not "reboot and hope".

3. When should an application use a system account with `/usr/sbin/nologin` instead of a normal login user?

Answer

Use a **system account** with `nologin` (or `/bin/false`) when a service needs a UID to own files and processes but **must not** allow interactive login. Login shells are for people. Service accounts reduce risk if a password or SSH key is stolen, and make app folder ownership clearer in audits.

4. Why is `ALL=(ALL) NOPASSWD:ALL` dangerous on a shared jump server, and what would you grant instead for "restart this one service"?

Answer

Server-wide `NOPASSWD:ALL` means any code running as that user becomes **full root** with no password prompt. Prefer a narrow rule with full command paths, for example
`user ALL=(root) NOPASSWD: /bin/systemctl restart myapp.service`.
Prove with `sudo -l` and a negative test that a blocked command is rejected. Keep sudo use in auth logs.

5. How would you prove in an interview (or ticket) that a sudo change is least privilege?

Answer

Show the drop-in file, a successful `visudo -c`, `sudo -l` for that user listing **only** the intended commands, and a **deny** test (`sudo /bin/true` or similar) that fails. Attach that proof to the change ticket. Least privilege is shown by what is *not* allowed, as well as by what is allowed.

6. UIDs differ between two servers that mount the same NFS share — what breaks, and how do teams usually avoid that?

Answer

Network File System (NFS) and similar shared storage check **numeric UID/GID**, not the login name. The same username with different UIDs on two servers can see wrong ownership or `Permission denied`. Teams avoid this with a central directory (LDAP/FreeIPA/IdM), fixed UID ranges, or by not sharing POSIX folders across unmanaged servers.

7. How does cloud "admin" group membership on the default image relate to local sudo configuration?

Answer

Most cloud images put the first login user in a group such as `sudo` (Ubuntu) or `wheel` (RHEL-like systems). That group is already mapped to broad sudo in `/etc/sudoers`. This is useful for first setup, but production hardening often reduces that access, adds role-based files under `/etc/sudoers.d/`, and keeps separate emergency admin accounts. Always check with `id` and `sudo -l` on a new image before you trust the default.

Permissions, ACLs, and Special Bits
MODULE 4 · USERS & PERMISSIONS

Chapter 7. Permissions, ACLs, and Special Bits

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/permissions-acls-and-special-bits/

Overview

Most "Permission denied" tickets are about **mode**, **ownership**, or **umask** — not mysterious kernel bugs. Linux file access starts with three classes: **user (owner)**, **group**, and **other**, each with read (4), write (2), and execute (1). **umask** masks default permissions when new files are created. **Access Control Lists (ACLs)** add named users or groups beyond those three classes. **Special bits** — sticky, setuid (SUID), and setgid (SGID) — change delete rules or the effective identity when a program runs.

Shared deploy folders need group write without opening "other". `/tmp` needs the sticky bit so users cannot delete each other's files. Unexpected SUID binaries are a classic hardening finding. In this tutorial you will set modes and ownership, apply an ACL, demonstrate a sticky directory, and save proof under `~/rebash-linux/lab07`.

In production, wrong modes on Secure Shell (SSH) keys, application configs, or CI artefact directories break deployments and create security findings. Prefer group or ACL sharing over world-writable paths. Audit SUID/SGID regularly on jump servers and build agents.

This is **Tutorial 7** in **Module 4: Users & Permissions** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Users, Groups, and sudo
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Package `acl` for `setfacl`/`getfacl` (`sudo apt-get install -y acl` if missing)
- You may create and delete lab users (do **not** run this on a shared production server)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Set and verify POSIX modes with `chmod` and ownership with `chown`/`chgrp`
- [ ] Explain umask and check the current umask
- [ ] Grant a named user access with `setfacl` and prove it with `getfacl`
- [ ] Apply the sticky bit on a shared drop directory and explain why it matters
- [ ] Pack evidence under `~/rebash-linux/lab07`

Architecture

Permissions sit between identity (UID/GID) and the filesystem. Mode bits, ACLs, and special bits decide whether a process may read, write, traverse, or delete.

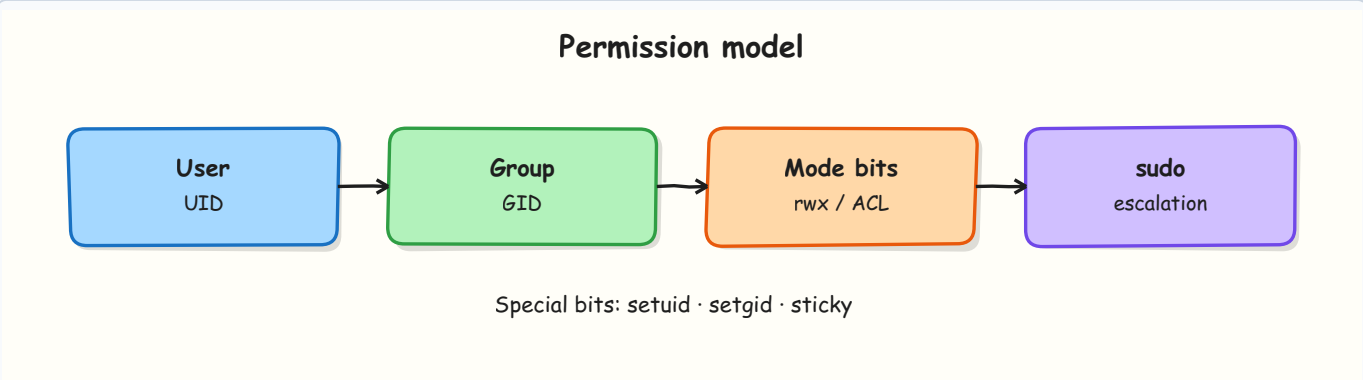

Figure 7.1: Permissions, ACLs, and Special Bits

Theory

What it is

POSIX permissions use three classes. Directories need the execute bit to **traverse** (enter) the path. ACLs add entries such as `user:alice:rwx`. Sticky (`+t`) on a directory restricts unlinking to the file owner (plus root). SUID on an executable runs it as the file owner; SGID runs as the file group. SGID on a directory often makes new files inherit the directory's group.

```
ls -l
stat -c '%A %U %G %n' file
umask
getfacl file
```

Why it matters

"Permission denied" is rarely mysterious once you inspect mode, owner, group, ACL, and Mandatory Access Control (MAC) such as AppArmor or SELinux. Shared project trees need group write without `o+rwx`. World-writable deploy directories are a common audit failure. Wrong modes on `~/.ssh` (`700`/`600`) lock people out of servers.

How it works

`chmod` sets mode (octal `640` or symbolic `u=rw,g=r,o=`). Capital `X` in symbolic mode sets execute only on directories or on files that already had execute. `chown`/`chgrp` change ownership (often requiring root). umask `0022` typically yields `644` files and `755` directories; `0027` is tighter. `setfacl`/`getfacl` manage ACLs; default ACLs on directories apply to new children.

Mechanism	Granularity	Typical use
POSIX mode	owner/group/other	Default access model
ACL	named users/groups	Shared dirs without widening other
Sticky	directory delete rules	`/tmp`, shared drop boxes
SGID directory	group inheritance	Team project trees
SUID/SGID file	effective UID/GID at run	Rare; prefer capabilities

umask	Typical file / dir
`0022`	`644` / `755`
`0002`	`664` / `775` (group-friendly)
`0027`	`640` / `750` (tighter)

Common pitfalls

- Widening `o+rwx` instead of using a group or ACL.
- Forgetting execute on directories — path lookup fails.
- Leaving unexpected SUID/SGID binaries after experiments.
- Applying ACLs on filesystems mounted without ACL support.

- Changing ownership of SSH keys and locking yourself out.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, create a shared project directory with correct group modes, grant an ACL to a lab user, demonstrate sticky-bit delete behaviour, and save evidence under `~/rebash-linux/lab07`.

Prerequisites

- Ubuntu with `sudo`, `chmod`, `chown`, and the `acl` package
- Ability to create temporary users

Lab environment

Workspace: `~/rebash-linux/lab07`

```
mkdir -p ~/rebash-linux/lab07 && cd ~/rebash-linux/lab07
set -euo pipefail
sudo apt-get update -qq
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y acl
umask | tee umask-default.txt
whoami | tee lab-admin.txt
```

Expected output: `acl` tools available; `umask-default.txt` shows a value such as `0022`.

Real-world scenario

Your team shares a deploy drop folder on a practice VM. Security wants: (1) group-writable for deployers, (2) one contractor with read-only ACL access, (3) sticky bit so people cannot delete each other's artefacts. You implement and prove it before the change ticket closes.

Step-by-step tasks

Task 1 – Group, users, and POSIX modes

```
cd ~/rebash-linux/lab07
set -euo pipefail

sudo groupadd rebash-perm || true
if ! id rebash-dev >/dev/null 2>&1; then
  sudo useradd -m -s /bin/bash -G rebash-perm rebash-dev
fi
if ! id rebash-contractor >/dev/null 2>&1; then
  sudo useradd -m -s /bin/bash rebash-contractor
fi
sudo usermod -aG rebash-perm rebash-dev

sudo mkdir -p /opt/rebash-perm/shared
sudo chown root:rebash-perm /opt/rebash-perm/shared
sudo chmod 2770 /opt/rebash-perm/shared

sudo -u rebash-dev bash -c 'echo deploy-artefact > /opt/rebash-perm/shared/app.txt'
sudo -u rebash-dev bash -c 'ls -l /opt/rebash-perm/shared/app.txt' | tee ls-shared-file.txt
stat -c '%A %U %G %n' /opt/rebash-perm/shared /opt/rebash-perm/shared/app.txt | tee stat-shared.txt
getent group rebash-perm | tee group-rebash-perm.txt
```

Expected output: directory mode shows `rwx` for group and `s` in the group-execute position (SGID); `app.txt` group is `rebash-perm`.

Task 2 – ACL for the contractor (read-only)

```
cd ~/rebash-linux/lab07
set -euo pipefail

sudo setfacl -m u:rebash-contractor:rx /opt/rebash-perm/shared
sudo setfacl -m u:rebash-contractor:r /opt/rebash-perm/shared/app.txt
sudo getfacl /opt/rebash-perm/shared /opt/rebash-perm/shared/app.txt | tee getfacl-shared.txt

sudo -u rebash-contractor cat /opt/rebash-perm/shared/app.txt | tee contractor-read.txt
if sudo -u rebash-contractor bash -c 'echo hack > /opt/rebash-perm/shared/app.txt' 2>contractor-write-deny.txt; then
  echo "ERROR: contractor could write" >&2
  exit 1
fi
```

```
grep -Ei 'Permission denied|denied' contractor-write-deny.txt
grep -F 'user:rebash-contractor:r' getfacl-shared.txt
```

Expected output: `getfacl` lists the contractor ACL; read succeeds; write is denied.

Task 3 – Sticky drop box and evidence pack

```
cd ~/rebash-linux/lab07
set -euo pipefail

sudo mkdir -p /opt/rebash-perm/drop
sudo chown root:rebash-perm /opt/rebash-perm/drop
sudo chmod 1770 /opt/rebash-perm/drop

sudo -u rebash-dev bash -c 'echo from-dev > /opt/rebash-perm/drop/dev.txt'
if ! id rebash-dev2 >/dev/null 2>&1; then
  sudo useradd -m -s /bin/bash -G rebash-perm rebash-dev2
fi
sudo -u rebash-dev2 bash -c 'echo from-dev2 > /opt/rebash-perm/drop/dev2.txt'

if sudo -u rebash-dev2 rm -f /opt/rebash-perm/drop/dev.txt 2>sticky-deny.txt; then
  echo "ERROR: sticky bit did not block delete" >&2
  exit 1
fi
grep -Ei 'Permission denied|denied|Operation not permitted' sticky-deny.txt
ls -ld /opt/rebash-perm/drop | tee ls-drop.txt
grep -E 't|T' ls-drop.txt

tar -czf permissions-evidence.tgz \
  lab-admin.txt umask-default.txt \
  ls-shared-file.txt stat-shared.txt group-rebash-perm.txt \
  getfacl-shared.txt contractor-read.txt contractor-write-deny.txt \
  sticky-deny.txt ls-drop.txt
ls -l permissions-evidence.tgz | tee evidence-ls.txt
```

Expected output: sticky delete is denied; `ls-drop.txt` shows `t` in the mode; evidence archive exists.

Validation steps

- [ ] `/opt/rebash-perm/shared` is mode `2770` (or equivalent `drwxrws---`)
- [ ] `getfacl` shows `rebash-contractor` with read on the file
- [ ] Contractor write fails; sticky delete fails for the other user
- [ ] `permissions-evidence.tgz` exists under `~/rebash-linux/lab07`

Common errors and fixes

Error	Cause	Fix
`setfacl: command not found`	`acl` package missing	`sudo apt-get install -y acl`
`Operation not supported`	Filesystem without ACL	Use ext4/xfs home/data disk; avoid some network mounts
Contractor cannot enter directory	Missing `x` on directory ACL	`setfacl -m u:user:rx` on the directory
Sticky test unexpectedly succeeds	Mode not sticky	`chmod +t` / `chmod 1770` and re-check `ls -ld`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Add a **default ACL** on `/opt/rebash-perm/shared` so new files grant `rebash-contractor` read access automatically (`setfacl -d -m u:rebash-contractor:r`). Create a new file as `rebash-dev`, run `getfacl` on that file, and save output to `default-acl-proof.txt`.

Learning outcomes

- Applied SGID group-shared directory modes
- Used ACLs for a named user without widening “other”
- Proved sticky-bit delete protection
- Saved permission evidence for a change ticket

Cleanup

```
cd ~/rebash-linux/lab07
set -euo pipefail
```

```
sudo rm -rf /opt/rebash-perm
sudo userdel -r rebash-dev 2>/dev/null || true
sudo userdel -r rebash-dev2 2>/dev/null || true
sudo userdel -r rebash-contractor 2>/dev/null || true
sudo groupdel rebash-perm 2>/dev/null || true
# Keep permissions-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab07/` with evidence files
- [ ] You can explain POSIX mode vs ACL vs sticky bit
- [ ] You know why world-writable shared dirs are a bad default
- [ ] You can list when SUID on files is a security finding

Code Walkthrough

Production permission work usually follows:

1. **Identify** — `ls -l`, `stat`, `namei -l`, `id`
2. **Prefer group or ACL** over `chmod o+rwx`
3. **Prove allow and deny** for the real users
4. **Special bits** — sticky on shared drop dirs; avoid casual SUID
5. **Automate** modes in configuration management, not one-off SSH

Security Considerations

- Never make secrets world-readable (`o+r` on key material)
- Keep `~/.ssh` at `700` and private keys at `600`
- Audit SUID/SGID binaries (`find / -perm /6000`) on hardened images
- ACLs can hide access — always include `getfacl` in reviews
- Sticky bit on shared directories reduces cross-user deletion attacks

Common Mistakes

Fixing access with `chmod 777`

World-writable paths fail audits and invite tampering. **Fix:** use a group (`2770`) or a named ACL.

Forgetting directory execute

Users can “see” a file mode but still get “Permission denied” on the path. **Fix:** ensure `x` on every directory in the path.

Assuming ACL is optional documentation

Some NAS/NFS mounts ignore ACLs. **Fix:** verify with `getfacl` after mount; test as the named user.

Shipping SUID binaries for convenience

Any bug becomes a privilege escalation path. **Fix:** prefer capabilities, sudo allow-lists, or root helpers with narrow scope.

Best Practices

- Encode modes and ownership in Ansible/Puppet/cloud-init
- Use SGID directories for team trees; sticky for drop boxes
- Review umask on CI runners that publish artefacts
- Document ACL entries next to the directory purpose
- Re-test access after user leave (remove ACLs and group membership)

Troubleshooting

Symptom	Likely cause	Fix
Permission denied	Mode/owner/path `x` missing	`namei -l`, `stat`, fix mode
Works for one user only	Group/ACL missing	`id`, `getfacl`, `usermod -aG`
New files wrong group	Directory not SGID	`chmod g+s` on the directory
Delete others' files in shared dir	Sticky missing	`chmod +t`
ACL "ignored"	Mount options / FS type	Check `mount` options; remount with ACL

Summary

Modes, ownership, umask, ACLs, and special bits decide **who can read, write, traverse, and delete**. Prefer least privilege with groups and ACLs, prove allow and deny, and keep SUID rare. Next: Text Processing with grep, sed, and awk.

Interview Questions

INTERVIEW PRACTICE

1. What do the three POSIX permission classes mean, and what does execute mean on a directory?

Answer

The classes are **owner**, **group**, and **other**. On files, execute means run as a program. On directories, execute means **traverse** (enter/search) that directory. Without directory `x`, you cannot reach files inside even if the file mode looks open.

2. How do ACLs help when two users need access but should not share a primary group?

Answer

ACLs add named-user or named-group entries with `setfacl` without widening "other" or forcing a shared primary group. Prove with `getfacl` and a login/sudo-as that user. Default ACLs on directories apply to new children.

3. What is the sticky bit for, and how do you recognise it in `ls -ld`?

Answer

On a directory, sticky (`+t`) means only the **file owner** (or root) can unlink/rename files there — classic for `/tmp`. In `ls -ld`, you see a `t` or `T` in the other-execute position of the mode string (for example `drwxrwxrwt`).

4. Why is `chmod 777` on a deploy directory a security problem?

Answer

Any local user (or compromised low-privilege process) can change or replace artefacts. Prefer `2770` with a deploy group, or ACLs for specific users, and keep secrets out of that tree. Auditors flag world-writable paths quickly.

5. What is the difference between SUID on a file and SGID on a directory?

Answer

SUID on a file runs the program with the file owner's UID (powerful; audit carefully). **SGID on a directory** typically makes new files inherit the directory's group, which helps team-shared trees. Do not confuse the two in interviews.

6. How would you debug "Permission denied" on `/opt/app/bin/start`?

Answer

Check the full path with `namei -l`, then `stat` / `ls -l` on each component, `id` for the runtime user, `getfacl` if ACLs are in use, and MAC logs (AppArmor/SELinux) if modes look correct. Fix the first component that denies traverse or execute.

7. How does umask affect files created by CI jobs?

Answer

umask masks default permissions. A loose umask can create world-readable artefacts (secret leakage); a tight umask can make the next job unable to read outputs. Set umask deliberately in the job environment and verify with `stat` on a created file.

Text Processing with grep, sed, and awk
MODULE 5 · TEXT PROCESSING

Chapter 8. Text Processing with grep, sed, and awk

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/text-processing-grep-sed-awk/

Overview

In Cloud and DevOps work, most day-to-day evidence is **text**: application logs, Nginx or load-balancer access lines, Continuous Integration (CI) job output, Kubernetes events exported to a file, and configuration dumps. You rarely open a spreadsheet. You **search**, **edit**, and **summarise** streams of lines with classic Unix tools.

grep finds (or excludes) lines that match a pattern. **sed** edits a stream — substitute text, delete lines, print ranges. **awk** splits each line into fields and builds small reports (counts, totals, column extracts). Helpers such as `cut`, `tr`, `sort`, `uniq`, `wc`, and `xargs` complete the toolkit. Together they turn a noisy log into a clear answer in minutes during an incident or a change ticket.

On cloud virtual machines (VMs), jump servers, build agents, and nodes, these tools appear in runbooks and in automation. A junior engineer who can only open an editor is slow at 03:00. A professional who can write a safe pipeline (`grep` → `awk` → `sort | uniq -c`) finds the failing host, the top error code, or the bad config key without guessing. Production judgement means: keep a backup before `sed -i`, avoid recursive greps through huge binary trees, set field separators on purpose, and use `find … -print0 | xargs -0` when file names may contain spaces.

This is **Tutorial 8** in **Module 5: Text Processing** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a small evidence pack from a realistic log that you can explain in an interview.

Prerequisites

- Permissions, ACLs, and Special Bits
- A **practice Ubuntu 22.04/24.04 VM** (or similar) with a normal user account
- Basic shell skills: redirect with `>`, pipes `|`, and `tee`

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Choose grep, sed, or awk for a given log or config task
- [ ] Filter and invert matches with `grep`, including useful flags (`-E`, `-v`, `-n`, `-I`)
- [ ] Edit text safely with `sed` (stream edit and backup before in-place change)
- [ ] Build a field report with `awk` and rank frequencies with `sort | uniq -c`
- [ ] Package a copy-paste pipeline and evidence files under `~/rebash-linux/lab08`

Architecture

Text tools sit between raw files or streams and the answers you need for ops: filter → transform → summarise → feed the next command.

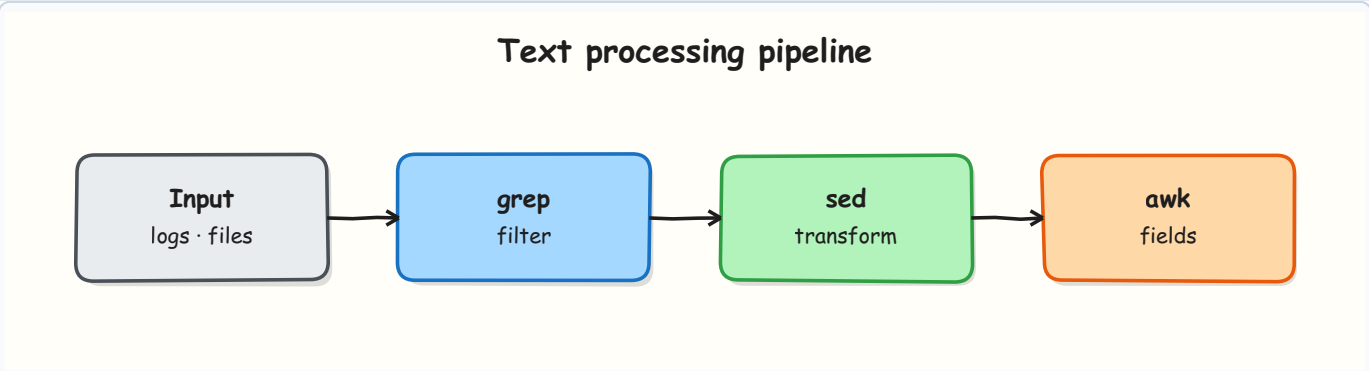

Figure 8.1: Text Processing with grep, sed, and awk

Theory

What it is

Text processing means working on line-oriented data without loading everything into a graphical tool.

Tool	Plain meaning	Typical job
`grep`	"Show me matching lines"	Find `ERROR`, exclude noise with `-v`
`sed`	"Edit the stream"	Replace a config value, strip prefixes
`awk`	"Split fields and report"	Column extract, counts, simple maths
`cut` / `tr`	Slice characters or change sets	Fixed delimiters, lowercase
`sort` / `uniq` / `wc`	Order, unique, count	Top-N errors, line counts
`xargs`	Turn lines into command arguments	Run a command on many files

```
grep -n ERROR app.log
sed 's/DEBUG/INFO/g' app.log
awk -F: '{print $1}' /etc/passwd | head
```

Why it matters

Incidents are mostly reading text under time pressure. Engineers who can filter with `grep -E`, extract with `awk -F`, and rank with `sort | uniq -c | sort -nr` resolve tickets faster. The same patterns appear in CI checks (fail the build if a pattern appears) and in configuration management validation. On shared logs, wrong regex or a recursive grep through `/var` can waste CPU and flood your terminal — so precision matters as much as speed.

How it works

1. **Find** — `grep -RIn --exclude-dir=.git PATTERN dir` (or search one file). `-E` enables extended Regular Expressions (regex). `-v` inverts. `-I` skips binary files.
2. **Edit** — `sed 's/old/new/g'` prints a changed stream. For files, prefer `sed -i.bak '…' file` so you keep a backup, then validate the result.
3. **Report** — `awk` splits on whitespace by default, or use `-F':'` / `-F','`. Fields are `$1`, `$2`, …; `NF` is field count; an `END { … }` block runs after the last line.
4. **Compose** — pipes chain tools: `grep ERROR file | awk '{print $1}' | sort | uniq -c | sort -nr`.
5. **Safe xargs** — `find . -name '*.log' -print0 | xargs -0 grep -H ERROR` so spaces in names do not break the command.

```
# Rank status codes from a space-separated sample access log (field 9 often holds the code)
awk '{print $9}' access.log | sort | uniq -c | sort -nr | head
```

Set `LC_ALL=C` when you need byte-order sort that does not depend on locale.

Key concepts and comparisons

Need	Prefer	Avoid when
Locate error lines	`grep -n` / `grep -E`	Opening a multi-GB file in a GUI editor
One substitution across many lines	`sed`	Hand-editing hundreds of lines
Column report / count	`awk`	Fragile `cut` on irregular spaces
Fixed delimiter columns	`cut -d: -f1`	Logs with quoted commas (use awk carefully)
Run a command per file	`find … -print0 | xargs -0`	Plain `xargs` on names with spaces

Flag / pattern	Meaning
`grep -v`	Exclude matching lines
`grep -E`	Extended regex
`sed -i.bak`	In-place edit with backup
`awk -F:`	Field separator is colon
`uniq -c`	Count adjacent identical lines (sort first)

Common pitfalls

- Using `sed -i` on production configs with **no backup** and no syntax check.
- Recursive `grep -R` through container image layers or binary directories without excludes.
- Forgetting `-F` in awk when fields are colon- or comma-separated.
- Using `xargs` without `-0` when names may contain spaces or newlines.
- Assuming locale sort order — set `LC_ALL=C` for stable, script-friendly ordering.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Build a small incident-style pipeline on sample logs: filter errors with grep, clean a field with sed, rank top messages with awk/`sort`/`uniq`, and save proof under `~/rebash-linux/lab08`.

Prerequisites

- Ubuntu 22.04/24.04 (or Debian) with bash
- Packages: `grep`, `sed`, `gawk` or `mawk`, `coreutils` (normally already installed)
- No sudo required for this lab

Lab environment

Workspace: `~/rebash-linux/lab08`

```
mkdir -p ~/rebash-linux/lab08 && cd ~/rebash-linux/lab08
set -euo pipefail
whoami | tee lab-user.txt
```

Expected output: directory exists; `lab-user.txt` contains your username.

Real-world scenario

A payment API on a practice VM is throwing errors after a deploy. You have a copied application log and a small access log. The on-call engineer asks: which error strings are most common, which HTTP status codes appear, and can you produce a one-command summary for the change ticket? You answer with a pipeline and saved output — not with screenshots of an editor.

Step-by-step tasks

Task 1 – Create sample logs

Create realistic sample files you can re-run the lab against.

```
cd ~/rebash-linux/lab08
set -euo pipefail

cat > app.log << 'EOF'
2026-08-02T10:01:01Z INFO  worker=1 started
2026-08-02T10:01:02Z ERROR worker=1 code=TIMEOUT msg=upstream_timeout host=api-1
2026-08-02T10:01:03Z WARN  worker=2 code=RETRY msg=retrying host=api-1
2026-08-02T10:01:04Z ERROR worker=1 code=TIMEOUT msg=upstream_timeout host=api-1
2026-08-02T10:01:05Z ERROR worker=3 code=AUTH msg=token_expired host=api-2
2026-08-02T10:01:06Z INFO  worker=2 request_ok host=api-1
2026-08-02T10:01:07Z ERROR worker=3 code=AUTH msg=token_expired host=api-2
2026-08-02T10:01:08Z ERROR worker=1 code=TIMEOUT msg=upstream_timeout host=api-1
2026-08-02T10:01:09Z DEBUG worker=4 ping host=api-3
2026-08-02T10:01:10Z ERROR worker=2 code=DB msg=connection_reset host=api-1
EOF

cat > access.log << 'EOF'
10.0.0.11 - - [02/Aug/2026:10:01:01 +0000] "GET /health HTTP/1.1" 200 12
10.0.0.22 - - [02/Aug/2026:10:01:02 +0000] "POST /pay HTTP/1.1" 502 34
10.0.0.33 - - [02/Aug/2026:10:01:03 +0000] "POST /pay HTTP/1.1" 502 34
10.0.0.22 - - [02/Aug/2026:10:01:04 +0000] "GET /health HTTP/1.1" 200 12
10.0.0.44 - - [02/Aug/2026:10:01:05 +0000] "POST /pay HTTP/1.1" 500 40
10.0.0.55 - - [02/Aug/2026:10:01:06 +0000] "GET /ready HTTP/1.1" 200 8
10.0.0.22 - - [02/Aug/2026:10:01:07 +0000] "POST /pay HTTP/1.1" 502 34
EOF

wc -l app.log access.log | tee wc-samples.txt
test "$(wc -l < app.log)" -eq 10
test "$(wc -l < access.log)" -eq 7
```

Expected output: `wc-samples.txt` shows 10 and 7 lines for the two files.

Task 2 – grep filter and sed cleanup

Extract ERROR lines, then build a clean `code=…` column with sed.

```
cd ~/rebash-linux/lab08
set -euo pipefail

grep -n 'ERROR' app.log | tee errors-raw.txt
test -s errors-raw.txt
grep -c 'ERROR' app.log | tee errors-count.txt
test "$(cat errors-count.txt)" -eq 5

# Keep only the code=VALUE token from each ERROR line
grep 'ERROR' app.log \
  | sed -E 's/.*code=([A-Z_]+).*/\1/' \
  | tee error-codes.txt

grep -E '^(TIMEOUT|AUTH|DB)$' error-codes.txt | tee error-codes-check.txt
test "$(wc -l < error-codes.txt)" -eq 5
```

Expected output: five ERROR lines; `error-codes.txt` lists codes such as `TIMEOUT`, `AUTH`, `DB` (one per line).

Task 3 – awk reports and frequency ranking

Rank error codes and HTTP status codes; pack evidence.

```
cd ~/rebash-linux/lab08
set -euo pipefail

# Top error codes from the cleaned list
sort error-codes.txt | uniq -c | sort -nr | tee top-error-codes.txt
grep -q 'TIMEOUT' top-error-codes.txt

# HTTP status codes from access.log (field 9 in this Combined-like format)
awk '{print $9}' access.log | sort | uniq -c | sort -nr | tee top-status-codes.txt
grep -q '502' top-status-codes.txt

# Hosts that saw ERROR in app.log
awk '/ERROR/ {
  for (i = 1; i <= NF; i++) {
    if ($i ~ /^host=/) {
      split($i, a, "=")
      print a[2]
    }
  }
}' app.log | sort | uniq -c | sort -nr | tee error-hosts.txt
grep -q 'api-1' error-hosts.txt

tar -czf text-processing-evidence.tgz \
  lab-user.txt wc-samples.txt \
  errors-raw.txt errors-count.txt error-codes.txt \
  top-error-codes.txt top-status-codes.txt error-hosts.txt
```

```
ls -l text-processing-evidence.tgz | tee evidence-ls.txt
test -s text-processing-evidence.tgz
```

Expected output: `TIMEOUT` ranks highest among error codes; `502` appears in status ranks; `text-processing-evidence.tgz` is non-empty.

Validation steps

- [ ] `errors-count.txt` is `5`
- [ ] `top-error-codes.txt` shows `TIMEOUT` with the highest count
- [ ] `top-status-codes.txt` includes `502` and `200`
- [ ] `error-hosts.txt` mentions `api-1`
- [ ] `text-processing-evidence.tgz` exists under `~/rebash-linux/lab08`

Common errors and fixes

Error	Cause	Fix
`sed` outputs whole lines, not codes	Regex did not match	Use the exact `sed -E` from Task 2; check sample log spelling
Wrong status field in awk	Different log format	Count fields with `awk '{print NF; exit}' access.log` and adjust `$9`
`uniq -c` shows all counts as 1	Forgot to `sort` first	Always `sort` before `uniq`
Empty `errors-raw.txt`	Pattern case mismatch	Sample uses `ERROR` in uppercase — match that

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write an executable script `~/rebash-linux/lab08/incident-summary.sh` that takes a log path as `$1`, prints (1) ERROR count, (2) top three `code=` values, and (3) top three `host=` values for ERROR lines. Run it on `app.log`, save stdout to `challenge-summary.txt`, and `chmod +x` the script. Keep it under 40 lines.

Learning outcomes

- Filtered and counted ERROR lines with grep
- Extracted fields with sed and awk
- Ranked frequencies with sort/uniq
- Built an evidence archive suitable for a change ticket

Cleanup

```
cd ~/rebash-linux/lab08
# Keep the evidence archive if you want it; otherwise remove lab artefacts:
# rm -f app.log access.log *.txt incident-summary.sh text-processing-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab08/` with evidence files
- [ ] You can explain when to use grep vs sed vs awk
- [ ] You know why `sort` must come before `uniq -c`
- [ ] You can describe the risk of `sed -i` without a backup on a real config

Code Walkthrough

In real servers, text work for ops usually follows this order:

1. **Sample first** — `head`, `tail`, or a time-bounded export; do not pipe multi-GB files blindly
2. **Filter early** — `grep` / `grep -v` to cut noise before heavier tools
3. **Set separators** — `-F` in awk, `-d` in cut; do not guess columns
4. **Keep backups** — `sed -i.bak` or write to a new file, then validate
5. **Save evidence** — `tee` or redirect into ticket attachments

Later you may wrap pipelines in scripts or CI jobs. People still review the regex and the field numbers.

Security Considerations

- Do not paste secrets from logs into tickets or chat — redact tokens and passwords
- Restrict who can read production log directories (`/var/log`, journal)
- Prefer read-only copies of logs in the lab; avoid editing live configs with ad-hoc sed
- Be careful with `sudo grep` over home directories of other users
- In CI, fail closed on secret-like patterns (API keys) rather than only on ERROR strings

Common Mistakes

Running `sed -i` with no backup on a live config

A bad substitute can break SSH or the application on the next reload. **Fix:** use `sed -i.bak`, test on a copy, and keep a console session open when changing remote access configs.

Forgetting to sort before `uniq -c`

`uniq` only collapses **adjacent** duplicate lines. **Fix:** `sort file | uniq -c | sort -nr`.

Using plain `xargs` on file names from `find`

Names with spaces split into wrong arguments. **Fix:** `find … -print0 | xargs -0 …`.

Recursive grep through huge trees

You waste time and I/O. **Fix:** narrow the path, add `--exclude-dir`, and prefer known log files.

Best Practices

- Prefer `grep -E` / `sed -E` when extended regex makes the pattern clearer
- Use `LC_ALL=C` in scripts for stable sorting
- Name evidence files by intent (`top-error-codes.txt`), not `out1.txt`
- Put reusable pipelines in small scripts with `"$1"` arguments
- Document the field numbers you assumed for each log format

Troubleshooting

Symptom	Likely cause	Fix
No matches for a pattern you see in the file	Wrong case or Windows CRLF	Try `grep -i`; strip `\r` with `tr -d '\r'`
awk prints wrong column	Different whitespace / quoted fields	Inspect with `awk '{print NF,$0}' | head`
`sed` changes nothing	Pattern does not match	Test one line with `echo … | sed …`
Pipeline hangs	Waiting on stdin	Pass a file argument or end of input
Permission denied on `/var/log`	Not in the right group	Use a copied log, or ask for read access — do not chmod production logs casually

Summary

grep, sed, and awk turn logs and configs into answers: find, edit, and report. Practise composing short pipelines, keep backups for edits, and save command output as ticket evidence. Next, learn how to inspect and control running programmes in Process Management.

Interview Questions

INTERVIEW PRACTICE

1. When would you use grep, when sed, and when awk? Give one example each from production ops.

Answer

Use **grep** to select or exclude lines (`grep -n ERROR app.log`). Use **sed** to transform text in a stream or file (`sed -i.bak 's/old/new/'`). Use **awk** when you need fields or a small report (`awk '{print $9}' access.log | sort | uniq -c`). Interviewers want clear job fit, not tool loyalty.

2. Why must you sort before `uniq -c`, and how do you get a top-N list?

Answer

`uniq` only merges **neighbouring** duplicate lines. Without `sort`, counts are wrong. Top-N pattern: `sort file | uniq -c | sort -nr | head -n 10`. This is a standard incident summary for error codes or status codes.

3. A junior engineer runs `sed -i 's/…/…/' /etc/nginx/nginx.conf` and Nginx fails to start. What went wrong in process, and how do you prevent it?

Answer

They edited in place with **no backup** and no config test. Recover from backup or package reinstall / version control, then `nginx -t` before reload. Prevent with `sed -i.bak`, edit a copy, validate syntax, and prefer configuration management for lasting changes.

4. How do you safely run grep over many files whose names may contain spaces?

Answer

Use null-delimited paths: `find dir -type f -name '*.log' -print0 | xargs -0 grep -H PATTERN`. Without `-print0` / `-0`, spaces split into fake arguments and you miss files or hit the wrong ones.

5. Access log field numbers differ between formats. How do you avoid hard-coding the wrong column in awk?

Answer

Inspect first: `head -n 1 access.log` and `awk '{print NF; exit}'`. Prefer formats with clear delimiters (`-F`), or match labelled tokens (`host=`) instead of only `$N` when the line is irregular. Document the assumed format next to the one-liner in the runbook.

6. What is a production risk of `grep -R` from `/` or from a large container layer directory?

Answer

High I/O and CPU, long delays, and accidental matches inside binaries or secrets files. Narrow the search path, skip binaries (`-I`), exclude noisy directories, and search known log locations first.

7. How would you prove in a change ticket that TIMEOUT was the top application error after a deploy?

Answer

Attach pipeline output: ERROR filter, extracted `code=` values, and `sort | uniq -c | sort -nr` showing TIMEOUT first — plus the time range of the log sample. Evidence beats “we think it was timeouts”.

Process Management
MODULE 6 · PROCESSES

Chapter 9. Process Management

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/process-management/

Overview

Every command you run becomes a **process**: a running programme with a Process ID (PID), a parent, an environment, and resource use (CPU, memory, open files). On a busy cloud virtual machine (VM), knowing how to **see**, **stop**, and **prioritise** processes is basic Site Reliability Engineering (SRE) hygiene.

You inspect with `ps`, `top`, or `htop`. You send **signals** with `kill` and `pkill` (prefer a polite `TERM` before a forced `KILL`). Shell **job control** (`jobs`, `fg`, `bg`, `Ctrl-Z`) manages work tied to your terminal. **nice** / **renice** adjust CPU scheduling priority. **nohup** keeps a command alive after logout for ad-hoc work — but long-running production work should use **systemd** services (next tutorials), which add restart policy and journal logs.

Runaway processes burn CPU budget and money on cloud VMs. Stuck deploys and zombie parents block releases. Jumping straight to `kill -9` can leave locks and half-written data. In production you measure first (`ps`, load, memory), signal carefully, and move durable workloads under a supervisor. This tutorial builds that judgement on a practice VM.

This is **Tutorial 9** in **Module 6: Process Management** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, SRE, and platform engineers. By the end, you will have started, inspected, reniced, and stopped lab processes with saved evidence.

Prerequisites

- Text Processing with grep, sed, and awk
- A **practice Ubuntu 22.04/24.04 VM** with a normal user account
- Optional: `htop` (`sudo apt install htop`) — not required for the lab

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain PID, parent process, signals (`TERM` vs `KILL`), and niceness
- [ ] Inspect processes with `ps` and read useful columns
- [ ] Start background work, use job control, and stop processes cleanly
- [ ] Adjust priority with `nice` / `renice` and state when systemd is better than `nohup`
- [ ] Complete the lab under `~/rebash-linux/lab09` with evidence files

Architecture

User commands and services become processes scheduled by the kernel. Operators observe them, send signals, and optionally adjust priority — or hand long-running work to systemd.

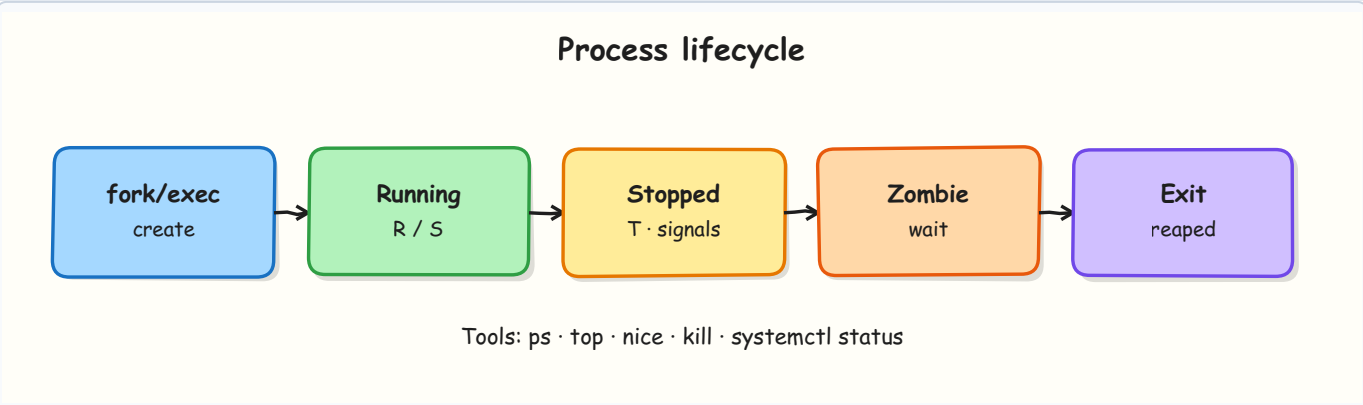

Figure 9.1: Process Management

Theory

What it is

A **process** is an instance of a running programme. Important fields include PID, Parent PID (PPID), user, state (running, sleeping, zombie), and resource use.

Tool	Role
`ps`	Point-in-time snapshot
`top` / `htop`	Live interactive view
`kill` / `pkill`	Send signals by PID or name
`jobs` / `fg` / `bg`	Shell job control
`nice` / `renice`	CPU scheduling priority
`nohup`	Survive terminal hangup (ad hoc)
systemd service	Supervised long-running work

```
ps -eo pid,ppid,user,stat,pcpu,pmem,cmd --sort=-pcpu | head
```

Why it matters

High CPU, memory leaks, and stuck batch jobs all show up as process problems. Knowing the difference between **TERM** (ask to exit cleanly) and **KILL** (force stop, no cleanup) prevents data corruption. Knowing the difference between an interactive shell job and a **systemd** unit prevents "I closed my laptop and the migration died". On shared hosts, lowering the priority of heavy batch work with `nice` protects interactive control-plane tools.

How it works

1. **Inspect** — `ps aux` or `ps -ef`; filter with `pgrep -a name` or `ps … | grep`.
2. **Signal** — `kill -TERM PID` (signal 15) first; wait; only then `kill -KILL PID` (signal 9) if needed. `pkill -f pattern` matches the command line — use narrow patterns.
3. **Jobs** — `command &` backgrounds; `jobs` lists; `Ctrl-Z` suspends; `bg` / `fg` resume.
4. **Priority** — niceness from about **-20** (more CPU favour) to **19** (less). Unprivileged users can usually only **increase** niceness (make themselves nicer / lower priority).
5. **Survive logout** — `nohup cmd &` ignores hangup and often writes `nohup.out`. Prefer a systemd unit for anything that must restart and log properly.

Signal	Number	Meaning
TERM	15	Graceful stop — try first
INT	2	Interrupt (like Ctrl-C)
HUP	1	Hangup; many daemons reload config
KILL	9	Force stop — cannot be caught

Key concepts and comparisons

Pattern	Prefer when	Avoid when
`kill -TERM` then wait	App can flush and exit	You already know the process ignores TERM
`kill -KILL`	Process is hung after TERM	First reaction to every problem
`nohup` / `screen` / `tmux`	One-off admin task	Production API or always-on worker
systemd unit	Needs restart, deps, journal	Quick interactive experiment
Higher niceness (e.g. 10)	Batch / compile on shared VM	Latency-critical request path

Common pitfalls

- Jumping to `kill -9` and leaving locks or half-written files.
- Broad `pkill -f` patterns that kill the wrong processes (including your SSH session tooling).
- Relying on `nohup` for production workloads that need restart and logs.
- Misreading **load average** without checking run queue, I/O wait, and steal time on cloud VMs.
- Renicing critical daemons without understanding latency impact.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, start lab worker processes, inspect them with `ps`, adjust niceness, stop them with `TERM` (and prove they exited), and save evidence under `~/rebash-linux/lab09`.

Prerequisites

- Ubuntu 22.04/24.04 with bash
- Packages: `procps` (provides `ps`, `kill`, `pgrep` — already on Ubuntu)
- No sudo required unless you choose to install `htop`

Lab environment

Workspace: `~/rebash-linux/lab09`

```
mkdir -p ~/rebash-linux/lab09 && cd ~/rebash-linux/lab09
set -euo pipefail
whoami | tee lab-user.txt
ps -p $$ -o pid,ppid,cmd | tee shell-ps.txt
```

Expected output: `lab-user.txt` and `shell-ps.txt` exist; your shell PID is listed.

Real-world scenario

A batch “report” job was started in SSH and is still consuming CPU after the engineer disconnected. You need to find the process, confirm it is yours, lower its priority if it must finish, or stop it cleanly with `TERM` and keep proof for the ticket. You practise that path with disposable `sleep` workers (safe stand-ins for long jobs).

Step-by-step tasks

Task 1 – Start workers and inspect with ps

```
cd ~/rebash-linux/lab09
set -euo pipefail

# Two disposable workers (sleep is safe and easy to spot)
sleep 3600 &
echo $! | tee worker1.pid
sleep 3600 &
echo $! | tee worker2.pid

W1=$(cat worker1.pid)
W2=$(cat worker2.pid)
ps -p "$W1","$W2" -o pid,ppid,user,ni,stat,cmd | tee workers-ps.txt
pgrep -af 'sleep 3600' | tee workers-pgrep.txt
grep -q "$W1" workers-ps.txt
grep -q "$W2" workers-ps.txt
```

Expected output: both PIDs appear in `workers-ps.txt` with command `sleep 3600`.

Task 2 – Renice and job-control style background proof

```
cd ~/rebash-linux/lab09
set -euo pipefail

W1=$(cat worker1.pid)

# Lower CPU priority (higher nice value) for worker1
renice +10 -p "$W1" | tee renice-out.txt
ps -p "$W1" -o pid,ni,cmd | tee worker1-nice.txt
awk 'NR==2 {exit !($2 == 10)}' worker1-nice.txt

# Start a third worker with nice from the beginning
nice -n 15 sleep 3600 &
echo $! | tee worker3.pid
W3=$(cat worker3.pid)
ps -p "$W3" -o pid,ni,cmd | tee worker3-nice.txt
awk 'NR==2 {exit !($2 == 15)}' worker3-nice.txt
```

Expected output: worker1 niceness is `10`; worker3 niceness is `15`.

Task 3 – Graceful stop with TERM and evidence pack

```
cd ~/rebash-linux/lab09
set -euo pipefail

W1=$(cat worker1.pid)
W2=$(cat worker2.pid)
W3=$(cat worker3.pid)

kill -TERM "$W1" "$W2" "$W3"
# Brief wait for exit
sleep 1

# These PIDs must be gone
if ps -p "$W1","$W2","$W3" >/dev/null 2>&1; then
  echo "ERROR: workers still running" >&2
  ps -p "$W1","$W2","$W3" -o pid,stat,cmd || true
  exit 1
fi
echo "all lab workers stopped" | tee stop-ok.txt

# Demonstrate KILL is last resort (start and force-stop one short worker)
sleep 3600 &
echo $! | tee worker-kill.pid
WK=$(cat worker-kill.pid)
kill -KILL "$WK"
sleep 0.5
if ps -p "$WK" >/dev/null 2>&1; then
  echo "ERROR: KILL failed" >&2
  exit 1
fi
echo "KILL demo ok" | tee kill-demo.txt

tar -czf process-evidence.tgz \
  lab-user.txt shell-ps.txt \
  worker1.pid worker2.pid worker3.pid worker-kill.pid \
  workers-ps.txt workers-pgrep.txt \
  renice-out.txt worker1-nice.txt worker3-nice.txt \
  stop-ok.txt kill-demo.txt
ls -l process-evidence.tgz | tee evidence-ls.txt
test -s process-evidence.tgz
```

Expected output: `stop-ok.txt` and `kill-demo.txt` exist; no lab `sleep 3600` workers remain; archive is non-empty.

Validation steps

- [ ] You captured PIDs and `ps` output for lab workers
- [ ] `renice` / `nice` values show in `worker1-nice.txt` and `worker3-nice.txt`
- [ ] Workers were stopped with `TERM` and confirmed gone
- [ ] `process-evidence.tgz` exists under `~/rebash-linux/lab09`

Common errors and fixes

Error	Cause	Fix
`renice: failed to set priority`	Trying to lower niceness (boost priority) without root	Use positive niceness (`+10`) as in the lab
`No such process`	Already exited / wrong PID	Re-read `worker*.pid`; do not reuse old PIDs
`pkill` killed unexpected processes	Pattern too broad	Match a unique command line; prefer exact PID from `pgrep -a`
Workers still listed after TERM	Slow exit / ignored signal	Wait briefly; then `kill -KILL` only for that PID

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `~/rebash-linux/lab09/graceful-stop.sh` that: (1) starts `sleep 120` in the background, (2) writes its PID to `challenge.pid`, (3) sends `TERM`, (4) waits up to 5 seconds, (5) exits `0` only if the PID is gone (else sends `KILL` and exits `1`). Run it once and save stdout/stderr to `challenge-run.txt`.

Learning outcomes

- Started and identified processes with `ps` / `pgrep`
- Adjusted niceness with `nice` / `renice`
- Stopped processes with `TERM` first and used `KILL` only as a demo last resort
- Saved process evidence for a ticket

Cleanup

```
cd ~/rebash-linux/lab09
set -euo pipefail
# Stop any leftover lab sleeps matching our pattern
pkill -f 'sleep 3600' 2>/dev/null || true
pkill -f 'sleep 120' 2>/dev/null || true
# Keep evidence if you want; otherwise:
# rm -f *.pid *.txt process-evidence.tgz graceful-stop.sh
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab09/` with evidence files
- [ ] You can explain TERM vs KILL and why TERM comes first
- [ ] You can explain when to use systemd instead of `nohup`
- [ ] You know that unprivileged users usually cannot renice *down* (boost priority)

Code Walkthrough

In real servers, process work usually follows this order:

1. **Measure** — `ps`, `top`/`htop`, load, memory — before killing anything
2. **Identify owner and command** — confirm it is safe to stop
3. **TERM, wait, verify** — then KILL only if needed
4. **Narrow matches** — PID or careful `pkill -f`
5. **Supervise durable work** — systemd units over `nohup` for production

Keep the blast radius small: one PID, one service, one node — then widen if needed.

Security Considerations

- Only kill processes you understand — wrong PID can stop SSH, databases, or agents
- Restrict who has rights to signal other users' processes
- Do not run untrusted binaries with `nohup` and forget them
- Audit unexpected persistent processes after incidents
- Prefer service accounts and systemd for apps — not root shells left in `screen`

Common Mistakes

Using `kill -9` as the first step

The process cannot clean up. **Fix:** `kill -TERM PID`, wait and re-check with `ps`, then `KILL` only if still present.

Broad `pkill -f sleep`

You may kill unrelated work. **Fix:** use the exact PID from your pidfile or a unique command line.

Leaving production jobs on `nohup`

No restart policy, weak logging, easy to forget. **Fix:** create a systemd service (next tutorial).

Misreading high load as 'need more CPU'

Load can be I/O wait or cloud steal time. **Fix:** check `vmstat` / `iostat` and process states, not only the load number.

Best Practices

- Capture `ps` / `pgrep` output before and after interventions
- Prefer service restarts (`systemctl restart`) over killing random children of a supervised service
- Use higher niceness for compile/batch jobs on shared bastions
- Document PIDs and commands in the incident ticket
- Practise graceful stop scripts before you need them in production

Troubleshooting

Symptom	Likely cause	Fix
Process in `Z` state (zombie)	Parent not reaping	Fix/restart the parent; zombie itself holds almost no resources
`Operation not permitted` on kill	Not your process / privilege	Use sudo only when appropriate; check owner with `ps`
Process ignores TERM	Custom signal handling / stuck in uninterruptible I/O (`D`)	Investigate I/O; KILL if policy allows; check disks
CPU high after kill	Wrong process / respawn by supervisor	Check systemd/container runtime; stop the unit, not only one child
Job disappears after SSH logout	SIGHUP to session	Use systemd, or carefully `nohup` / `tmux` for one-off work

Summary

Process management is how you see, prioritise, and stop running work safely. Prefer `TERM` before `KILL`, use niceness for batch load, and move durable workloads to systemd. Next: systemd Services and journalctl.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between `kill -15` (TERM) and `kill -9` (KILL), and which should you try first?

Answer

TERM asks the process to exit so it can flush buffers and release locks. **KILL** forces immediate stop and cannot be caught — no cleanup. Always try TERM first, wait and verify with `ps`, then use KILL only if the process is hung and policy allows.

2. A process keeps coming back after you kill it. What do you check next?

Answer

A **supervisor** is restarting it: systemd, a container runtime, Kubernetes, or a parent script. Find the unit or parent (`systemctl status`, `ps -ef` / PPID tree) and stop or fix the supervisor — do not only kill children in a loop.

3. What does niceness mean, and can a normal user make their process higher priority?

Answer

Niceness influences CPU scheduling: **higher nice** means **lower** priority. Unprivileged users can usually only **increase** niceness (become "nicer"). Lowering niceness (boosting priority) typically needs root. Use higher nice for batch jobs on shared hosts.

4. When is `nohup` acceptable, and when should you use a systemd service instead?

Answer

`nohup` is fine for a **one-off** admin command that must survive logout. Use **systemd** when you need start-at-boot, restart on failure, dependencies, and journal logging — i.e. almost all production long-running work.

5. How do you find and stop a process safely when you only remember part of the command line?

Answer

List candidates with `pgrep -af 'unique-substring'` or `ps -ef | grep`, confirm the PID and user, then `kill -TERM` that PID. Avoid vague patterns like `pkill -f java` on a host that runs many Java apps.

6. Load average is high but `top` shows low %CPU. What else might be going on?

Answer

Processes may be in **uninterruptible I/O** (`D`), or the host may have high **I/O wait** or cloud **steal** time. Check `vmstat`, `iostat`, disk health, and whether tasks are blocked on storage — not only user CPU %.

7. How would you prove in a ticket that you stopped a runaway job cleanly?

Answer

Attach before/after `ps` or `pgrep` output, the PID, the signal used (`TERM`), and confirmation the PID is gone. If you had to use `KILL`, say why TERM failed. Evidence beats "I killed it".

systemd Services and journalctl
MODULE 7 · SERVICES & BOOT

Chapter 10. systemd Services and journalctl

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/systemd-services-and-journalctl/

Overview

On almost every modern Linux cloud image, **systemd** is process ID 1 (PID 1). It starts **units** (services, sockets, timers, mounts), tracks dependencies, places processes in control groups (cgroups), and records structured logs through **journald**. Day-to-day you use **systemctl** to control services and **journalctl** to read their logs.

If a service will not start after a deploy, or disappears after reboot because it was never **enabled**, systemd is the control plane you debug. Editing vendor unit files under `/lib/systemd/system` (or `/usr/lib`) loses changes on package upgrade — the supported path is a unit or **drop-in** under `/etc/systemd/system`. The journal is often faster evidence than grepping flat files alone, especially with `-u` and `--since`.

In production, good habits are: `daemon-reload` after unit changes, `systemctl status` + `journalctl -u` before guessing, `enable --now` when you need boot persistence, and hardening directives (`NoNewPrivileges=`, `ProtectSystem=`) where they fit. This tutorial builds a small lab service you can start, log, override, and remove cleanly.

This is **Tutorial 10** in **Module 7: Services & Boot** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Process Management
- A **practice Ubuntu 22.04/24.04 VM** where you have `sudo`
- Do **not** run this lab on a shared production server

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain unit files, vendor vs `/etc` locations, and drop-ins
- [ ] Create a simple systemd service, reload, start, and enable it
- [ ] Query logs with `journalctl -u` and time filters
- [ ] Override a setting with a drop-in without editing the vendor/lab unit body incorrectly
- [ ] Remove the lab unit cleanly and save evidence under `~/rebash-linux/lab10`

Architecture

systemd sits as PID 1: it activates units, supervises processes, and records journal logs that operators query with journalctl.

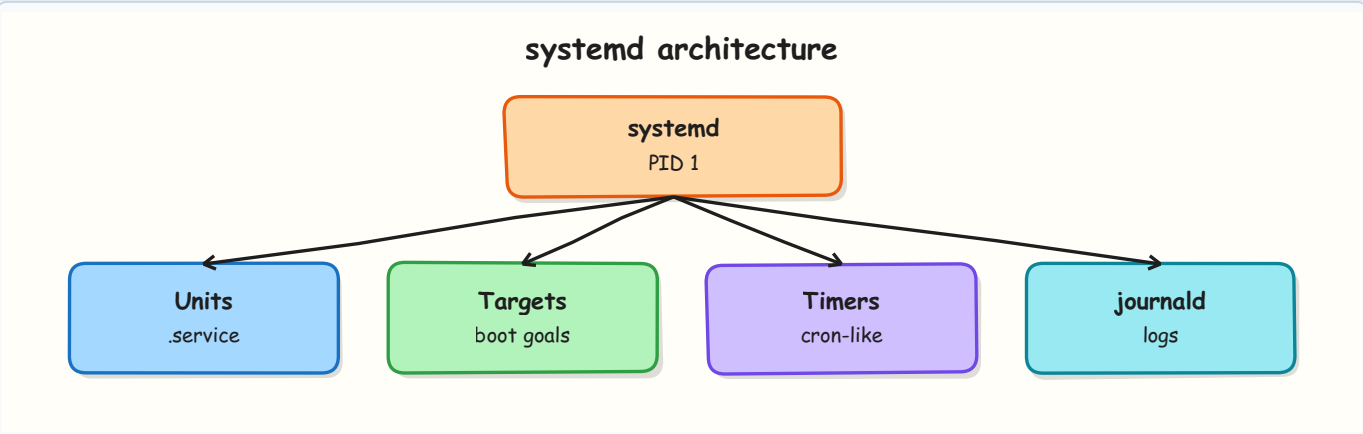

Figure 10.1: systemd Services and journalctl

Theory

What it is

A **unit** describes something systemd can manage. A **service** unit usually has an `[Unit]` section (description, ordering), a `[Service]` section (`ExecStart=`, user, restart policy), and an `[Install]` section (`WantedBy=` for boot).

Action	Effect
`start` / `stop`	Runtime only
`enable` / `disable`	Boot persistence (symlinks)
`reload`	Re-read app config if the unit supports it
`restart`	Stop then start
`edit` / drop-in	Override without rewriting the whole unit
`mask`	Make start impossible (stronger than disable)

```
systemctl status cron.service
systemctl cat cron.service
journalctl -u cron.service -n 20 --no-pager
```

Why it matters

Cloud VMs boot into a graph of units. Apps that were started only with `nohup` or a manual shell die on reboot or after crash without restart. Wrong working directory, missing `User=`, or a failed `ExecStart` shows up immediately in `systemctl status` and the journal. Teams that treat units as code (reviewed drop-ins) recover faster than teams that hand-edit vendor files.

How it works

1. **Write** a unit under `/etc/systemd/system/name.service` (local) or add a drop-in under `name.service.d/*.conf`.
2. **`systemctl daemon-reload`** — systemd re-reads unit files.
3. **`systemctl start name`** — start now; **`enable`** — start on boot; **`enable --now`** — both.
4. **Inspect** — `systemctl status`, `systemctl cat`, `systemctl show`.
5. **Logs** — `journalctl -u name -e --no-pager`; add `--since '10 min ago'` or `-p err`.

Persist journals under `/var/log/journal` when you need logs after reboot (many cloud images already do this).

Query	Example
Unit	`journalctl -u rebash-lab.service -e`
Since	`journalctl --since '1 hour ago'`
Priority	`journalctl -p err..alert -b`
Follow	`journalctl -f -u rebash-lab.service`

Key concepts and comparisons

Location	Use for
`/lib/systemd/system` or `/usr/lib/systemd/system`	Vendor/package units — do not edit in place
`/etc/systemd/system`	Local units and overrides
`/etc/systemd/system/foo.service.d/*.conf`	Drop-in fragments

Directive	Typical meaning
`Type=simple`	Main process is `ExecStart` (default pattern)
`Type=oneshot`	Short task; often `RemainAfterExit=yes`
`Restart=on-failure`	Restart when the process fails
`After=` / `Requires=`	Ordering and hard dependency

Common pitfalls

- Editing `/lib/systemd/system` and losing changes on upgrade.
- Forgetting `daemon-reload` after adding or changing units.
- `enable` without noticing the unit is **failed** (`systemctl --failed`).
- Assuming the journal is persistent when the host only keeps a volatile journal.
- Using `restart` when `reload` would be gentler for the application.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Create a local `rebash-lab.service` that writes a heartbeat line on a schedule-friendly loop, manage it with systemctl, read logs with journalctl, add a drop-in, and save evidence under `~/rebash-linux/lab10`.

Prerequisites

- Ubuntu 22.04/24.04 with systemd and sudo
- Packages: `systemd` (default on Ubuntu)

Lab environment

Workspace: `~/rebash-linux/lab10`

```
mkdir -p ~/rebash-linux/lab10 && cd ~/rebash-linux/lab10
set -euo pipefail
whoami | tee lab-user.txt
systemctl is-system-running | tee systemd-state.txt || true
test -n "$(command -v systemctl)"
sudo -n true 2>/dev/null || sudo -v
```

Expected output: `systemctl` works; sudo is available (password once if needed).

Real-world scenario

Your team needs a tiny “sidecar” style helper on an Ubuntu app VM: a supervised process that appends a heartbeat to a file and appears in the journal. Security asks that it run as a normal service (not a forgotten `nohup`), that you can show logs with `journalctl -u`, and that overrides use a drop-in. You build that on a practice VM and keep proof for the change ticket.

Step-by-step tasks

Task 1 – Install unit and start the service

```
cd ~/rebash-linux/lab10
set -euo pipefail

# Working directory and script owned by your user; service will run as you
mkdir -p "$HOME/rebash-linux/lab10/run"
cat > "$HOME/rebash-linux/lab10/run/heartbeat.sh" << 'EOF'
#!/bin/bash
```

```
set -euo pipefail
LOG_DIR="${HOME}/rebash-linux/lab10/run"
mkdir -p "$LOG_DIR"
while true; do
  ts="$(date -Is)"
  echo "${ts} rebash-lab heartbeat ok" | tee -a "${LOG_DIR}/heartbeat.log"
  sleep 15
done
EOF
chmod +x "$HOME/rebash-linux/lab10/run/heartbeat.sh"

UNIT_USER="$(whoami)"
UNIT_HOME="$HOME"

sudo tee /etc/systemd/system/rebash-lab.service >/dev/null << EOF
[Unit]
Description=REBASH lab heartbeat service
After=network.target

[Service]
Type=simple
User=${UNIT_USER}
WorkingDirectory=${UNIT_HOME}/rebash-linux/lab10/run
ExecStart=${UNIT_HOME}/rebash-linux/lab10/run/heartbeat.sh
Restart=on-failure
RestartSec=2

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl daemon-reload
sudo systemctl enable --now rebash-lab.service
systemctl is-active rebash-lab.service | tee service-active.txt
test "$(cat service-active.txt)" = "active"
systemctl status rebash-lab.service --no-pager | tee service-status.txt
```

Expected output: `service-active.txt` is `active`; status shows the unit running.

Task 2 – journalctl evidence and heartbeat file

```
cd ~/rebash-linux/lab10
set -euo pipefail

# Wait for at least one heartbeat line
for i in 1 2 3 4 5 6; do
  if [[ -s "$HOME/rebash-linux/lab10/run/heartbeat.log" ]]; then
    break
  fi
  sleep 3
done
test -s "$HOME/rebash-linux/lab10/run/heartbeat.log"
cp "$HOME/rebash-linux/lab10/run/heartbeat.log" heartbeat-log-copy.txt

journalctl -u rebash-lab.service -n 50 --no-pager | tee journal-unit.txt
grep -q 'heartbeat ok' journal-unit.txt || grep -q 'heartbeat ok' heartbeat-log-copy.txt

systemctl show rebash-lab.service -p FragmentPath -p DropInPaths -p MainPID \
  | tee service-show.txt
```

Expected output: heartbeat log has lines; journal or log copy shows `heartbeat ok`; `MainPID` is set.

Task 3 – Drop-in override and evidence pack

```
cd ~/rebash-linux/lab10
set -euo pipefail

# Add an environment variable via drop-in (does not edit the main unit file body by hand)
sudo mkdir -p /etc/systemd/system/rebash-lab.service.d
sudo tee /etc/systemd/system/rebash-lab.service.d/10-lab.conf >/dev/null << 'EOF'
[Service]
Environment=REBASH_LAB=1
EOF

sudo systemctl daemon-reload
sudo systemctl restart rebash-lab.service
systemctl is-active rebash-lab.service | tee service-active-after-dropin.txt
test "$(cat service-active-after-dropin.txt)" = "active"

systemctl cat rebash-lab.service | tee service-cat.txt
grep -q 'REBASH_LAB=1' service-cat.txt
systemctl show rebash-lab.service -p Environment | tee service-env.txt
grep -q 'REBASH_LAB=1' service-env.txt

tar -czf systemd-service-evidence.tgz \
  lab-user.txt systemd-state.txt \
```

```
  service-active.txt service-status.txt \
  heartbeat-log-copy.txt journal-unit.txt service-show.txt \
  service-active-after-dropin.txt service-cat.txt service-env.txt
ls -l systemd-service-evidence.tgz | tee evidence-ls.txt
test -s systemd-service-evidence.tgz
```

Expected output: `systemctl cat` includes the drop-in; Environment shows `REBASH_LAB=1`; archive is non-empty.

Validation steps

- [ ] `systemctl is-active rebash-lab.service` is `active`
- [ ] Heartbeat lines exist under `~/rebash-linux/lab10/run/heartbeat.log`
- [ ] `journalctl -u rebash-lab.service` returns recent entries
- [ ] Drop-in appears in `systemctl cat`
- [ ] `systemd-service-evidence.tgz` exists

Common errors and fixes

Error	Cause	Fix
`Unit rebash-lab.service not found`	Forgot `daemon-reload`	`sudo systemctl daemon-reload`
`activating (auto-restart)` loop	Script path/permissions wrong	`journalctl -u rebash-lab -e`; fix `ExecStart` and `chmod +x`
Empty journal lines	Permissions / rate limit / wrong unit name	Confirm `-u rebash-lab.service`; check heartbeat file
Drop-in ignored	Wrong directory name	Must be `rebash-lab.service.d/*.conf` then `daemon-reload`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Add a second drop-in `20-security.conf` that sets `NoNewPrivileges=yes` for `rebash-lab.service`. Reload, restart, and prove with `systemctl show rebash-lab.service -p NoNewPrivileges` saved to `challenge-nnp.txt`. Remove this drop-in in Cleanup if you create it.

Learning outcomes

- Created and enabled a local systemd service
- Used journalctl and status for evidence
- Applied a drop-in override safely
- Packaged proof for a change ticket

Cleanup

```
cd ~/rebash-linux/lab10
set -euo pipefail

sudo systemctl disable --now rebash-lab.service 2>/dev/null || true
sudo rm -f /etc/systemd/system/rebash-lab.service
sudo rm -rf /etc/systemd/system/rebash-lab.service.d
sudo systemctl daemon-reload
sudo systemctl reset-failed rebash-lab.service 2>/dev/null || true

# Keep evidence archive if you want; optional:
# rm -rf run *.txt systemd-service-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab10/` with evidence files
- [ ] You can explain start vs enable and why drop-ins beat editing vendor units
- [ ] You can gather logs with `journalctl -u` for one service
- [ ] Cleanup removed the lab unit

Code Walkthrough

In real servers, systemd service work usually follows this order:

1. **Inspect** — `status`, `cat`, `journalctl -u` before changing
2. **Change under `/etc`** — local unit or drop-in, never vendor files in place
3. **`daemon-reload`** then start/restart
4. **Prove** — `is-active`, journal lines, application health check
5. **Least privilege** — `User=`, hardening directives, narrow sudo for restarts

Automate units with configuration management when the pattern is stable; still keep a manual recovery path.

Security Considerations

- Run services as a dedicated user, not root, when possible
- Prefer drop-ins reviewed in git over improvised production edits
- Limit who can `systemctl` manage sensitive units via policy / sudo
- Treat journal access as sensitive — it may contain secrets from apps
- Use `NoNewPrivileges=`, `ProtectSystem=`, and friends where compatible

Common Mistakes

Editing vendor units under `/lib` or `/usr/lib`

Package upgrades overwrite your changes. **Fix:** copy to `/etc` or use `systemctl edit` drop-ins.

Forgetting `daemon-reload`

systemd still uses the old unit definition. **Fix:** reload, then restart the unit.

Assuming enable means healthy

A unit can be enabled and still **failed**. **Fix:** check `systemctl --failed` and the journal after every change.

Using `restart` when `reload` is enough

Unnecessary connection drops. **Fix:** use reload when the app supports it; restart when the binary or unit must change.

Best Practices

- Name units clearly (`app-api.service`) and document `ExecStart`
- Always pair deploy changes with journal checks
- Prefer `enable --now` in runbooks when boot persistence is required
- Keep drop-ins small and commented
- Alert on failed units (`systemctl --failed`) in monitoring

Troubleshooting

Symptom	Likely cause	Fix
`status=203/EXEC`	Bad `ExecStart` path or not executable	Fix path; `chmod +x`; daemon-reload
`status=216/GROUP` or user errors	Wrong `User=` / `Group=`	Create user or fix names
No logs in journalctl	Wrong unit / permissions / stdout not captured	Confirm unit name; ensure process logs to stdout/stderr or journal
Changes ignored	No daemon-reload / wrong drop-in path	Fix path; reload; `systemctl cat`
Starts then exits	`Type=` mismatch / oneshot without RemainAfterExit	Match Type to programme behaviour

Summary

systemd services give you supervised start, stop, enable-on-boot, and journal logs. Prefer `/etc` units and drop-ins, prove with `systemctl` and `journalctl`, and clean up lab units. Next: systemd Targets, Timers, and Boot.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between `systemctl start` and `systemctl enable`?

Answer

start affects the **current** boot only (runtime). **enable** creates symlinks so the unit is pulled in on future boots. Use `enable --now` when you want both. Interviewers look for this distinction constantly.

2. Why should you not edit unit files under `/lib/systemd/system`?

Answer

Those files belong to **packages**. Upgrades overwrite them. Put local units or drop-ins under `/etc/systemd/system` (for example via `systemctl edit`) so changes survive updates and are visible in `systemctl cat`.

3. A service is in failed state after deploy. What is your first evidence path?

Answer

`systemctl status name.service` and `journalctl -u name.service -e --no-pager` (optionally `--since`). Read `ExecStart` errors, exit codes, and recent commits/config changes before restarting in a loop.

4. What is a systemd drop-in, and when do you use one?

Answer

A drop-in is a fragment under `name.service.d/*.conf` that **overrides or adds** directives without rewriting the whole unit. Use it to change `Environment=`, restart policy, or limits while keeping the vendor unit intact.

5. How do `restart` and `reload` differ?

Answer

restart stops then starts the service (new process). **reload** asks the process to re-read configuration if the unit defines `ExecReload=` and the app supports it — often gentler for listeners. If reload is not supported, restart is required.

6. How do you confirm a unit will survive reboot?

Answer

Check `systemctl is-enabled name` (should be `enabled`), inspect `[Install]` / WantedBy links, and preferably verify on a practice VM with a reboot window. `is-active` alone does not prove enablement.

7. What security hardening might you add to a simple service unit?

Answer

Run as a dedicated `User=`, set `NoNewPrivileges=yes`, consider `ProtectSystem=`, `ProtectHome=`, and tight filesystem permissions on `ExecStart`. Prove with `systemctl show` and test that the app still works after hardening.

systemd Targets, Timers, and Boot
MODULE 7 · SERVICES & BOOT

Chapter 11. systemd Targets, Timers, and Boot

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/systemd-targets-timers-and-boot/

Overview

Targets are systemd units that group other units into a desired system state — the modern replacement for old SysV **runlevels**. Common examples are `rescue.target`, `multi-user.target` (typical server), and `graphical.target`. **Timers** activate services on a calendar or after boot, often replacing cron when you already use systemd. Together with **boot analysis** tools, they answer: when does the system become ready, and when does recurring work run?

Wrong default target can pull in a desktop stack you do not want on a server, or leave apps starting before the network is really up. Timers give dependency ordering, optional random delay (jitter), and journal integration — useful on fleets compared with silent cron mail. After package updates, boot regressions show up in `systemd-analyze critical-chain` and failed dependencies.

This tutorial stays **safe**: you inspect targets and boot timing, and you create a **lab timer** you remove in cleanup. You do **not** isolate to rescue or emergency on a remote VM without console access.

This is **Tutorial 11** in **Module 7: Services & Boot** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- systemd Services and journalctl
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Comfortable with `systemctl` and `journalctl` from the previous tutorial

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain targets vs old runlevels and name common targets
- [ ] Read the default target and list units belonging to it
- [ ] Create a `.timer` + `.service` pair, enable the timer, and verify with `list-timers`
- [ ] Use `systemd-analyze` to reason about boot cost
- [ ] Remove lab timer units cleanly and keep evidence under `~/rebash-linux/lab11`

Architecture

At boot, systemd pulls in the default target and its dependencies. Timers later activate services on schedule; operators inspect both with systemctl and analyse delay with systemd-analyze.

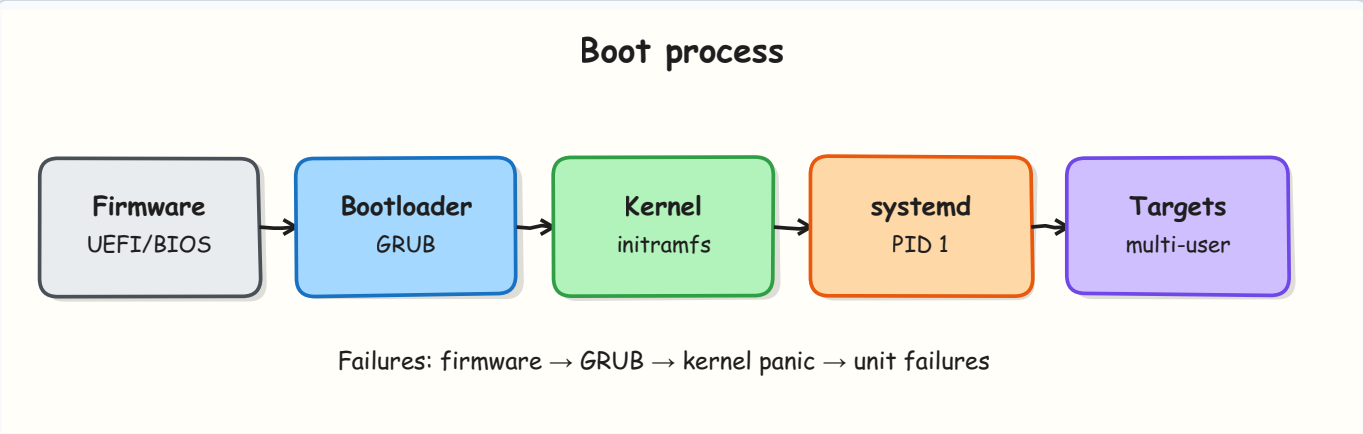

Figure 11.1: systemd Targets, Timers, and Boot

Theory

What it is

A **target** is a synchronisation point (a named goal state). A **timer** is a unit that activates another unit (usually a service) when time conditions match.

Target	Role
`rescue.target`	Minimal recovery environment
`emergency.target`	Even smaller; often root shell via sulogin
`multi-user.target`	Standard server (no GUI)
`graphical.target`	Desktop plus multi-user
`network-online.target`	Network configured (as defined by the network stack)

Scheduler	Strength
cron	Simple per-user tables; ubiquitous
systemd timer	Dependencies, journal, jitter, unit hardening

```
systemctl get-default
systemctl list-timers --all
systemd-analyze
```

Why it matters

Apps that use `After=network.target` may still start before routes and DNS work; many need `network-online.target`. Timers that were never **enabled** never run — a common "cron migration" mistake (people enable the service instead of the timer). Slow boot after an agent install shows up in `systemd-analyze blame` long before users open tickets.

How it works

1. **Default target** — `systemctl get-default` (often `graphical.target` or `multi-user.target`). Change only with care: `systemctl set-default multi-user.target`.
2. **Boot graph** — systemd activates dependencies of the default target after early sysinit and filesystem work.
3. **Timers** — pair `foo.timer` with `foo.service`. Calendar: `OnCalendar=*-*-* 02:30:00`. Monotonic: `OnBootSec=5min`. Enable the **timer**: `systemctl enable --now foo.timer`.
4. **Inspect** — `systemctl list-timers`, `systemctl status foo.timer`, `journalctl -u foo.service`.
5. **Analyse** — `systemd-analyze`, `systemd-analyze blame`, `systemd-analyze critical-chain`.

`systemctl isolate some.target` switches the running system toward that target — **disruptive**. Do not isolate rescue/emergency on a remote cloud VM without serial/console access planned.

Key concepts and comparisons

Timer field	Meaning
`OnCalendar=`	Wall-clock schedule
`OnBootSec=`	Time after boot
`OnUnitActiveSec=`	Time after the unit last activated
`RandomizedDelaySec=`	Jitter to avoid thundering herd
`Persistent=true`	Catch up missed runs (calendar timers)

Pattern	Prefer when	Avoid when
systemd timer	Needs deps, journal, hardening	One-off user reminder (cron.d may be enough)
cron	Simple user crontab already standard	Complex dependency on other units
`network-online.target`	App needs real connectivity	App only needs local sockets

Common pitfalls

- Isolating rescue/emergency remotely without console access.
- Using `After=network.target` when the app needs `network-online.target`.
- Enabling the **service** instead of the **timer**.
- Timezone surprises — timers use the system timezone unless configured otherwise.
- Ignoring long `systemd-analyze blame` entries that delay SSH readiness.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Inspect the default target and boot analysis, create a lab oneshot service activated by a timer, prove it with `list-timers` and journal output, and save evidence under `~/rebash-linux/lab11`.

Prerequisites

- Ubuntu 22.04/24.04 with systemd and sudo
- Do not change the default target permanently on a shared machine

Lab environment

Workspace: `~/rebash-linux/lab11`

```
mkdir -p ~/rebash-linux/lab11 && cd ~/rebash-linux/lab11
set -euo pipefail
whoami | tee lab-user.txt
systemctl get-default | tee default-target.txt
sudo -n true 2>/dev/null || sudo -v
```

Expected output: `default-target.txt` shows something like `multi-user.target` or `graphical.target`.

Real-world scenario

Your platform team wants a small housekeeping job every few minutes on app VMs — with journal logs and a clear unit name — instead of an undocumented crontab line. You prototype a timer + oneshot service on a practice VM, show `list-timers` and one successful run, and remove it when the experiment ends.

Step-by-step tasks

Task 1 – Inspect targets and boot analysis

```
cd ~/rebash-linux/lab11
set -euo pipefail

systemctl get-default | tee default-target.txt
systemctl list-units --type=target --no-pager | tee targets-list.txt
systemctl status multi-user.target --no-pager | tee multi-user-status.txt || true
```

```
systemd-analyze | tee analyze.txt
systemd-analyze blame | head -n 20 | tee analyze-blame.txt
systemd-analyze critical-chain | tee analyze-critical-chain.txt || true

grep -E 'Startup finished|multi-user|graphical' analyze.txt default-target.txt || true
test -s analyze-blame.txt
```

Expected output: default target recorded; `analyze.txt` / blame output non-empty (wording varies by distro version).

Task 2 – Create oneshot service + timer

```
cd ~/rebash-linux/lab11
set -euo pipefail

mkdir -p "$HOME/rebash-linux/lab11/run"
UNIT_USER="$(whoami)"
UNIT_HOME="$HOME"
STAMP="${UNIT_HOME}/rebash-linux/lab11/run/timer-stamp.log"

# Oneshot service: append one line then exit
sudo tee /etc/systemd/system/rebash-lab-timer.service >/dev/null << EOF
[Unit]
Description=REBASH lab timer oneshot

[Service]
Type=oneshot
User=${UNIT_USER}
ExecStart=/bin/bash -c '/bin/echo "\$(date -Is) rebash-lab-timer fired" >> ${STAMP}'
EOF

# Timer: first run soon after enable, then every 2 minutes
sudo tee /etc/systemd/system/rebash-lab-timer.timer >/dev/null << 'EOF'
[Unit]
Description=REBASH lab timer schedule

[Timer]
OnBootSec=1min
OnUnitActiveSec=2min
AccuracySec=1s
Unit=rebash-lab-timer.service

[Install]
WantedBy=timers.target
EOF

sudo systemctl daemon-reload
sudo systemctl enable --now rebash-lab-timer.timer
systemctl is-active rebash-lab-timer.timer | tee timer-active.txt
test "$(cat timer-active.txt)" = "active"

systemctl list-timers --all | tee list-timers.txt
grep -q 'rebash-lab-timer.timer' list-timers.txt
```

Expected output: timer is `active`; `list-timers.txt` includes `rebash-lab-timer.timer`.

Task 3 – Trigger once, prove journal, pack evidence

```
cd ~/rebash-linux/lab11
set -euo pipefail

# Do not wait for the calendar — start the service once now
sudo systemctl start rebash-lab-timer.service
sleep 1
test -s "$HOME/rebash-linux/lab11/run/timer-stamp.log"
cp "$HOME/rebash-linux/lab11/run/timer-stamp.log" timer-stamp-copy.txt
grep -q 'rebash-lab-timer fired' timer-stamp-copy.txt

journalctl -u rebash-lab-timer.service -n 20 --no-pager | tee journal-timer-service.txt
systemctl status rebash-lab-timer.timer --no-pager | tee timer-status.txt

# Show relationship: timer watches the service unit
systemctl cat rebash-lab-timer.timer | tee timer-cat.txt
grep -q 'OnUnitActiveSec' timer-cat.txt

tar -czf targets-timers-evidence.tgz \
  lab-user.txt default-target.txt targets-list.txt \
  analyze.txt analyze-blame.txt analyze-critical-chain.txt \
  timer-active.txt list-timers.txt timer-stamp-copy.txt \
  journal-timer-service.txt timer-status.txt timer-cat.txt
ls -l targets-timers-evidence.tgz | tee evidence-ls.txt
test -s targets-timers-evidence.tgz
```

Expected output: stamp file has a fired line; archive is non-empty.

Validation steps

- [ ] `default-target.txt` exists
- [ ] `systemd-analyze` output captured
- [ ] `rebash-lab-timer.timer` appears in `list-timers`
- [ ] Stamp log shows at least one fire
- [ ] `targets-timers-evidence.tgz` exists

Common errors and fixes

Error	Cause	Fix
Timer active but stamp empty	Only waited on schedule	`systemctl start rebash-lab-timer.service` as in Task 3
`Unit not found`	No daemon-reload	`sudo systemctl daemon-reload`
Enabled service but nothing runs	Enabled `.service` instead of `.timer`	`enable --now …timer`
Permission denied writing stamp	Path/user mismatch	Match `User=` and directory ownership

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Add `RandomizedDelaySec=30` to the lab timer via a drop-in directory `rebash-lab-timer.timer.d/10-jitter.conf`, daemon-reload, restart the timer, and save `systemctl cat rebash-lab-timer.timer` to `challenge-timer-cat.txt` showing the jitter. Remove the drop-in in Cleanup.

Learning outcomes

- Inspected default target and boot analysis
- Created and enabled a systemd timer + oneshot service
- Proved a run with stamp file and journal
- Packaged evidence for a change ticket

Cleanup

```
cd ~/rebash-linux/lab11
set -euo pipefail

sudo systemctl disable --now rebash-lab-timer.timer 2>/dev/null || true
sudo rm -f /etc/systemd/system/rebash-lab-timer.timer
sudo rm -f /etc/systemd/system/rebash-lab-timer.service
sudo rm -rf /etc/systemd/system/rebash-lab-timer.timer.d
sudo systemctl daemon-reload
sudo systemctl reset-failed rebash-lab-timer.service 2>/dev/null || true

# Optional: rm -rf run *.txt targets-timers-evidence.tgz
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab11/` with evidence files
- [ ] You can explain target vs timer vs service
- [ ] You know to enable the timer, not only the service
- [ ] You understand why remote `isolate rescue.target` is dangerous without console access

Code Walkthrough

In real servers, boot and schedule work usually follows this order:

1. **Inspect default target and failed units** before changing boot behaviour
2. **Prefer timers under `/etc`** with clear service oneshots for housekeeping
3. **Enable the timer**; verify with `list-timers`
4. **Use journalctl** on the service unit for proof
5. **Analyse boot** after agent installs (`blame` / `critical-chain`)

Keep rescue/emergency drills for lab VMs with console access.

Security Considerations

- Restrict who can change default targets or install timers as root
- Treat timer-run scripts like any privileged automation — least privilege `User=`
- Avoid running arbitrary downloaded scripts from timers
- Remember journal lines from timers may contain sensitive paths
- On shared hosts, review `list-timers --all` for unexpected schedules

Common Mistakes

Enabling the service instead of the timer

The job never schedules. **Fix:** `systemctl enable --now name.timer` and check `list-timers`.

Isolating rescue on a remote cloud VM

You may lose SSH with no console. **Fix:** use provider serial console first; practise isolate only on disposable VMs.

Assuming `network.target` means connectivity

It often means the network stack is up, not that routes/DNS work. **Fix:** order with `network-online.target` when the app needs the network.

Ignoring timezone on OnCalendar

Jobs fire at unexpected wall times. **Fix:** confirm `timedatectl` and document timezone in the runbook.

Best Practices

- Name timers and services as a pair (`app-cleanup.timer` / `.service`)
- Add `RandomizedDelaySec=` for fleet-wide schedules
- Prefer oneshot + timer for periodic work; long loops belong in services
- Capture `systemd-analyze` before/after heavy agents
- Keep cron only where policy already standardises it — do not mix silently

Troubleshooting

Symptom	Likely cause	Fix
Timer never fires	Not enabled / wrong WantedBy	`enable --now`; check `timers.target`
Service fails when fired	ExecStart error	`journalctl -u name.service`
Boot very slow	Slow units in blame	Investigate top blame entries; consider deferring
App starts too early	Wrong After=/Wants=	Use `network-online.target` when needed
Missed calendar runs	Host was off; Persistent not set	Consider `Persistent=true` for calendar timers

Summary

Targets define system goals; timers schedule unit activation; `systemd-analyze` explains boot cost. Inspect safely, enable timers correctly, and prove runs with journal and stamp files. Next: Storage — Disks, Partitions, and Filesystems.

Interview Questions

INTERVIEW PRACTICE

1. What replaced SysV runlevels in systemd, and what is a common server default target?

Answer

Targets replaced runlevels. Many servers use `multi-user.target` (non-graphical multi-user). Desktops often use `graphical.target`, which pulls in multi-user plus display services. Check with `systemctl get-default`.

2. Why enable a `.timer` instead of enabling the `.service` for scheduled work?

Answer

The **timer** is what triggers the service on schedule. Enabling only the service may start it at boot (if WantedBy is set that way) or do nothing useful for periodic runs. For schedules, `enable --now name.timer` and verify with `systemctl list-timers`.

3. When would you choose a systemd timer over cron?

Answer

Choose a **timer** when you need unit dependencies, journal logging, randomised delay, or the same hardening as other systemd services. Cron remains fine for simple per-user jobs where the organisation already standardises crontab.

4. What is dangerous about `systemctl isolate rescue.target` on a remote VM?

Answer

Isolate switches the system toward that target and can **stop SSH and normal services**. Without serial/console access you may lock yourself out. Use only with a planned console path, preferably on a practice VM.

5. How do `network.target` and `network-online.target` differ for application ordering?

Answer

`network.target` means the network management stack is up; it does **not** guarantee a usable default route or DNS. Apps that need real connectivity should typically order after `network-online.target` (understanding that "online" still depends on network configuration quality).

6. How do you investigate a slower boot after installing a monitoring agent?

Answer

Run `systemd-analyze`, `systemd-analyze blame`, and `systemd-analyze critical-chain`. Find units that added large delays or failed dependencies, then fix ordering, defer non-critical work, or open a vendor issue — with before/after evidence.

7. What does `RandomizedDelaySec=` solve on a large fleet?

Answer

It adds **jitter** so thousands of nodes do not hit the same registry, API, or package mirror at the exact same second (thundering herd). Useful for update and housekeeping timers.

Disks, Partitions, and Filesystems
MODULE 8 · STORAGE

Chapter 12. Disks, Partitions, and Filesystems

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/storage-disks-partitions-and-filesystems/

Overview

Attaching a cloud disk is useless until you **discover** it, **partition** it, create a **filesystem**, **mount** it, and make the mount survive reboot safely. On Linux, block devices appear under `/dev`. You usually create a GPT partition, format it (often **ext4** or **XFS**), mount it on a directory, and persist it with **UUID** in `/etc/fstab` (or a systemd `.mount` unit).

Wrong device names wipe the wrong disk. Device letters such as `/dev/sdb` can change after reboot; **UUIDs** do not. Secondary cloud volumes should often use `nofail` so the host still boots if the volume is detached. In this tutorial you will practise the full flow on a **file-backed loop device** so you never touch the real OS disk, and save proof under `~/rebash-linux/lab12`.

In production, databases and logs belong on dedicated mounts. Capacity and performance incidents start with knowing which device backs `/var` or `/data`. Always run `lsblk -f` twice before `mkfs`.

This is **Tutorial 12** in **Module 8: Storage** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- systemd Targets, Timers, and Boot
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Packages: `util-linux`, `e2fsprogs` (normally present); `parted` optional
- Do **not** run `mkfs` on real cloud disks until you can identify them with certainty

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Map disks, partitions, filesystem types, and mount points with `lsblk -f` and `blkid`
- [ ] Create a GPT partition on a loop-backed disk image safely
- [ ] Create an ext4 filesystem and mount it by UUID
- [ ] Explain why UUID/`nofail` matter for cloud secondary disks
- [ ] Clean up loop devices and keep evidence under `~/rebash-linux/lab12`

Architecture

Storage layers stack from disk → partition → filesystem → mount point in the single directory tree.

Disks, Partitions, and Filesystems

Figure 12.1: Disks, Partitions, and Filesystems

Theory

What it is

A **block device** is a disk (virtual or physical). A **partition** carves that disk. A **filesystem** organises files inside a partition. A **mount point** is the directory where that filesystem appears.

```
lsblk -f
sudo blkid
findmnt
```

Why it matters

Formatting the OS disk, omitting persistent mounts, or using unstable `/dev/sdX` names in `fstab` causes outages and data loss. Cloud VMs routinely add second disks for data; without correct layout and `fstab` policy, reboot behaviour becomes a lottery.

How it works

1. **Discover** — `lsblk -f`, `blkid`, `findmnt`
2. **Partition** — `parted` or `fdisk` (lab: loop image only)
3. **Format** — `mkfs.ext4` / `mkfs.xfs`
4. **Mount** — `mount UUID=… /mnt/data`
5. **Persist** — `/etc/fstab` with `UUID=` (and often `nofail` for optional disks)

Layer	Example
Disk	`/dev/nvme1n1`, `/dev/sdb`, loop device
Partition	`/dev/nvme1n1p1`
Filesystem	ext4, XFS
Mount point	`/mnt/data`, `/var/lib/postgresql`

Persistence	Notes
`/etc/fstab`	Classic; prefer `UUID=`
systemd `.mount`	Unit dependencies and journal

Common pitfalls

- Formatting the wrong disk because `lsblk` was not checked twice.
- Using `/dev/sdb1` in `fstab` after devices reorder.
- Forgetting `nofail` on optional data disks and hanging boot.
- Unmounting while applications still hold open files.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Create a 512 MiB disk image, attach it as a loop device, partition and format ext4, mount by UUID under `/mnt/rebash-lab12`, write a test file, then cleanly detach — with evidence under `~/rebash-linux/lab12`.

Prerequisites

- Ubuntu with `sudo`, `losetup`, `parted` or `sfdisk`, `mkfs.ext4`
- ~1 GiB free under `$HOME`

Lab environment

Workspace: `~/rebash-linux/lab12`

```
mkdir -p ~/rebash-linux/lab12 && cd ~/rebash-linux/lab12
set -euo pipefail
sudo apt-get update -qq
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y parted e2fsprogs
lsblk -f | tee lsblk-before.txt
df -h . | tee df-home.txt
```

Expected output: `lsblk-before.txt` exists; enough free space on the home filesystem.

Real-world scenario

A new empty data volume will be attached to an app VM later. Before you touch a real cloud disk, you rehearse partition → mkfs → UUID mount on a loop-backed image, and keep command output for the change ticket.

Step-by-step tasks

Task 1 – Create disk image and attach loop device

```
cd ~/rebash-linux/lab12
set -euo pipefail

dd if=/dev/zero of=labdisk.img bs=1M count=512 status=progress
sudo losetup -fP --show labdisk.img | tee loop-device.txt
LOOP="$(cat loop-device.txt)"
test -b "$LOOP"
lsblk -f "$LOOP" | tee lsblk-loop.txt
echo "Using loop device: $LOOP"
```

Expected output: `loop-device.txt` contains a path such as `/dev/loop0`; `lsblk` shows the loop disk.

Task 2 – Partition, format, and capture UUID

```
cd ~/rebash-linux/lab12
set -euo pipefail
LOOP="$(cat loop-device.txt)"

sudo parted -s "$LOOP" mklabel gpt
sudo parted -s "$LOOP" mkpart primary ext4 1MiB 100%
sudo partprobe "$LOOP" || true
sleep 1

PART="${LOOP}p1"
if [[ ! -b "$PART" ]]; then
  PART="$(lsblk -lnpo NAME,TYPE "$LOOP" | awk '$2=="part"{print $1; exit}')"
fi
test -b "$PART"
echo "$PART" | tee partition.txt

sudo mkfs.ext4 -F -L rebashlab12 "$PART"
sudo blkid "$PART" | tee blkid.txt
UUID="$(blkid -s UUID -o value "$PART")"
test -n "$UUID"
echo "$UUID" | tee uuid.txt
```

Expected output: `uuid.txt` holds a UUID; `blkid.txt` shows `TYPE="ext4"` and label `rebashlab12`.

Task 3 – Mount by UUID, prove write, evidence pack

```
cd ~/rebash-linux/lab12
set -euo pipefail
UUID="$(cat uuid.txt)"

sudo mkdir -p /mnt/rebash-lab12
sudo mount -U "$UUID" /mnt/rebash-lab12
findmnt /mnt/rebash-lab12 | tee findmnt-lab.txt
echo "hello-storage" | sudo tee /mnt/rebash-lab12/proof.txt >/dev/null
sudo cat /mnt/rebash-lab12/proof.txt | tee mount-proof.txt
grep -F 'hello-storage' mount-proof.txt

printf 'UUID=%s /mnt/rebash-lab12 ext4 defaults,nofail 0 2\n' "$UUID" | tee fstab-example.txt

tar -czf storage-evidence.tgz \
  lsblk-before.txt df-home.txt loop-device.txt lsblk-loop.txt \
  partition.txt blkid.txt uuid.txt findmnt-lab.txt mount-proof.txt fstab-example.txt
ls -l storage-evidence.tgz | tee evidence-ls.txt
```

Expected output: `findmnt-lab.txt` shows `/mnt/rebash-lab12`; `mount-proof.txt` contains `hello-storage`; evidence archive exists.

Validation steps

- [ ] Loop device was created from `labdisk.img`
- [ ] Partition has ext4 and a UUID in `uuid.txt`
- [ ] `/mnt/rebash-lab12/proof.txt` was readable while mounted
- [ ] `storage-evidence.tgz` exists under `~/rebash-linux/lab12`

Common errors and fixes

Error	Cause	Fix
`PART` device missing	Partition not rescanned	`sudo partprobe`; check `lsblk`; use `${LOOP}p1`
`mount: special device does not exist`	Wrong UUID / not formatted	Re-run `blkid`; confirm `mkfs` succeeded
`losetup: cannot find unused loop`	Many loops in use	`losetup -a`; detach unused with `losetup -d`
Accidentally targeting `/dev/sda`	Skipped loop workflow	**Stop** — this lab only uses the loop from `loop-device.txt`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Unmount `/mnt/rebash-lab12`, remount using the **LABEL** (`rebashlab12`) instead of UUID, write `label-remount.txt` with `findmnt` output, then unmount again. Keep the loop attached until Cleanup.

Learning outcomes

- Built a safe loop-backed disk workflow
- Partitioned and formatted without touching the OS disk
- Mounted by UUID and drafted an `fstab` line with `nofail`
- Saved storage evidence for a ticket

Cleanup

```
cd ~/rebash-linux/lab12
set -euo pipefail

sudo umount /mnt/rebash-lab12 2>/dev/null || true
LOOP="$(cat loop-device.txt 2>/dev/null || true)"
if [[ -n "${LOOP:-}" ]]; then
  sudo losetup -d "$LOOP" || true
fi
sudo rmdir /mnt/rebash-lab12 2>/dev/null || true
rm -f labdisk.img
# Keep storage-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab12/` with evidence files
- [ ] You can explain disk → partition → filesystem → mount
- [ ] You prefer UUID over `/dev/sdX` in `fstab`
- [ ] You know why `nofail` helps optional cloud volumes

Code Walkthrough

Production attach workflow:

1. **Snapshot / backup** if the disk is not empty
2. **`lsblk -f` twice** — confirm the new empty device
3. **Partition + mkfs** only on the intended device
4. **Mount by UUID**, write a canary file
5. **Persist** with `fstab` or systemd mount; test reboot on a practice VM

Security Considerations

- Restrict who can run `mkfs`, `fdisk`, and raw disk access
- Encrypt sensitive data volumes (LUKS) when policy requires it
- Use `nosuid` / `nodev` mount options on untrusted data mounts when appropriate
- Do not leave world-writable mount points
- Protect `fstab` changes with change control — a bad line can block boot

Common Mistakes

Formatting `/dev/sda` because it was "the second disk yesterday"

Names reorder. **Fix:** identify by size, serial, and emptiness with `lsblk -f` / `blkid` every time.

Persisting `/dev/nvme0n1p1` in fstab

After hardware changes the path may differ. **Fix:** use `UUID=` or `LABEL=`.

Omitting `nofail` on an optional data disk

Boot can hang waiting for a detached volume. **Fix:** add `nofail` (and review systemd mount timeouts).

Forcing `umount` while databases are running

Risk of corruption. **Fix:** stop services, then unmount; investigate open files with `lsof` / `fuser`.

Best Practices

- Separate OS and data disks on cloud VMs
- Document mount points in configuration management
- Prefer GPT + UUID mounts
- Test restore and remount on a twin practice VM
- Monitor free space per mount, not only `/`

Troubleshooting

Symptom	Likely cause	Fix
Device missing after attach	Not rescanned / wrong guest tools	Rescan SCSI/NVMe; check cloud console
`mount` fails	Bad UUID / dirty FS	`blkid`, `fsck` on **unmounted** FS only
Boot hangs on disk	fstab without `nofail`	Rescue mode; fix fstab
Empty mount point after reboot	fstab not updated	Add UUID line; `findmnt`
Wrong disk wiped	Skipped discovery	Restore from backup; enforce checklists

Summary

Discover with `lsblk`, change only the intended device, format deliberately, mount by UUID, and persist mounts carefully. Practise on loop devices before real cloud volumes. Next: LVM, Swap, and Disk Monitoring.

Interview Questions

INTERVIEW PRACTICE

1. Why prefer UUID in `/etc/fstab` over `/dev/sdb1`?

Answer

Kernel device names can **reorder** across reboots or when disks are added. A UUID from `blkid` stays with the filesystem, so the correct volume mounts even if the `/dev` name changes.

2. Walk through attaching a new empty cloud data disk on Ubuntu.

Answer

Attach in the cloud console → `lsblk -f` to identify the empty disk → partition (GPT) → `mkfs` → create mount point → mount by UUID → write a canary file → add `fstab` with `UUID=` and usually `nofail` → reboot test on a non-prod twin. Never skip discovery.

3. What does `nofail` do in fstab, and when do you use it?

Answer

`nofail` lets the system continue booting if that mount fails (for example a secondary volume is detached). Use it for optional data disks. Do not put `nofail` on the root filesystem.

4. How do you unmount a busy filesystem safely?

Answer

Stop services using it, find open files with `lsof` / `fuser`, then `umount`. Avoid lazy/force unmounts on databases unless you accept corruption risk and have a recovery plan.

5. When would you choose XFS over ext4 for a data volume?

Answer

XFS is common for large volumes and some RHEL-family defaults; **ext4** is ubiquitous and simple on Ubuntu. Choose based on distro standards, size, and team operational experience — and know how to grow the filesystem you pick.

6. How does a loop device help you practise safely?

Answer

A **loop device** presents a file as a block device. You can partition and `mkfs` without risking the real OS disk. It is ideal for labs and CI storage tests.

7. What evidence would you attach to a "data disk mounted" change ticket?

Answer

`lsblk -f`, `blkid`, `findmnt` for the mount point, the exact `fstab` line (UUID), and a canary file read test. Before/after reboot proof is even better.

LVM, Swap, and Disk Monitoring
MODULE 8 · STORAGE

Chapter 13. LVM, Swap, and Disk Monitoring

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/lvm-swap-and-disk-monitoring/

Overview

Logical Volume Manager (LVM) turns "we need 50 GiB more" into an online extend instead of a migration weekend. You group physical volumes (PVs) into a volume group (VG), carve logical volumes (LVs), put a filesystem on an LV, and later grow the LV and filesystem when the VG has free space.

Swap is overflow space when RAM is under pressure. Too little swap can cause the Out-Of-Memory (OOM) killer; too much slow disk swap hurts latency. **Disk monitoring** watches free space, inode use, and Input/Output (I/O) errors before users feel pain.

In this tutorial you will create a small loop-backed LVM stack, extend a logical volume, inspect swap, capture basic monitoring signals, and save proof under `~/rebash-linux/lab13`. Practise only on disposable images — never experiment with LVM on the live root disk of a shared server.

This is **Tutorial 13** in **Module 8: Storage** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Disks, Partitions, and Filesystems
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Packages: `lvm2`, `e2fsprogs`
- ~1 GiB free under `$HOME`

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain PV → VG → LV and why LVM helps grow data volumes
- [ ] Create a loop-backed PV/VG/LV, format and mount it
- [ ] Extend an LV and grow an ext4 filesystem online
- [ ] Inspect swap with `swapon --show` and `free`
- [ ] Capture monitoring evidence under `~/rebash-linux/lab13`

Architecture

LVM sits between physical disks and filesystems so you can grow logical volumes without remapping application mount points.

LVM, Swap, and Disk Monitoring

Figure 13.1: LVM, Swap, and Disk Monitoring

Theory

What it is

Object	Role
PV	Physical volume — disk or partition initialised for LVM
VG	Volume group — pool of space from one or more PVs
LV	Logical volume — block device you format and mount
Swap	Backing for anonymous memory under RAM pressure

```
sudo pvs; sudo vgs; sudo lvs
swapon --show
free -h
```

Why it matters

Cloud disks can be resized in the console, but applications need the **LV and filesystem** grown too. Without monitoring, you learn about full disks from failed writes. Swap misconfiguration shows up as latency spikes or sudden OOM kills.

How it works

1. `pvcreate` → `vgcreate` → `lvcreate`
2. `mkfs` on `/dev/vg/lv` → mount
3. Add space: grow disk/PV or add PV → `lvextend` → `resize2fs` / `xfs_growfs`
4. Watch: `df`, `vgs`, smart/cloud disk metrics, `iostat`

Task	Command family
Create	`pvcreate`, `vgcreate`, `lvcreate`
Grow LV	`lvextend -L +size` or `-l +100%FREE`
Grow ext4	`resize2fs` (often online while mounted)
Swap	`swapon --show`, `mkswap` (careful on prod)

Common pitfalls

- Extending the cloud disk but forgetting `pvresize` / `lvextend` / filesystem grow.
- Running LVM experiments on the root VG of a shared host.
- Assuming swap fixes a memory leak (it only delays failure).
- Ignoring read-only remounts after I/O errors.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Build a loop-backed VG with a small LV, mount it, extend the LV using free VG space, grow ext4, inspect swap, and pack evidence under `~/rebash-linux/lab13`.

Prerequisites

- Ubuntu with `sudo` and ability to install `lvm2`

Lab environment

Workspace: `~/rebash-linux/lab13`

```
mkdir -p ~/rebash-linux/lab13 && cd ~/rebash-linux/lab13
set -euo pipefail
sudo apt-get update -qq
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y lvm2 e2fsprogs
sudo vgs | tee vgs-before.txt || true
```

```
free -h | tee free-before.txt
swapon --show | tee swapon-before.txt || true
```

Expected output: `lvm2` installed; baseline memory/swap files exist.

Real-world scenario

An app data volume will need growth next month. You rehearse LVM create + online extend on a loop-backed lab VG named `rebashvg`, prove the filesystem grew, and attach command output to the capacity plan.

Step-by-step tasks

Task 1 – Loop PVs, VG, and LV

```
cd ~/rebash-linux/lab13
set -euo pipefail

dd if=/dev/zero of=pv1.img bs=1M count=384 status=none
dd if=/dev/zero of=pv2.img bs=1M count=384 status=none
LOOP1="$(sudo losetup -fP --show pv1.img)"
LOOP2="$(sudo losetup -fP --show pv2.img)"
echo "$LOOP1" | tee loop1.txt
echo "$LOOP2" | tee loop2.txt

sudo pvcreate "$LOOP1" "$LOOP2"
sudo vgcreate rebashvg "$LOOP1" "$LOOP2"
# Create a 200MiB LV, leave free space in the VG for Task 2
sudo lvcreate -n data -L 200M rebashvg
sudo mkfs.ext4 -F /dev/rebashvg/data

sudo mkdir -p /mnt/rebash-lvm
sudo mount /dev/rebashvg/data /mnt/rebash-lvm
df -h /mnt/rebash-lvm | tee df-before-extend.txt
sudo vgs rebashvg | tee vgs-lab.txt
sudo lvs rebashvg | tee lvs-lab.txt
echo 'before-extend' | sudo tee /mnt/rebash-lvm/note.txt >/dev/null
```

Expected output: `vgs-lab.txt` shows free space in `rebashvg`; `df-before-extend.txt` shows ~200M-class size for the mount.

Task 2 – Online extend LV + filesystem

```
cd ~/rebash-linux/lab13
set -euo pipefail

# Grow LV by 100MiB using free VG space
sudo lvextend -L +100M /dev/rebashvg/data
sudo resize2fs /dev/rebashvg/data
df -h /mnt/rebash-lvm | tee df-after-extend.txt
sudo lvs rebashvg | tee lvs-after.txt
sudo cat /mnt/rebash-lvm/note.txt | tee note-after.txt

# Size after extend should be larger than before (compare 2nd column roughly)
test -s df-after-extend.txt
grep -F 'before-extend' note-after.txt
```

Expected output: `lvs-after.txt` shows a larger data LV (~300M); file content still present; `df-after-extend.txt` reflects growth.

Task 3 – Swap + monitoring signals + evidence

```
cd ~/rebash-linux/lab13
set -euo pipefail

free -h | tee free-after.txt
swapon --show | tee swapon-after.txt || true
cat /proc/swaps | tee proc-swaps.txt
df -hT /mnt/rebash-lvm / | tee df-monitor.txt
# Light I/O sample (if iostat present)
iostat -xz 1 2 | tee iostat.txt 2>/dev/null || echo 'iostat not installed' | tee iostat.txt

tar -czf lvm-evidence.tgz \
  vgs-before.txt free-before.txt swapon-before.txt \
  loop1.txt loop2.txt vgs-lab.txt lvs-lab.txt \
  df-before-extend.txt df-after-extend.txt lvs-after.txt note-after.txt \
  free-after.txt swapon-after.txt proc-swaps.txt df-monitor.txt iostat.txt
ls -l lvm-evidence.tgz | tee evidence-ls.txt
```

Expected output: evidence archive exists; swap/memory files captured (swap may be empty on some cloud images — that is OK).

Validation steps

- [ ] `sudo vgs rebashvg` shows the lab volume group
- [ ] LV grew and `resize2fs` completed while mounted
- [ ] Canary file survived the extend
- [ ] `lvm-evidence.tgz` exists under `~/rebash-linux/lab13`

Common errors and fixes

Error	Cause	Fix
`vgcreate` name in use	Previous lab left VG	Run Cleanup, or use a new VG name
`resize2fs: Permission denied`	Not root / wrong path	Use `sudo resize2fs /dev/rebashvg/data`
No free space to extend	LV consumed whole VG	Create smaller LV first (as in Task 1)
`iostat: command not found`	`sysstat` missing	Optional: `sudo apt-get install -y sysstat`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Add a third loop file `pv3.img` (128 MiB), `pvcreate` + `vgextend rebashvg`, then `lvextend -l +100%FREE` and `resize2fs`. Save `vgs` / `lvs` / `df` to `challenge-extend.txt`.

Learning outcomes

- Built PV/VG/LV on loop devices
- Extended an LV and grew ext4 online
- Inspected swap and basic disk signals
- Packed LVM evidence for a capacity plan

Cleanup

```
cd ~/rebash-linux/lab13
set -euo pipefail

sudo umount /mnt/rebash-lvm 2>/dev/null || true
sudo lvremove -fy rebashvg/data 2>/dev/null || true
sudo vgremove -fy rebashvg 2>/dev/null || true
for f in loop1.txt loop2.txt; do
  if [[ -f "$f" ]]; then sudo losetup -d "$(cat "$f")" 2>/dev/null || true; fi
done
# Detach any remaining loops for pv3 if you did the challenge
sudo rmdir /mnt/rebash-lvm 2>/dev/null || true
rm -f pv1.img pv2.img pv3.img
# Keep lvm-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab13/` with evidence files
- [ ] You can explain PV, VG, and LV in one minute
- [ ] You know the grow order: disk/PV → LV → filesystem
- [ ] You can read swap state with `free` and `swapon --show`

Code Walkthrough

Production grow path:

1. Confirm which LV backs the full mount (`lsblk`, `df`, `findmnt`)
2. Grow the cloud disk / add PV
3. `pvresize` if the PV disk grew
4. `lvextend` then filesystem grow tool
5. Re-check `df` and application health

Security Considerations

- Limit who can run LVM and disk resize tools

- Encrypt sensitive LVs when policy requires it
- Do not disable swap as a “security tip” without understanding OOM behaviour
- Monitor for sudden disk errors that can indicate failing hardware
- Keep backups before destructive `lvremove`

Common Mistakes

Growing the cloud disk only

The guest still sees the old size until PV/LV/filesystem steps run. **Fix:** complete `pvresize` → `lvextend` → filesystem grow.

Experimenting on the root VG

Mistakes can make the host unbootable. **Fix:** use loop labs or a disposable data VG.

Thinking more swap fixes memory leaks

Swap delays OOM and adds latency. **Fix:** find the leak; size RAM/swap deliberately.

Ignoring read-only remounts

Kernel may remount ext4 read-only after errors. **Fix:** check `dmesg` / `journalctl`, run filesystem checks offline, restore from backup if needed.

Best Practices

- Keep free space in VGs for emergency extends
- Alert on VG% and filesystem%
- Document LV → mount → application mapping
- Test extend procedures on staging
- Pair capacity metrics with I/O latency metrics

Troubleshooting

Symptom	Likely cause	Fix
`lvextend` fails	No free VG extents	Add PV / grow disk
`df` unchanged after `lvextend`	Forgot filesystem grow	`resize2fs` / `xfs_growfs`
High swap use	RAM pressure	Inspect processes; add RAM; fix leaks
I/O errors in journal	Failing disk / bad path	Cloud replace disk; restore
VG missing after reboot	Loop/lab disks gone	Expected for loop labs; real PVs need persistent devices

Summary

LVM gives you flexible volumes; swap backs RAM pressure; monitoring tells you before writes fail. Grow in the right order and prove with `df` / `lvs`. Next: Linux Networking Tools.

Interview Questions

INTERVIEW PRACTICE

1. Explain PV, VG, and LV in simple terms.

Answer

A **physical volume (PV)** is a disk/partition given to LVM. A **volume group (VG)** pools one or more PVs. A **logical volume (LV)** is a virtual disk carved from the VG that you format and mount. Applications mount the LV path, so you can grow underneath without changing the mount point name.

2. What is the correct order to grow an ext4 filesystem on LVM after the cloud disk is enlarged?

Answer

Rescan/resize the PV (`pvresize`) → extend the LV (`lvextend`) → grow the filesystem (`resize2fs` for ext4, often online). Then verify with `df` and application checks. Skipping the filesystem step is a classic mistake.

3. When is swap helpful, and when is it a problem?

Answer

Swap helps absorb short memory spikes and can support hibernation on some systems. Heavy sustained swapping causes severe latency. Size swap for the workload; fix memory leaks rather than "adding infinite swap".

4. How do you monitor disks before users notice?

Answer

Alert on filesystem use and inodes (`df`), VG free space (`vgs`), I/O latency/errors (`iostat`, cloud metrics), and SMART/cloud disk health. Attach runbooks that start with `df -hT`, `lsblk`, and the application mount map.

5. Why practise LVM on loop devices in a lab?

Answer

Loop-backed images let you create and destroy PVs without risking the OS disk. You still use real LVM commands, which transfers to cloud data volumes safely.

6. What happens if you `lvremove` the wrong volume?

Answer

Data on that LV is destroyed (unless you have backups/snapshots). Always confirm LV name, mount point, and that the volume is unmounted. Production changes need change control and backups.

7. How does LVM relate to Kubernetes node storage?

Answer

Some clusters use LVM for local persistent volumes or node ephemeral layouts; others use cloud disks/CSI only. Operators still need host skills to interpret `df` / `lsblk` when a pod cannot write because the node filesystem or LV is full.

Linux Networking Tools
MODULE 9 · NETWORKING

Chapter 14. Linux Networking Tools

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/linux-networking-tools/

Overview

When an application "cannot connect", you need a clear order of checks: **Do I have an IP address? Is a process listening? Does Domain Name System (DNS) resolve? Does Transmission Control Protocol (TCP) reach the port?** Linux networking tools answer those questions on the host itself.

This tutorial teaches the modern stack: **`ip`** for addresses and routes, **`ss`** for sockets (prefer this over old `netstat`), **`ping`** / **`traceroute`** (or `tracepath`) for path checks, **`dig`** / **`host`** for DNS, **`curl`** / **`wget`** for Hypertext Transfer Protocol (HTTP) checks, **`nc`** **(netcat)** for port probes, and a short **`tcpdump`** capture when you need packet proof. On cloud virtual machines (VMs), jump servers, Continuous Integration (CI) runners, and Kubernetes nodes, these tools separate "host problem" from "security group / firewall problem" from "application bug".

In production, wrong routes black-hole traffic, broken resolvers make every hostname look dead, and firewalls often block Internet Control Message Protocol (ICMP) even when TCP works. Good engineers collect small evidence files (`ip`, `ss`, `dig`, `curl -I`) before they change security groups or reopen tickets. Prefer `ip` / `ss` on current Ubuntu and Red Hat images — `ifconfig` and `netstat` are legacy.

This is **Tutorial 14** in **Module 9: Linux Networking** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a small network evidence pack you can attach to an incident ticket.

Prerequisites

- LVM, Swap, and Disk Monitoring
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Outbound DNS and HTTPS allowed (lab uses public DNS and `example.com`)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Read addresses, routes, and neighbours with `ip`
- [ ] List listening sockets with `ss -tulpn` and explain what you see
- [ ] Query DNS with `dig` / `host` and test HTTP with `curl`
- [ ] Probe a TCP port with `nc` and capture a short packet sample with `tcpdump`
- [ ] Build a network evidence pack suitable for an incident ticket

Architecture

Networking tools sit above the kernel TCP/IP stack. Applications open sockets; the kernel routes packets; host firewalls and the network interface card (NIC) send and receive traffic.

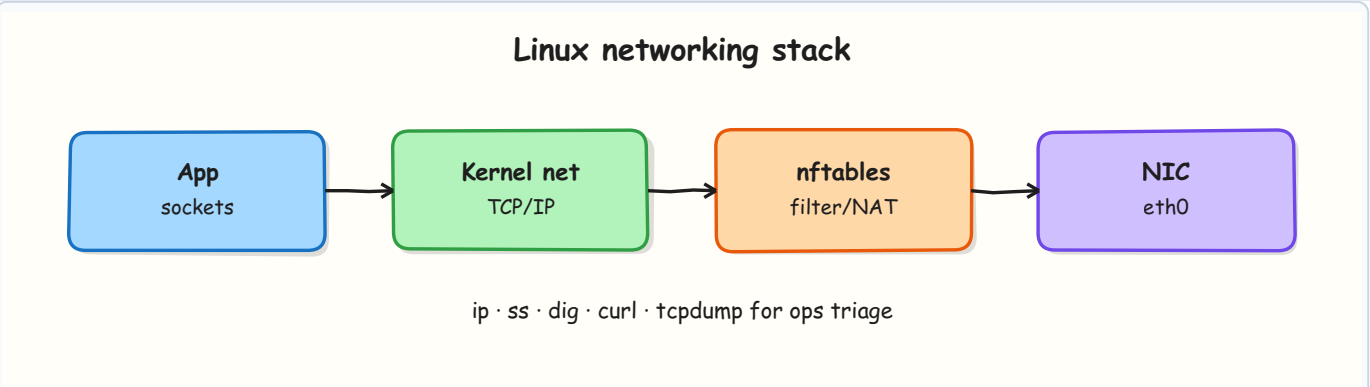

Figure 14.1: Linux Networking Tools

Theory

What it is

Question	Tool
Do I have an address / route?	`ip`
Is anything listening?	`ss -tulpn`
Can I reach a host (ICMP)?	`ping`
Where does the path go?	`traceroute` / `tracepath`
Does the name resolve?	`dig`, `host`, `nslookup`
Does HTTP/TLS work?	`curl`, `wget`
Is the TCP port open?	`nc` (netcat)
What packets are on the wire?	`tcpdump`

`ip` replaces `ifconfig` for addresses, routes, and the neighbour (ARP) cache. `ss` replaces `netstat` for socket state. Keep both ideas: **configuration** (`ip`) and **who is talking** (`ss`).

```
ip -br a
ip route
ss -tulpn
```

Why it matters

Most “app is down” tickets are network or DNS until proven otherwise. Cloud security groups and host firewalls fail closed. A wrong default route or empty resolver list looks like a total outage. Fast, accurate use of `ip`, `ss`, and `dig` saves hours and avoids random reboots.

How it works

1. **Local stack** — `ip -br a` shows interfaces; `ip route` shows the routing table; `ip neigh` shows ARP/neighbour entries.
2. **Listeners** — `ss -tulpn` lists TCP/UDP listeners with process names when permissions allow.
3. **Reachability** — `ping -c 3` proves ICMP; remember some networks block ICMP while TCP still works.
4. **DNS** — `dig +short example.com A` asks for an A record deliberately; `host example.com` is a short alternative.
5. **Application layer** — `curl -I https://example.com` checks HTTP headers and TLS; `wget` can download files.
6. **Port probe** — `nc -vz host port` checks whether a TCP port accepts a connection.
7. **Capture** — `tcpdump` records packets for a short, targeted window; stop quickly and treat captures as sensitive.

```
dig +short example.com A
curl -I --max-time 10 https://example.com
nc -vz 1.1.1.1 53
```

Key concepts and comparisons

Modern	Legacy (avoid as first choice)
`ip`	`ifconfig` / `route`
`ss`	`netstat`
`dig` / `host`	only `nslookup`

Signal	Means	Next check
No address on NIC	DHCP / cloud metadata / cable	`ip -br a`, cloud console
Address OK, no route	Missing default route	`ip route`
DNS fails	Resolver / Network Manager	`/etc/resolv.conf`, `dig @8.8.8.8`
DNS OK, TCP fails	Firewall / security group / wrong port	`ss`, `nc`, cloud SG
TCP OK, HTTP fails	App / TLS / reverse proxy	`curl -v`, app logs

Common pitfalls

- Trusting `ping` alone when ICMP is blocked but HTTPS works.
- Using `ifconfig`/`netstat` on images where they are missing.
- Running long `tcpdump` captures that fill the disk or leak secrets.
- Forgetting that `ss -p` may need root to show process names.
- Changing cloud security groups before proving the host has an address and a listener.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, collect addresses, routes, listeners, DNS, HTTP, a port probe, and a short packet capture under `~/rebash-linux/lab14`, then pack the proof into one archive.

Prerequisites

- Ubuntu 22.04/24.04 with `sudo`
- Packages: `iproute2`, `iputils-ping`, `dnsutils` (for `dig`/`host`), `curl`, `netcat-openbsd` or `ncat`, `tcpdump` (install if missing)
- Outbound DNS (UDP/TCP 53) and HTTPS (443)

Lab environment

Workspace: `~/rebash-linux/lab14`

```
mkdir -p ~/rebash-linux/lab14 && cd ~/rebash-linux/lab14
set -euo pipefail
whoami | tee admin-user.txt
sudo -n true 2>/dev/null || sudo -v

# Install missing tools (Ubuntu/Debian)
sudo apt-get update -y
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y \
  iproute2 iputils-ping dnsutils curl netcat-openbsd tcpdump traceroute
```

Expected output: packages install (or already present); you can run `ip`, `ss`, `dig`, `curl`, `nc`, `tcpdump`.

Real-world scenario

Users say the portal is slow. Before you touch the load balancer, the on-call engineer asks for host proof: IP and route, listening ports, DNS for the public name, an HTTPS header check, and a short capture showing DNS or TCP. You gather that evidence on the practice VM the same way you would on a bastion or app node.

Step-by-step tasks

Task 1 – Addresses, routes, and listeners

```
cd ~/rebash-linux/lab14
set -euo pipefail

ip -br a | tee ip-brief.txt
ip route | tee ip-route.txt
ip neigh | tee ip-neigh.txt || true
ss -tulpn | tee ss-listen.txt

# Basic asserts
grep -E 'UP|UNKNOWN' ip-brief.txt
test -s ip-route.txt
grep -E 'LISTEN|UNCONN' ss-listen.txt
```

Expected output: at least one UP interface with an address; a non-empty route table; `ss` shows listening sockets (for example `ssh` on port 22).

Task 2 – DNS and HTTP checks

```
cd ~/rebash-linux/lab14
set -euo pipefail

# Resolver file (may be a symlink on systemd-resolved systems)
ls -l /etc/resolv.conf | tee resolv-ls.txt
cat /etc/resolv.conf | tee resolv.conf.txt

dig +short example.com A | tee dig-a.txt
host example.com | tee host-example.txt
test -s dig-a.txt

# ICMP may be blocked in some networks — record result, do not fail the lab
ping -c 3 -W 2 1.1.1.1 | tee ping-cloudflare.txt || echo "ping blocked or failed" | tee -a ping-cloudflare.txt

curl -sS -I --max-time 15 https://example.com | tee curl-headers.txt
grep -E 'HTTP/|location:| LocatioN:' -i curl-headers.txt
```

Expected output: `dig-a.txt` has at least one IPv4 address; `curl-headers.txt` shows an HTTP status line (often `HTTP/2 200` or a redirect).

Task 3 – Port probe, short capture, evidence pack

```
cd ~/rebash-linux/lab14
set -euo pipefail

# TCP probe to a well-known DNS server (port 53)
nc -vz -w 5 1.1.1.1 53 2>&1 | tee nc-dns.txt
grep -Ei 'succeeded|open|connected' nc-dns.txt

# Short tcpdump while generating DNS traffic (needs sudo)
sudo timeout 8 tcpdump -nn -c 20 -i any udp port 53 \
  -w dns-sample.pcap 2>tcpdump-stderr.txt || true
sudo chmod a+r dns-sample.pcap 2>/dev/null || true
test -s dns-sample.pcap
tcpdump -nn -r dns-sample.pcap 2>/dev/null | head -n 20 | tee dns-sample-read.txt || true

tar -czf network-evidence.tgz \
  admin-user.txt ip-brief.txt ip-route.txt ip-neigh.txt ss-listen.txt \
  resolv-ls.txt resolv.conf.txt dig-a.txt host-example.txt \
  ping-cloudflare.txt curl-headers.txt nc-dns.txt \
  dns-sample.pcap dns-sample-read.txt tcpdump-stderr.txt
ls -l network-evidence.tgz | tee evidence-ls.txt
```

Expected output: `nc` reports success/open to `1.1.1.1:53`; `dns-sample.pcap` is not empty; `network-evidence.tgz` exists.

Validation steps

- [ ] `ip -br a` shows an UP interface with an address
- [ ] `ss -tulpn` lists listening sockets
- [ ] `dig +short example.com A` returns an address
- [ ] `curl -I https://example.com` returns HTTP headers
- [ ] `network-evidence.tgz` exists under `~/rebash-linux/lab14`

Common errors and fixes

Error	Cause	Fix
`dig: command not found`	`dnsutils` missing	`sudo apt-get install -y dnsutils`
`nc: command not found`	netcat package missing	`sudo apt-get install -y netcat-openbsd`
`curl: Could not resolve host`	DNS broken	Check `/etc/resolv.conf`; try `dig @1.1.1.1 example.com`
Empty `pcap`	No UDP/53 traffic or wrong interface	Re-run capture; use `-i any`; generate traffic with `dig` in another shell
`ss` shows no process names	Not root	Use `sudo ss -tulpn` when you need PIDs

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Start a temporary listener with `nc -l 127.0.0.1 9999` in one terminal (or background), prove it with `ss -tlnp | grep 9999`, then connect with `nc -vz 127.0.0.1 9999`. Save both outputs as `challenge-ss.txt` and `challenge-nc.txt`. Stop the listener when done.

Learning outcomes

- Collected host IP, route, and socket evidence
- Proved DNS and HTTPS from the VM
- Used `nc` and a short `tcpdump` capture
- Packed evidence for an incident ticket

Cleanup

```
cd ~/rebash-linux/lab14
set -euo pipefail
# Stop any leftover challenge listener if you started one
pkill -f 'nc -l 127.0.0.1 9999' 2>/dev/null || true
# Keep the evidence archive if you want it; otherwise:
# rm -f network-evidence.tgz *.txt *.pcap
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab14/` with evidence files
- [ ] You can explain when to use `ip` vs `ss` vs `dig` vs `curl`
- [ ] You know why ping success is not the same as TCP/HTTP success
- [ ] You can describe one production failure (wrong route, bad DNS, closed port)

Code Walkthrough

In real incidents, host networking checks usually follow this order:

1. **Local identity** — `ip -br a`, `ip route`
2. **Listeners** — `ss -tulpn` (is the app even listening?)
3. **Name → address** — `dig` / `host`
4. **Port / HTTP** — `nc`, `curl -I` / `curl -v`
5. **Proof** — short `tcpdump` only when the above is not enough

Prefer modern tools. Keep captures short. Attach text evidence to the ticket.

Security Considerations

- Packet captures may contain secrets (cookies, tokens) — store and share carefully
- Do not run wide `tcpdump` on production without change control
- Prefer least privilege: many checks work without root; use sudo only for capture / process names
- Treat public DNS tests as connectivity checks, not as load tests
- Never disable host firewalls “just to test” without a rollback plan

Common Mistakes

Using only ping to declare the network healthy

Many clouds block ICMP. **Fix:** also test TCP (`nc`) and HTTP (`curl`).

Relying on ifconfig/netstat

They may be missing on minimal images. **Fix:** learn `ip` and `ss` first.

Long tcpdump captures

Disk fills; privacy risk rises. **Fix:** use `-c` count or `timeout`, filter ports, stop quickly.

Changing security groups before host proof

You may "fix" the wrong layer. **Fix:** collect `ip` / `ss` / `dig` first.

Best Practices

- Keep a fixed check order: address → route → listen → DNS → port → HTTP
- Save command output for tickets (`tee`)
- Prefer `dig +short` for scripts; use `curl -v` when TLS fails
- Use `ss -tulpn` before restarting services "to fix networking"
- Document which checks need outbound internet vs private VPC only

Troubleshooting

Symptom	Likely cause	Fix
`Network is unreachable`	No default route	Check `ip route`, cloud routing tables
`Temporary failure in name resolution`	Broken resolver	Fix `/etc/resolv.conf` / systemd-resolved
Connection timed out	Firewall / SG / wrong IP	`nc -vz`, check cloud security group
Connection refused	Nothing listening	`ss -tulpn`, start or fix the service
HTTP works, ping fails	ICMP filtered	Ignore ping; trust TCP/HTTP

Summary

Linux networking tools give you a clear path from **address and route** to **socket**, **DNS**, **port**, and **HTTP**. Prefer `ip` and `ss`, prove each layer with short commands, and save evidence before you change cloud firewalls. Next, learn day-to-day remote access in SSH and Remote Access.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between `ip` and `ss`, and when do you use each in an incident?

Answer

`ip` shows host network configuration: interfaces, addresses, routes, and neighbours. `ss` shows socket state: who is listening and who is connected. In an incident, first confirm the host has an address and a route (`ip`), then check whether the application port is listening (`ss`), then test DNS and the remote port.

2. Ping fails but `curl https://…` works. What does that tell you?

Answer

ICMP may be blocked by a firewall or security group while TCP 443 is allowed. Ping is useful but not required for "network healthy". Prefer TCP probes (`nc`) and application checks (`curl`) as proof of service reachability.

3. How do you prove DNS is the problem versus the application?

Answer

Run `dig +short name A` (or `host name`). If DNS fails, try `dig @1.1.1.1 name` to see whether a public resolver works. If dig works but the app fails, check the app's resolver settings, caching, or wrong hostname. If dig fails everywhere, fix resolvers (`/etc/resolv.conf`, systemd-resolved, VPC DNS).

4. Why prefer `ss` over `netstat` on modern Ubuntu cloud images?

Answer

`ss` is maintained with `iproute2` and is present on most minimal images. `netstat` often needs the older `net-tools` package. Interviewers expect you to default to `ss -tulpn` for listeners.

5. How would you use `tcpdump` safely during a production incident?

Answer

Capture for a short time with a tight filter (`port 53`, `host x.x.x.x`), use `-c` or `timeout`, write to a known path, and treat the file as sensitive. Prefer proving the issue with `ip` / `ss` / `dig` / `curl` first; use packet capture when you need wire proof.

6. `nc -vz host 443` succeeds but the browser shows a certificate error. Which layer failed?

Answer

The **TCP port is open**, so routing and firewall likely allow traffic. The failure is at **TLS/certificate** (or hostname mismatch), not basic connectivity. Next use `curl -vI https://host` and inspect the certificate chain.

7. A new cloud VM has no outbound internet. Which three commands do you run first?

Answer

(1) `ip -br a` and `ip route` for address and default route; (2) `cat /etc/resolv.conf` and `dig` for DNS; (3) `curl -I` or `nc -vz` to a known endpoint. Then check cloud route tables, NAT gateway, and security groups — but only after host-local proof.

SSH and Remote Access
MODULE 9 · NETWORKING

Chapter 15. SSH and Remote Access

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/ssh-and-remote-access/

Overview

Secure Shell (SSH) is how you log in to almost every Linux cloud virtual machine (VM), jump server (bastion), and build agent. You type commands on your laptop; they run on the remote host over an encrypted channel. Day-to-day work also uses SSH for file copy (`scp`, `rsync`) and simple port forwarding to reach private services through a bastion.

This tutorial focuses on **access fundamentals**: generate an **Ed25519** key pair, install the public key in `authorized_keys`, log in without a password (to localhost in the lab), write a small `~/.ssh/config` Host stanza, and copy a file with `scp` / `rsync`. Module 13 covers hardening (disable password login, firewalls). Here the goal is correct, key-based access you can explain in an interview or change ticket.

In production, weak key hygiene, shared private keys in chat, `StrictHostKeyChecking=no` "to make CI work", or agent forwarding to untrusted hosts create lasting security debt. Cloud images often allow the first user with a vendor-injected key. You still need to understand `sshd`, `~/.ssh/authorized_keys` modes (`700` / `600`), and client config aliases so automation (`ssh host 'uptime'`) is reliable under pressure.

This is **Tutorial 15** in **Module 9: Linux Networking** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Linux Networking Tools
- A **practice Ubuntu 22.04/24.04 VM** where OpenSSH server is installed (or you can install it)
- You may create lab keys under `~/rebash-linux/lab15` (do **not** overwrite your real production keys without a backup)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain how SSH client, `sshd`, keys, and `authorized_keys` work together
- [ ] Generate an Ed25519 key pair and install the public key for passwordless login
- [ ] Create a `~/.ssh/config` Host alias and use it for remote commands
- [ ] Copy files with `scp` and `rsync` over SSH
- [ ] Check basic permissions and `sshd` status when login fails

Architecture

The SSH client authenticates to `sshd` using a key (preferred) or a password. After login, the same channel can carry remote commands and file transfers.

SSH and Remote Access

Figure 15.1: SSH and Remote Access

Theory

What it is

SSH provides encrypted remote login and a secure channel for commands and file copy. Important pieces:

Piece	Role
`ssh` client	Connects from your machine
`sshd`	Server daemon on the remote host
Key pair	Private key stays with you; public key goes on the server
`~/.ssh/authorized_keys`	List of public keys allowed to log in as that user
`~/.ssh/config`	Client aliases (Host, User, IdentityFile, Port)
`scp` / `rsync`	Copy files over SSH

```
ssh-keygen -t ed25519 -f ./lab_ed25519 -N ""
ssh -i ./lab_ed25519 user@host
```

Why it matters

Almost every cloud VM interaction starts with SSH. Password login on the public internet is risky. Key-based login is the default expectation in DevOps and SRE roles. Fluent client config reduces typing mistakes during incidents and makes scripts reliable.

How it works

1. **Generate keys** — `ssh-keygen -t ed25519` (strong default on modern OpenSSH).
2. **Install public key** — append the `.pub` line to the remote user's `~/.ssh/authorized_keys` (`ssh-copy-id` helps when password login still works).
3. **Fix modes** — `~/.ssh` should be `700`; `authorized_keys` and private keys should be `600`.
4. **Connect** — `ssh -i key user@host` or a `Host` stanza in `~/.ssh/config`.
5. **Verify host keys** — the client stores server fingerprints in `known_hosts` to detect impersonation.
6. **Transfer** — `scp file host:path` or `rsync -avz -e ssh …`.
7. **Forward (optional)** — `ssh -L localport:target:port bastion` reaches a private service through a jump host.

```
ssh-copy-id -i ./lab_ed25519.pub user@host
ssh -i ./lab_ed25519 user@host 'hostname; whoami'
```

Hardening (disable passwords, restrict users, Fail2Ban) comes later. Here, prove keys and config work.

Key concepts and comparisons

Method	Prefer when	Avoid when
Public-key auth	Production, cloud VMs, CI	Never share the private key
Password auth	Emergency bootstrap only	Internet-facing default
Certificate auth	Large fleets, short-lived access	Overkill for a single lab VM
Agent forwarding	Rare, carefully controlled	Blind forwarding to untrusted hosts

Common pitfalls

- Overwriting `~/.ssh/id_ed25519` without backup.
- Wrong modes on `.ssh` or `authorized_keys` → silent key rejection.
- Using `StrictHostKeyChecking=no` permanently in scripts.
- Putting the **private** key on the server.
- Expecting a new group membership to apply inside an old SSH session without reconnecting.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, generate a lab-only Ed25519 key, enable passwordless SSH to **localhost** as your user, add a `Host` alias, run a remote command, and copy a file with `scp` and `rsync`. Save proof under `~/rebash-linux/lab15`.

Prerequisites

- Ubuntu 22.04/24.04 with `sudo`
- Packages: `openssh-client`, `openssh-server`, `rsync`
- Local SSH to `127.0.0.1` allowed (default on Ubuntu)

Lab environment

Workspace: `~/rebash-linux/lab15`

```
mkdir -p ~/rebash-linux/lab15 && cd ~/rebash-linux/lab15
set -euo pipefail
whoami | tee admin-user.txt
id -u | tee admin-uid.txt
sudo -n true 2>/dev/null || sudo -v

sudo apt-get update -y
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y openssh-client openssh-server rsync
sudo systemctl enable --now ssh
sudo systemctl is-active ssh | tee sshd-active.txt
```

Expected output: `sshd-active.txt` contains `active`.

Real-world scenario

A new engineer needs repeatable SSH access to a bastion: key-based login, a short `Host` alias in `~/.ssh/config`, and a way to push a small config file with `rsync`. You practise the same workflow safely against localhost first, then reuse the pattern for real hosts.

Step-by-step tasks

Task 1 – Generate lab key and install for localhost

```
cd ~/rebash-linux/lab15
set -euo pipefail

# Lab-only key (empty passphrase for practice VM — use a passphrase on real laptops)
ssh-keygen -t ed25519 -f ./rebash_lab_ed25519 -N "" -C "rebash-lab15@$(hostname)"
test -f rebash_lab_ed25519
test -f rebash_lab_ed25519.pub
chmod 600 rebash_lab_ed25519

mkdir -p ~/.ssh
chmod 700 ~/.ssh
touch ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys

# Install public key if not already present
PUB="$(cat rebash_lab_ed25519.pub)"
grep -Fqx "$PUB" ~/.ssh/authorized_keys || echo "$PUB" >> ~/.ssh/authorized_keys

# Passwordless SSH to localhost
ssh -i ./rebash_lab_ed25519 \
  -o IdentitiesOnly=yes \
  -o BatchMode=yes \
  -o StrictHostKeyChecking=accept-new \
  "$USER@127.0.0.1" 'echo OK; hostname; whoami' | tee ssh-localhost.txt

grep -q OK ssh-localhost.txt
```

Expected output: `ssh-localhost.txt` shows `OK`, your hostname, and your username.

Task 2 – Client config Host alias

```
cd ~/rebash-linux/lab15
set -euo pipefail

LAB_KEY="$(pwd)/rebash_lab_ed25519"
CONFIG_SNIPPET="$HOME/.ssh/config"

# Backup existing config once
if [ -f "$CONFIG_SNIPPET" ] && [ ! -f "$HOME/.ssh/config.rebash-lab15.bak" ]; then
  cp -a "$CONFIG_SNIPPET" "$HOME/.ssh/config.rebash-lab15.bak"
```

```
fi

# Remove previous lab block if re-running
if [ -f "$CONFIG_SNIPPET" ]; then
  awk '
    /^# BEGIN REBASH-LAB15$/ {skip=1; next}
    /^# END REBASH-LAB15$/ {skip=0; next}
    !skip {print}
  ' "$CONFIG_SNIPPET" > "$CONFIG_SNIPPET.tmp" && mv "$CONFIG_SNIPPET.tmp" "$CONFIG_SNIPPET"
fi

cat >> "$CONFIG_SNIPPET" << EOF
# BEGIN REBASH-LAB15
Host rebash-lab15
  HostName 127.0.0.1
  User $USER
  IdentityFile $LAB_KEY
  IdentitiesOnly yes
# END REBASH-LAB15
EOF
chmod 600 "$CONFIG_SNIPPET"

ssh -o BatchMode=yes rebash-lab15 'uptime; echo CONFIG_OK' | tee ssh-alias.txt
grep -q CONFIG_OK ssh-alias.txt
```

Expected output: `ssh-alias.txt` contains `CONFIG_OK` and an `uptime` line.

Task 3 – scp, rsync, and evidence pack

```
cd ~/rebash-linux/lab15
set -euo pipefail

echo "rebash lab15 payload $(date -Is)" > payload.txt

# scp to a remote path (localhost)
scp -o BatchMode=yes payload.txt rebash-lab15:~/rebash-linux/lab15/payload-from-scp.txt
ssh -o BatchMode=yes rebash-lab15 'test -f ~/rebash-linux/lab15/payload-from-scp.txt && cat ~/rebash-linux/lab15/payload-from-scp.txt' \
  | tee scp-verify.txt

# rsync over SSH
rsync -avz -e "ssh -o BatchMode=yes" payload.txt rebash-lab15:~/rebash-linux/lab15/payload-from-rsync.txt
ssh -o BatchMode=yes rebash-lab15 'test -f ~/rebash-linux/lab15/payload-from-rsync.txt && wc -c ~/rebash-linux/lab15/payload-from-rsync.txt' \
  | tee rsync-verify.txt

# Permission proof
ls -ld ~/.ssh | tee ssh-dir-mode.txt
ls -l ~/.ssh/authorized_keys rebash_lab_ed25519 | tee key-modes.txt
ssh -o BatchMode=yes rebash-lab15 'sudo systemctl is-active ssh' | tee sshd-remote-check.txt

tar -czf ssh-evidence.tgz \
  admin-user.txt admin-uid.txt sshd-active.txt \
  rebash_lab_ed25519.pub ssh-localhost.txt ssh-alias.txt \
  payload.txt scp-verify.txt rsync-verify.txt \
  ssh-dir-mode.txt key-modes.txt sshd-remote-check.txt
# Do NOT pack the private key into shared tickets
ls -l ssh-evidence.tgz | tee evidence-ls.txt
```

Expected output: scp and rsync verify files exist; `ssh-evidence.tgz` is present; private key is **not** inside the archive.

Validation steps

- [ ] `ssh -i ./rebash_lab_ed25519 "$USER@127.0.0.1"` works in BatchMode
- [ ] `ssh rebash-lab15` works via `~/.ssh/config`
- [ ] `payload-from-scp.txt` and `payload-from-rsync.txt` exist
- [ ] `~/.ssh` is mode `700` and keys are not world-readable
- [ ] `ssh-evidence.tgz` exists under `~/rebash-linux/lab15`

Common errors and fixes

Error	Cause	Fix
`Permission denied (publickey)`	Public key not installed / wrong key	Check `authorized_keys` and `IdentityFile`
`Bad owner or permissions on .ssh`	Modes too open	`chmod 700 ~/.ssh`; `chmod 600 authorized_keys`
`Connection refused`	`sshd` not running	`sudo systemctl enable --now ssh`
Host key prompt blocks Batch-Mode	First connect	Use one interactive connect, or `accept-new` once in lab
Overwrote real keys	Wrong `-f` path	Always use a lab path like `./rebash_lab_ed25519`

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Create a second key `rebash_lab_ed25519_b`, append its public key to `~/.ssh/authorized_keys`, connect with `ssh -i ./rebash_lab_ed25519_b -o IdentitiesOnly=yes -o BatchMode=yes "$USER@127.0.0.1" 'echo SECOND_OK'`, and save the output as `ssh-second-key.txt`. Remove the second public key line and delete the second key files in Cleanup.

Learning outcomes

- Generated and installed an Ed25519 key for localhost SSH
- Used a `Host` alias from `~/.ssh/config`
- Copied files with `scp` and `rsync`
- Saved access proof without sharing the private key

Cleanup

```
cd ~/rebash-linux/lab15
set -euo pipefail

# Remove lab Host block from config
if [ -f "$HOME/.ssh/config" ]; then
  awk '
    /^# BEGIN REBASH-LAB15$/ {skip=1; next}
    /^# END REBASH-LAB15$/ {skip=0; next}
    !skip {print}
  ' "$HOME/.ssh/config" > "$HOME/.ssh/config.tmp" && mv "$HOME/.ssh/config.tmp" "$HOME/.ssh/config"
fi

# Remove lab public key lines from authorized_keys
if [ -f rebash_lab_ed25519.pub ]; then
  PUB="$(cat rebash_lab_ed25519.pub)"
  grep -Fvx "$PUB" ~/.ssh/authorized_keys > ~/.ssh/authorized_keys.tmp 2>/dev/null || true
  mv ~/.ssh/authorized_keys.tmp ~/.ssh/authorized_keys
  chmod 600 ~/.ssh/authorized_keys
fi
if [ -f rebash_lab_ed25519_b.pub ]; then
  PUBB="$(cat rebash_lab_ed25519_b.pub)"
  grep -Fvx "$PUBB" ~/.ssh/authorized_keys > ~/.ssh/authorized_keys.tmp 2>/dev/null || true
  mv ~/.ssh/authorized_keys.tmp ~/.ssh/authorized_keys
  chmod 600 ~/.ssh/authorized_keys
fi

rm -f rebash_lab_ed25519 rebash_lab_ed25519.pub rebash_lab_ed25519_b rebash_lab_ed25519_b.pub
# Optional: restore config backup if you prefer
# mv ~/.ssh/config.rebash-lab15.bak ~/.ssh/config
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab15/` with evidence files
- [ ] You can explain key-based SSH vs password login
- [ ] You know correct `.ssh` permission modes
- [ ] You can describe one production risk (shared private keys, disabled host key checks)

Code Walkthrough

Typical production access workflow:

1. **Generate** a personal key (passphrase on real laptops)
2. **Install** the public key only (never the private key)
3. **Alias** the host in `~/.ssh/config`
4. **Prove** with `ssh host 'hostname; whoami'`
5. **Transfer** with `rsync` over SSH for repeated syncs

Hardening comes after access works.

Security Considerations

- Never commit private keys to git or chat
- Prefer Ed25519; protect private keys with a passphrase outside lab VMs
- Keep `~/.ssh` mode `700` and private keys `600`
- Do not leave `StrictHostKeyChecking=no` as a permanent default
- Limit who can reach `sshd` with security groups / firewalls (covered in hardening)

Common Mistakes

Sharing one private key across the whole team

Compromise of one laptop compromises every server. **Fix:** one key per person (or short-lived certificates).

World-readable authorized_keys or .ssh

`sshd` may ignore the keys. **Fix:** `chmod 700 ~/.ssh` and `chmod 600 ~/.ssh/authorized_keys`.

Disabling host key checking in every script

You lose protection against impersonation. **Fix:** manage `known_hosts` properly; use `accept-new` only with care.

Putting the private key on the server

Attackers who gain the server get lateral movement. **Fix:** only the `.pub` file belongs on the server.

Best Practices

- Use `IdentitiesOnly yes` in Host stanzas to avoid offering the wrong keys
- Name keys by purpose (`work-bastion`, `ci-deploy`)
- Prefer `rsync -avz` for sync; use `scp` for one-off copies
- Document jump-host patterns (`ProxyJump`) for private subnets
- Rotate keys when people leave the team

Troubleshooting

Symptom	Likely cause	Fix
`Permission denied (publickey)`	Key not authorised / wrong user	Check `authorized_keys`, user, `IdentityFile`
`Connection refused`	sshd down / wrong port	`systemctl status ssh`; check port 22
Works interactively, fails in CI	Missing key / BatchMode / pass-phrase	Use deploy keys or an agent carefully
Host key verification failed	Server rebuilt	Verify fingerprint out-of-band; update `known_hosts`
Slow first connect	DNS / GSSAPI attempts	Set `GSSAPIAuthentication no` in client config if needed

Summary

SSH is the main remote access tool for Linux in the cloud. Generate keys, install only the public key, use a clear client `Host` alias, and prove login and file copy with saved output. Next, keep hosts consistent with Package Management. For hardening, continue later to SSH Hardening and Firewalls.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between the SSH private key and the public key, and which one goes on the server?

Answer

The **private key** stays on the client (your laptop or a secured secrets store). The **public key** is placed in the server user's `~/.ssh/authorized_keys`. Anyone with the private key can prove identity; the public key alone cannot log in. Never copy the private key to the server or into git.

2. Why do `~/.ssh` mode `777` or `authorized_keys` mode `644` often break key login?

Answer

OpenSSH refuses overly open permissions to stop other users on the same host from tampering with your keys. Typical safe modes are **`700`** for `~/.ssh` and **`600`** for `authorized_keys` and private keys. Check with `ls -ld ~/.ssh` and `ls -l ~/.ssh/authorized_keys`.

3. What does a `Host` stanza in `~/.ssh/config` buy you in daily ops?

Answer

It stores `HostName`, `User`, `IdentityFile`, `Port`, and optional `ProxyJump` under a short alias. Instead of long commands, you run `ssh bastion` or `ssh app1`. That reduces mistakes during incidents and keeps automation consistent.

4. When would you choose `rsync` over `scp`?

Answer

Use **`scp`** for a simple one-off copy. Prefer **`rsync`** when you sync directories repeatedly — it can transfer only changes, preserve permissions with `-a`, and is clearer for deploy artefacts. Both commonly run over SSH (`rsync -e ssh`).

5. A junior engineer sets `StrictHostKeyChecking=no` in CI to stop prompts. What is the risk, and what is better?

Answer

The client will accept a new host key without verifying it, which enables man-in-the-middle attacks. Better options: preload known host keys in the CI image, use a secrets-managed `known_hosts` file, or carefully use `accept-new` only when the environment is already trusted and controlled.

6. How do you prove passwordless SSH works for an interview or change ticket?

Answer

Show `ssh -o BatchMode=yes -i key user@host 'whoami; hostname'` succeeding (BatchMode refuses password prompts), show the public key line in `authorized_keys`, and show key file modes. Attach command output — not the private key.

7. How does cloud "inject my SSH key at boot" relate to `authorized_keys`?

Answer

Cloud-init (or the vendor agent) usually appends your uploaded public key to the default user's `~/.ssh/authorized_keys` on first boot. It is still normal OpenSSH afterwards — you manage extra users and keys the same way. Always verify with a test login and correct permissions after image changes.

Package Management
MODULE 10 · PACKAGES

Chapter 16. Package Management

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/package-management/

Overview

A **package manager** installs, updates, and removes software in a consistent way for your Linux distribution. On Ubuntu and Debian you use `apt` (and lower-level `dpkg`). On Red Hat Enterprise Linux (RHEL) family systems you use `dnf` (or older `yum`) with `rpm`. On SUSE you use `zypper`. Optional tools such as **snap** and **Flatpak** exist mainly for desktop apps and are usually secondary on servers.

Unpatched packages are security debt. Golden images, Continuous Integration (CI) runners, and configuration management all assume a known package state. In this tutorial you will refresh package metadata, install a small tool, query version and files, place an **apt hold** (pin so it does not upgrade by accident), remove a package cleanly, and save proof under `~/rebash-linux/lab16`. The lab uses Ubuntu `apt` because that matches most practice VMs; the Theory tables cover other families so you can work across clouds.

In production, prefer distribution packages for system daemons, reboot after kernel updates, and record critical versions in image builds rather than "hand-installed" binaries that nobody can reproduce. Automate patching carefully (`unattended-upgrades`, `dnf-automatic`) with a maintenance window.

This is **Tutorial 16** in **Module 10: Package Management** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- SSH and Remote Access
- A **practice Ubuntu 22.04/24.04 VM** with `sudo` and working `apt` repositories
- Do **not** run experimental holds/removes on a shared production server

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain what a package manager does and name the common tools per distro family
- [ ] Update apt metadata and install a package with `apt-get`
- [ ] Query package policy, files, and status with `apt` / `dpkg`
- [ ] Hold and unhold a package, then remove it cleanly
- [ ] Describe how kernel updates and image builds relate to patching

Architecture

Repositories publish packages. The package manager resolves dependencies, installs files, and tracks state in a local database (`dpkg` / `rpm`).

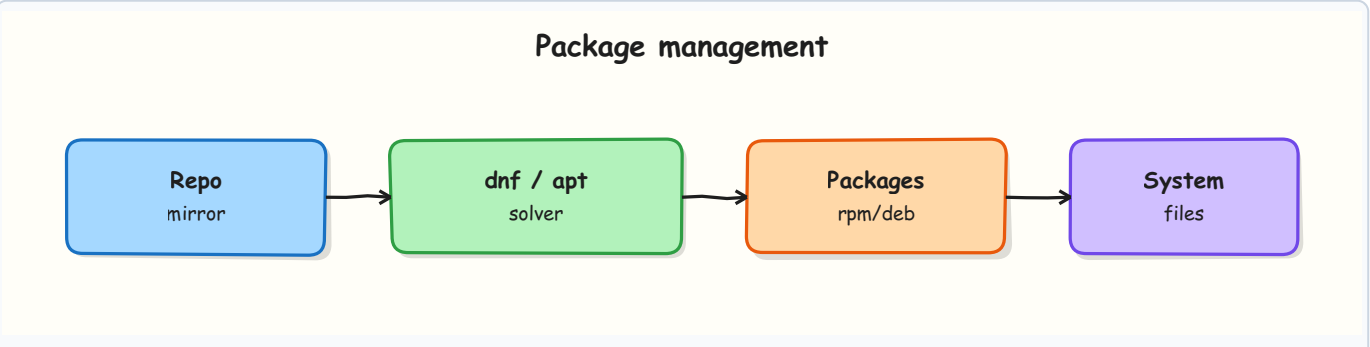

Figure 16.1: Package Management

Theory

What it is

Family	Install / update	Query	Database
Debian / Ubuntu	`apt` / `apt-get`	`apt policy`, `dpkg -l`	`dpkg`
RHEL / Fedora / Amazon Linux	`dnf` (or `yum`)	`rpm -q`, `dnf list`	`rpm`
SUSE	`zypper`	`zypper info`, `rpm -q`	`rpm`

Packages bring version metadata, dependencies, and a file inventory the OS can verify.

```
sudo apt-get update
apt-cache policy curl
dpkg -l curl
```

Why it matters

Manual binaries under `/usr/local` drift and are hard to patch. Unpatched kernels and libraries are a major host vulnerability class. Interviewers and hiring managers expect you to install tools the distro way, know how to check versions, and explain holds/pins when a change must wait.

How it works

1. **Refresh metadata** — `apt-get update`, `dnf check-update`, `zypper refresh`.
2. **Install** — `apt-get install`, `dnf install`, `zypper install`.
3. **Query** — `apt policy`, `dpkg -L package`, `rpm -ql package`.
4. **Upgrade** — apply security and bugfix releases; kernel updates usually need a **reboot**.
5. **Hold / pin** — stop a package from upgrading until you are ready (`apt-mark hold` on Ubuntu).
6. **Remove** — `apt-get remove` (keep config) or `purge` (remove config too); clean unused deps with `autoremove`.

```
sudo apt-get install -y tree
apt-mark showhold
sudo apt-mark hold tree
```

Prefer distro packages for system services. Use containers or language tools (pip, npm) for app runtimes when isolation matters. Snap/Flatpak are optional; many servers disable snap for simplicity.

Key concepts and comparisons

Action	apt (Ubuntu)	dnf (RHEL-like)
Refresh	`apt-get update`	`dnf check-update` / `makecache`
Install	`apt-get install pkg`	`dnf install pkg`
Remove	`apt-get remove pkg`	`dnf remove pkg`
Hold	`apt-mark hold pkg`	`dnf versionlock` (plugin)
Files list	`dpkg -L pkg`	`rpm -ql pkg`

Pattern	Prefer when	Avoid when
Distro package	System tools, daemons	You need a bleeding-edge app version
Container image	App runtime isolation	Simple host CLI tools
Manual binary in `/usr/local`	Rare emergency	Default for every tool

Common pitfalls

- Running `apt-get upgrade` on production without a window or snapshot.
- Forgetting `apt-get update` so installs fail or get stale versions.
- Holding packages forever and missing security fixes.
- Mixing random third-party `.deb` files without trusting the source.
- Assuming the same package name exists on every distro (`apache2` vs `httpd`).

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, install `tree`, prove it with queries, place and remove an apt hold, remove the package, and save a package evidence archive under `~/rebash-linux/lab16`.

Prerequisites

- Ubuntu 22.04/24.04 with `sudo` and working internet/apt mirrors
- Snapshot the VM first if your hypervisor supports it

Lab environment

Workspace: `~/rebash-linux/lab16`

```
mkdir -p ~/rebash-linux/lab16 && cd ~/rebash-linux/lab16
set -euo pipefail
whoami | tee admin-user.txt
. /etc/os-release
printf '%s\n' "$NAME" "$VERSION_ID" | tee os-release.txt
sudo -n true 2>/dev/null || sudo -v
```

Expected output: `os-release.txt` shows Ubuntu (or Debian).

Real-world scenario

Your team standardises a small diagnostic tool on bastion hosts. Change control asks you to install it from the distro repository, record the version, hold it during an application freeze week, then unhold and remove it from a decommissioned practice host — with command output attached to the ticket.

Step-by-step tasks

Task 1 – Update metadata and install `tree`

```
cd ~/rebash-linux/lab16
set -euo pipefail

sudo apt-get update -y | tee apt-update.txt
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y tree | tee apt-install-tree.txt

command -v tree | tee tree-path.txt
tree --version | tee tree-version.txt
test -x "$(command -v tree)"
```

Expected output: `tree` is on `PATH`; `tree-version.txt` shows a version string.

Task 2 – Query policy, files, and hold

```
cd ~/rebash-linux/lab16
set -euo pipefail

apt-cache policy tree | tee apt-policy-tree.txt
dpkg -l tree | tee dpkg-l-tree.txt
dpkg -L tree | head -n 40 | tee dpkg-L-tree-head.txt
grep -E '/usr/bin/tree|/bin/tree' dpkg-L-tree-head.txt

sudo apt-mark hold tree | tee apt-hold.txt
apt-mark showhold | tee apt-showhold.txt
grep -qx tree apt-showhold.txt

# Simulate “would upgrade” awareness
apt-get -s upgrade 2>/dev/null | tee apt-sim-upgrade.txt || true
```

Expected output: policy shows an installed version; `apt-showhold.txt` lists `tree`.

Task 3 – Unhold, remove, evidence pack

```
cd ~/rebash-linux/lab16
set -euo pipefail

sudo apt-mark unhold tree | tee apt-unhold.txt
sudo DEBIAN_FRONTEND=noninteractive apt-get remove -y tree | tee apt-remove-tree.txt

# Confirm removed from PATH (or not a regular file)
if command -v tree >/dev/null 2>&1; then
```

```
  echo "WARN: tree still on PATH — check other packages" | tee tree-after-remove.txt
else
  echo "tree removed from PATH" | tee tree-after-remove.txt
fi
dpkg -l tree 2>&1 | tee dpkg-after-remove.txt || true

# Optional clean of unused deps (safe on practice VM)
sudo DEBIAN_FRONTEND=noninteractive apt-get autoremove -y | tee apt-autoremove.txt || true

# Note other families for the ticket (documentation only)
cat > other-distros.txt << 'EOF'
RHEL-like: sudo dnf install -y tree && rpm -q tree && sudo dnf remove -y tree
SUSE:      sudo zypper install -y tree && rpm -q tree
EOF

tar -czf package-evidence.tgz \
  admin-user.txt os-release.txt apt-update.txt apt-install-tree.txt \
  tree-path.txt tree-version.txt apt-policy-tree.txt dpkg-l-tree.txt \
  dpkg-L-tree-head.txt apt-hold.txt apt-showhold.txt apt-sim-upgrade.txt \
  apt-unhold.txt apt-remove-tree.txt tree-after-remove.txt \
  dpkg-after-remove.txt apt-autoremove.txt other-distros.txt
ls -l package-evidence.tgz | tee evidence-ls.txt
```

Expected output: hold removed; package removed (or marked not installed); `package-evidence.tgz` exists.

Validation steps

- [ ] `apt-get update` completed without repository errors
- [ ] `tree` installed and version recorded, then removed
- [ ] `apt-mark hold` / `unhold` proven in output files
- [ ] `package-evidence.tgz` exists under `~/rebash-linux/lab16`

Common errors and fixes

Error	Cause	Fix
`Unable to locate package`	Stale cache / wrong suite	`sudo apt-get update`; check `/etc/apt/sources.list`
`Could not get lock`	Another apt process	Wait for unattended-upgrades; `ps aux | grep apt`
Hold ignored in some tools	Used wrong mark command	Use `apt-mark hold`; verify with `apt-mark showhold`
Removed package still “installed”	Config left behind	Use `apt-get purge` if you also want config removed
Breaks on production	Broad `upgrade`	Use staged patches and snapshots

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Install `jq`, record `apt-cache policy jq` and `jq --version`, hold `jq`, prove hold with `apt-mark showhold`, then unhold and remove `jq`. Save outputs as `challenge-jq-*.txt` in the lab directory.

Learning outcomes

- Installed and queried a distro package with apt/dpkg
- Used apt hold/unhold during a simulated freeze
- Removed the package and packed ticket evidence
- Mapped apt actions to dnf/zypper for other distros

Cleanup

```
cd ~/rebash-linux/lab16
set -euo pipefail
sudo apt-mark unhold tree 2>/dev/null || true
sudo apt-mark unhold jq 2>/dev/null || true
sudo DEBIAN_FRONTEND=noninteractive apt-get remove -y tree jq 2>/dev/null || true
# Keep package-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab16/`
- [ ] You can explain apt vs dnf vs zypper at a high level
- [ ] You know why kernel upgrades often need a reboot

- [ ] You can describe the risk of never patching vs holding forever

Code Walkthrough

Production package hygiene usually follows:

1. **Refresh** metadata before install
2. **Install** from trusted distro/repos only
3. **Record** versions in image builds or tickets
4. **Hold** only with an expiry plan
5. **Patch** in windows; reboot for kernel changes

Configuration management (Ansible, cloud-init) should own desired packages.

Security Considerations

- Prefer official mirrors and signed repositories
- Review third-party apt sources before adding them
- Patch regularly; track Common Vulnerabilities and Exposures (CVE) for critical packages
- Do not run random install scripts from the internet as root
- Limit who can run package installs with sudo rules

Common Mistakes

Skipping apt-get update

You install stale or missing packages. **Fix:** always update metadata first on practice and in automation.

Permanent holds with no review

Security fixes never arrive. **Fix:** document holds and remove them after the freeze.

curl | bash installers for system tools

Hard to audit and reverse. **Fix:** prefer distro packages or verified artefacts.

Forgetting reboot after kernel update

Host still runs the old kernel. **Fix:** plan reboot; confirm with `uname -r`.

Best Practices

- Build golden images with required packages baked in
- Use unattended security updates carefully on servers
- Keep a short allow-list of approved packages for bastions
- Prefer `DEBIAN_FRONTEND=noninteractive` in scripts
- Clean unused packages to shrink attack surface

Troubleshooting

Symptom	Likely cause	Fix
Hash sum mismatch	Mirror glitch	Retry `apt-get update`; switch mirror if needed
Unmet dependencies	Mixed releases / broken pins	Read apt error; avoid mixing suites
`dpkg was interrupted`	Partial upgrade	`sudo dpkg --configure -a`
Package held back	Hold or phased updates	`apt-mark showhold`; read apt notes
Wrong package name	Distro difference	Search (`apt-cache search`, `dnf search`)

Summary

Package managers keep Linux software installable, queryable, and patchable. On Ubuntu, practise `apt-get update`, install, `apt-cache policy`, `apt-mark hold`, and clean removal — then map the same ideas to `dnf` and `zypper`. Next, schedule recurring work in Scheduling with cron, at, and Timers.

Interview Questions

INTERVIEW PRACTICE

1. What problem does a package manager solve compared with copying binaries by hand?

Answer

It tracks **versions**, **dependencies**, and **installed files**, and it uses signed repositories. That makes installs repeatable, upgrades safer, and removal cleaner than dropping unknown binaries into `/usr/local`.

2. What is the difference between `apt-get update` and `apt-get upgrade`?

Answer

`update` refreshes package **metadata** (what versions are available). `upgrade` installs newer versions of packages already on the system. You usually update first, then upgrade in a planned window.

3. How do you stop one package from upgrading during a freeze week on Ubuntu?

Answer

Use `sudo apt-mark hold packagename`, verify with `apt-mark showhold`, and document why. After the freeze, `apt-mark unhold packagename` and patch. Do not hold critical security packages forever without a plan.

4. Why do kernel package updates often require a reboot?

Answer

The running kernel is already loaded in memory. Installing a new kernel package updates files on disk, but the host keeps running the old kernel until reboot. Confirm with `uname -r` after reboot.

5. How would you find which package owns `/usr/bin/curl` on Ubuntu vs RHEL?

Answer

On Ubuntu/Debian: `dpkg -S /usr/bin/curl`. On RHEL-like systems: `rpm -qf /usr/bin/curl`. This helps when a file is broken or you need to reinstall the correct package.

6. When are snap or Flatpak appropriate on a server?

Answer

Rarely for classic server daemons. They are more common for desktop apps. Many production servers prefer apt/dnf packages or containers for isolation. If snap is unused, teams often disable it to reduce complexity.

7. How do golden images and package management work together in cloud fleets?

Answer

Bake a known package set into the image (and record versions). Instances launch consistent. Patching then happens via new images or controlled in-place upgrades. This beats unique “snowflake” hosts where someone installed tools by hand months ago.

Scheduling with cron, at, and Timers
MODULE 11 · SCHEDULING

Chapter 17. Scheduling with cron, at, and Timers

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/scheduling-cron-at-and-timers/

Overview

Linux schedules deferred and recurring work with **cron** (calendar tables), **at** (one-shot jobs), and **systemd timers** (unit-based schedules). Backups, certificate checks, report scrapes, and cleanup scripts all need a reliable schedule and **visible logs**. Silent failures are a common cause of "stale data" incidents.

Cron entries set minute, hour, day of month, month, and day of week, plus a command. The `at` command queues a job for a future time. **systemd timers** activate an associated `.service` unit using calendar or monotonic expressions and integrate with `journalctl` and dependencies such as `network-online.target`. In Cloud and DevOps work you will see all three: user crontabs on bastions, `/etc/cron.d/` jobs from packages, and timers for software managed by systemd.

Jobs often fail because the scheduler environment is not your interactive shell — especially `PATH` and working directory. Always use absolute paths, redirect output to a log file, and prove the next/last run time. Prefer timers when you need randomised delay, ordering, or unified failure logs; keep cron for simple per-user tasks.

This is **Tutorial 17** in **Module 11: Scheduling & Automation** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Package Management
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Comfort with basic `systemctl` (see Module 7 if needed)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain when to use cron vs `at` vs systemd timers
- [ ] Install a user crontab job that writes a log with absolute paths
- [ ] Queue and verify a one-shot `at` job
- [ ] Create a systemd `.service` + `.timer`, enable it, and prove it with `list-timers` and journal logs
- [ ] Clean up lab schedules without leaving orphan jobs

Architecture

Schedulers trigger work later. Cron and `at` run commands directly; systemd timers activate service units and log through the journal.

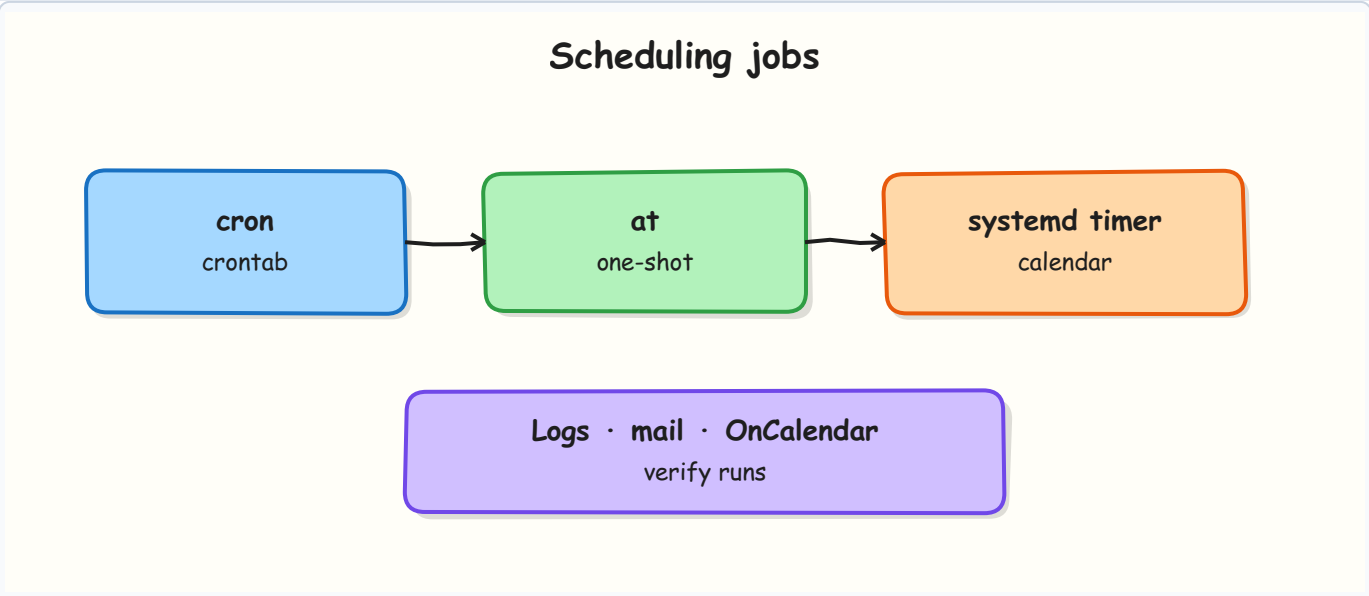

Figure 17.1: Scheduling with cron, at, and Timers

Theory

What it is

Mechanism	Best for
User `crontab`	Personal or per-account recurring jobs
`/etc/cron.d/`	Package- and system-shipped jobs
`at`	One-shot “run this once later”
systemd timer	Services already modelled as units; better journal integration

```
crontab -l
atq
systemctl list-timers --all
```

Why it matters

Missed backups and stale caches often trace to a cron job that never ran, ran with empty `PATH`, or wrote errors nobody read. Timers give clearer “next/last” status for operations teams. Interviewers expect you to debug schedules with logs, not guesses.

How it works

1. **User cron** — `crontab -e` edits your table; each line is schedule + command.
2. **System cron** — `/etc/crontab` and `/etc/cron.d/*` (include a user field).
3. **Environment** — set `PATH` in the crontab or use full paths (`/usr/bin/date`).
4. **Logging** — redirect `>> /path/log 2>&1`; do not rely only on local mail.
5. **`at`** — `echo command | at now + 1 minute`; manage with `atq` / `atrm`.
6. **Timers** — write `foo.service` (what to run) and `foo.timer` (when); `systemctl enable --now foo.timer`; check `systemctl list-timers` and `journalctl -u foo.service`.

```
# Example cron: every day at 02:15
# 15 2 * * * /usr/bin/date >> /var/tmp/date.log 2>&1
```

Key concepts and comparisons

Need	Prefer
Simple user job	`crontab`
Run once tomorrow	`at`
App shipped as systemd unit	`.timer`
Randomised load across fleet	timer `RandomizedDelaySec=`
Catch-up after downtime	timer `Persistent=true`

Cron schedule fields: `minute hour day-of-month month day-of-week`.

Common pitfalls

- Relative commands without `PATH` (`date` works in your shell, fails in cron).
- Editing `/etc/crontab` with the wrong number of fields (system crontab includes username).
- No log redirection — failures are invisible.
- Creating a timer without enabling it (`enable --now`).
- Leaving lab cron entries on shared hosts.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Create a user cron job, an `at` job, and a systemd user-space **system** timer that appends timestamps to lab log files. Prove each schedule with files and `list-timers`, then clean up. Workspace: `~/rebash-linux/lab17`.

Prerequisites

- Ubuntu 22.04/24.04 with `sudo`
- Packages: `cron`, `at`, `systemd` (install if missing)

Lab environment

Workspace: `~/rebash-linux/lab17`

```
mkdir -p ~/rebash-linux/lab17 && cd ~/rebash-linux/lab17
set -euo pipefail
whoami | tee admin-user.txt
sudo -n true 2>/dev/null || sudo -v

sudo apt-get update -y
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y cron at
sudo systemctl enable --now cron
sudo systemctl enable --now atd
sudo systemctl is-active cron | tee cron-active.txt
sudo systemctl is-active atd | tee atd-active.txt
```

Expected output: `cron` and `atd` are `active`.

Real-world scenario

Ops wants a small health stamp every minute for a practice app, a one-shot reminder job, and a systemd timer for a cleanup script that must show up in `list-timers` and the journal. You implement all three on a practice VM with clear log files for the change ticket.

Step-by-step tasks

Task 1 – User crontab job with absolute paths

```
cd ~/rebash-linux/lab17
set -euo pipefail

LAB="$HOME/rebash-linux/lab17"
LOG="$LAB/cron-heartbeat.log"
SCRIPT="$LAB/cron-heartbeat.sh"

cat > "$SCRIPT" << EOF
```

```
#!/bin/bash
set -euo pipefail
/usr/bin/date -Is >> "$LOG"
echo "cron-ok" >> "$LOG"
EOF
chmod 755 "$SCRIPT"

# Install crontab: every minute (lab only — remove in cleanup)
crontab -l 2>/dev/null | grep -v 'REBASH-LAB17' > "$LAB/crontab.prev" || true
{
  cat "$LAB/crontab.prev" 2>/dev/null || true
  echo "# REBASH-LAB17"
  echo "* * * * * $SCRIPT"
} | crontab -

crontab -l | tee crontab-installed.txt
grep -q REBASH-LAB17 crontab-installed.txt

echo "Waiting up to 75s for first cron run..."
for i in $(seq 1 15); do
  if [ -f "$LOG" ] && grep -q cron-ok "$LOG"; then
    break
  fi
  sleep 5
done
test -f "$LOG"
grep cron-ok "$LOG" | tee cron-heartbeat-proof.txt
```

Expected output: `cron-heartbeat.log` contains a timestamp and `cron-ok` within about one minute.

Task 2 – One-shot `at` job

```
cd ~/rebash-linux/lab17
set -euo pipefail

AT_LOG="$HOME/rebash-linux/lab17/at-job.log"
rm -f "$AT_LOG"

echo "/usr/bin/date -Is > $AT_LOG; echo at-ok >> $AT_LOG" | at now + 1 minute 2>&1 | tee at-submit.txt
atq | tee atq.txt
test -s atq.txt

echo "Waiting up to 90s for at job..."
for i in $(seq 1 18); do
  if [ -f "$AT_LOG" ] && grep -q at-ok "$AT_LOG"; then
    break
  fi
  sleep 5
done
test -f "$AT_LOG"
cat "$AT_LOG" | tee at-job-proof.txt
grep -q at-ok at-job-proof.txt
```

Expected output: `atq` showed a job; `at-job.log` contains `at-ok`.

Task 3 – systemd service + timer

```
cd ~/rebash-linux/lab17
set -euo pipefail

LAB="$HOME/rebash-linux/lab17"
TIMER_LOG="$LAB/timer-heartbeat.log"
UNIT_SCRIPT="$LAB/timer-heartbeat.sh"

cat > "$UNIT_SCRIPT" << EOF
#!/bin/bash
set -euo pipefail
/usr/bin/date -Is >> "$TIMER_LOG"
echo "timer-ok" >> "$TIMER_LOG"
EOF
chmod 755 "$UNIT_SCRIPT"

# System units (need sudo)
sudo tee /etc/systemd/system/rebash-lab17.service >/dev/null << EOF
[Unit]
Description=REBASH lab17 timer payload
[Service]
Type=oneshot
User=$USER
ExecStart=$UNIT_SCRIPT
EOF

sudo tee /etc/systemd/system/rebash-lab17.timer >/dev/null << 'EOF'
[Unit]
Description=REBASH lab17 periodic timer
[Timer]
```

```
OnBootSec=30s
OnUnitActiveSec=60s
AccuracySec=1s
Unit=rebash-lab17.service
[Install]
WantedBy=timers.target
EOF

sudo systemctl daemon-reload
sudo systemctl enable --now rebash-lab17.timer
systemctl list-timers --all | grep rebash-lab17 | tee list-timers.txt
test -s list-timers.txt

# Trigger once immediately for faster proof
sudo systemctl start rebash-lab17.service
sleep 1
test -f "$TIMER_LOG"
grep timer-ok "$TIMER_LOG" | tee timer-heartbeat-proof.txt
journalctl -u rebash-lab17.service -n 20 --no-pager | tee journal-timer.txt

tar -czf schedule-evidence.tgz \
  admin-user.txt cron-active.txt atd-active.txt \
  crontab-installed.txt cron-heartbeat.log cron-heartbeat-proof.txt \
  at-submit.txt atq.txt at-job.log at-job-proof.txt \
  list-timers.txt timer-heartbeat.log timer-heartbeat-proof.txt journal-timer.txt \
  cron-heartbeat.sh timer-heartbeat.sh
ls -l schedule-evidence.tgz | tee evidence-ls.txt
```

Expected output: `list-timers.txt` shows `rebash-lab17.timer`; `timer-heartbeat.log` contains `timer-ok`; journal shows the service run.

Validation steps

- [ ] User crontab contains the REBASH-LAB17 line and log updates
- [ ] `at` job wrote `at-job.log`
- [ ] `systemctl list-timers` shows `rebash-lab17.timer`
- [ ] `schedule-evidence.tgz` exists under `~/rebash-linux/lab17`

Common errors and fixes

Error	Cause	Fix
Cron never writes log	Wrong path / cron not running	Use absolute paths; `systemctl status cron`
`at: command not found`	Package missing	`sudo apt-get install -y at`
`atd` inactive	Service not started	`sudo systemctl enable --now atd`
Timer not listed	Not enabled	`sudo systemctl enable --now name.timer`
Permission denied in script	Wrong owner / mode	`chmod 755` script; set `User=` in service

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Add `Persistent=true` to the timer (or create `rebash-lab17-persist.timer`) and document in `challenge-persistent.txt` what Persistent means (run missed jobs after downtime). Trigger with `systemctl start rebash-lab17.service` and attach a new journal snippet. Remove any extra units in Cleanup.

Learning outcomes

- Installed a proven user cron job with logging
- Queued and verified an `at` job
- Enabled a systemd timer with journal evidence
- Packed schedule proof for a change ticket

Cleanup

```
cd ~/rebash-linux/lab17
set -euo pipefail

# Remove lab cron lines
crontab -l 2>/dev/null | grep -v 'REBASH-LAB17' | grep -v 'cron-heartbeat.sh' | crontab - || crontab -r 2>/dev/null || true

# Remove pending at jobs owned by you (careful on shared hosts)
```

```
atq | awk '{print $1}' | while read -r id; do atrm "$id" 2>/dev/null || true; done

sudo systemctl disable --now rebash-lab17.timer 2>/dev/null || true
sudo rm -f /etc/systemd/system/rebash-lab17.service /etc/systemd/system/rebash-lab17.timer
sudo systemctl daemon-reload
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab17/`
- [ ] You can explain cron vs at vs timers
- [ ] You know why absolute paths matter in cron
- [ ] You can find next/last run for a timer

Code Walkthrough

Production scheduling habits:

1. **Decide** mechanism (cron / at / timer)
2. **Write** a small script with absolute paths
3. **Log** stdout/stderr to a file or journal
4. **Enable** and prove next/last run
5. **Alert** on missing output (monitoring), not only on process start

Security Considerations

- Do not put secrets in world-readable cron scripts
- Limit who can use `at` / crontab (`/etc/cron.allow`) on shared hosts
- Run timers as the least-privileged user that still works
- Review `/etc/cron.d` after package installs
- Treat unexpected new scheduled jobs as a security signal

Common Mistakes

Relying on interactive PATH in cron

Commands “not found”. **Fix:** full paths or set `PATH=` at the top of the crontab.

No log file

Failures are invisible. **Fix:** `>> /path/log 2>&1` or use a timer + `journalctl`.

Creating a .timer but forgetting enable --now

Nothing runs. **Fix:** `systemctl enable --now name.timer` and check `list-timers`.

Every-minute cron left forever on production

Noise and load. **Fix:** use sane intervals; remove lab jobs in cleanup.

Best Practices

- Prefer systemd timers for software you already ship as units
- Use `RandomizedDelaySec` to avoid thundering herds
- Keep scripts idempotent (safe if run twice)
- Store system jobs in git / config management, not only on the host
- Monitor “last success time”, not only “timer active”

Troubleshooting

Symptom	Likely cause	Fix
Cron silent	Service down / bad schedule	`systemctl status cron`; `crontab -l`
`at` job stuck	`atd` down	Start `atd`; check `atq`
Timer inactive	Not enabled / wrong WantedBy	`enable --now`; `daemon-reload`
Script fails only in scheduler	Env / cwd / permissions	Log env; use absolute paths
Duplicate runs	Overlapping long jobs	Add locking (`flock`) in the script

Summary

Cron, `at`, and systemd timers all schedule work — choose based on one-shot vs recurring and how you want to observe failures. Use absolute paths, write logs, and prove schedules with crontab listings, `atq`, and `systemctl list-timers`. Next, follow those logs in Logging — syslog, journald, and logrotate.

Interview Questions

INTERVIEW PRACTICE

1. When do you choose a systemd timer instead of cron?

Answer

Prefer a **timer** when the work is modelled as a systemd service, when you need journal integration, dependencies (`After=network-online.target`), randomised delay, or `Persistent=` catch-up. Cron remains fine for simple per-user jobs and many classic admin scripts.

2. Why do cron jobs fail with "command not found" even though the command works in your shell?

Answer

Cron uses a **minimal environment**, often a short `PATH`. Your interactive shell has more directories. Fix by using absolute paths (`/usr/bin/python3`) or setting `PATH=` in the crontab, and by logging stderr.

3. What is the difference between user crontab and `/etc/cron.d/`?

Answer

A **user crontab** (`crontab -e`) runs as that user and has five schedule fields plus the command. Files in **`/etc/cron.d/`** are system drop-ins and usually include a **username** field (six fields before the command). Packages often install files under `/etc/cron.d/`.

4. How do you prove a systemd timer is armed and actually running the work?

Answer

Use `systemctl list-timers` (next/last), `systemctl status name.timer`, and `journalctl -u name.service` for the payload. Optionally start the service once manually to prove the unit works independent of the schedule.

5. What is `at` good for that cron is not?

Answer

`at` is for **one-shot** jobs at a future time ("run this once in 20 minutes"). Cron is for **recurring** calendar schedules. Use `atq` / `atrm` to list and remove pending `at` jobs.

6. What does `Persistent=true` on a timer mean?

Answer

If the system was powered off when a calendar timer should have fired, systemd can run the missed job when the machine comes back (within limits). Useful for daily maintenance on machines that are not always on.

7. How would you stop a runaway every-minute cron in production safely?

Answer

List with `crontab -l` or inspect `/etc/cron.d`, remove or comment the line, verify with `crontab -l`, and check logs for impact. For systemd, `systemctl disable --now name.timer`. Communicate in the incident channel; do not reboot as the first step.

Logging — syslog, journald, and logrotate
MODULE 12 · LOGGING & MONITORING

Chapter 18. Logging — syslog, journald, and logrotate

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/logging-syslog-journald-logrotate/

Overview

If it is not logged, it did not happen in the incident review. On modern Ubuntu, **journald** collects structured logs from the kernel, services, and stdout/stderr of systemd units. You read them with `journalctl`. Many programs still write classic text logs under `/var/log`, often via **syslog** (rsyslog or syslog-ng). **logrotate** compresses and retires those text files so disks do not fill forever.

In Cloud and DevOps work you follow a unit with `journalctl -u`, correlate boot windows with `--since`, and ensure application file logs have a **logrotate** rule. Silent log growth is a common cause of disk-full outages. In this tutorial you will query the journal, write a sample app log, add a logrotate configuration, force a rotation, and save proof under `~/rebash-linux/lab18`.

In production, ship important logs to a central system, restrict who can read authentication logs, and alert when journal or `/var/log` mounts grow too fast.

This is **Tutorial 18** in **Module 12: Logging & Monitoring** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Scheduling with cron, at, and Timers
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Packages: `systemd` (journalctl), `logrotate` (usually installed)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Query journald with `journalctl` filters (`-u`, `--since`, `-p`)
- [ ] Explain how journald relates to classic `/var/log` syslog files
- [ ] Create a logrotate rule for an application log and force a rotation
- [ ] Prove rotation with timestamps and rotated filenames
- [ ] Pack evidence under `~/rebash-linux/lab18`

Architecture

Applications and units emit logs to journald and/or text files; logrotate manages text file retention; operators query with journalctl and inspect `/var/log`.

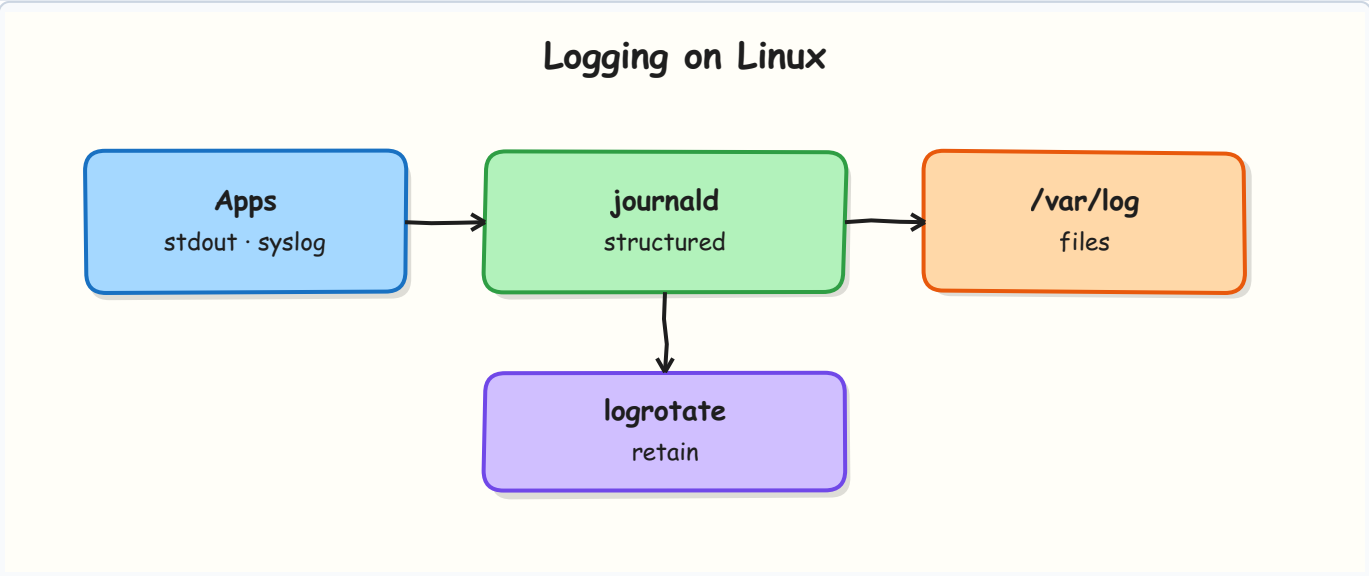

Figure 18.1: Logging

Theory

What it is

journald stores binary, indexed logs. **syslog** daemons traditionally write text files such as `/var/log/syslog`. **logrotate** runs on a schedule (often daily via cron/timers) to rotate, compress, and delete old log files based on rules in `/etc/logrotate.conf` and `/etc/logrotate.d/`.

```
journalctl -xe
journalctl -u ssh.service -n 20 --no-pager
ls /var/log
```

Why it matters

Without retention, logs fill disks. Without query skills, incidents take longer. Central logging (Elastic, Loki, CloudWatch, and similar) still starts with correct host-side collection and rotation.

How it works

1. Units log to the journal (and sometimes to files).
2. Operators filter with `journalctl`.
3. File logs grow under `/var/log` or app directories.
4. logrotate renames/compresses and may signal the app (`postrotate`).

Tool	Best for
`journalctl`	systemd units, boots, priorities
`/var/log/*`	Classic app/syslog text
`logrotate`	Retention for text logs

Common pitfalls

- Forgetting `Storage=` / vacuum settings until the journal fills `/var`.
- Rotating a log without telling the app to reopen the file (needs `postrotate` or copytruncate carefully).
- Reading only `/var/log/syslog` when the service only logs to the journal.
- World-readable logs that contain secrets.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Query journald, create a sample application log with a dedicated logrotate rule, force rotation, and save evidence under `~/rebash-linux/lab18`.

Prerequisites

- Ubuntu with `sudo`, `journalctl`, `logrotate`

Lab environment

Workspace: `~/rebash-linux/lab18`

```
mkdir -p ~/rebash-linux/lab18 && cd ~/rebash-linux/lab18
set -euo pipefail
test -n "$(command -v journalctl)"
test -n "$(command -v logrotate)"
journalctl --version | head -n 1 | tee journalctl-version.txt
```

Expected output: `journalctl` and `logrotate` exist; version file written.

Real-world scenario

A small app writes to `/var/log/rebash-lab18/app.log`. Disk alerts fired last month because nothing rotated that file. You add a logrotate rule, prove a forced rotation, and show how to pull matching journal lines for the same host window.

Step-by-step tasks

Task 1 – Query journald

```
cd ~/rebash-linux/lab18
set -euo pipefail

journalctl -b -n 30 --no-pager | tee journal-boot-tail.txt
journalctl -p err..alert -n 20 --no-pager | tee journal-err.txt || true
journalctl --list-boots | tee journal-boots.txt
# Logger message into syslog/journal
logger -t rebash-lab18 "rebash lab18 marker $(date -Is)"
sleep 1
journalctl -t rebash-lab18 -n 5 --no-pager | tee journal-marker.txt
grep -F 'rebash lab18 marker' journal-marker.txt
```

Expected output: `journal-marker.txt` contains your logger line; boot list captured.

Task 2 – Application log + logrotate rule

```
cd ~/rebash-linux/lab18
set -euo pipefail

sudo mkdir -p /var/log/rebash-lab18
sudo chown "$USER":"$USER" /var/log/rebash-lab18
printf 'line-%s\n' $(seq 1 200) > /var/log/rebash-lab18/app.log
wc -l /var/log/rebash-lab18/app.log | tee app-lines-before.txt

sudo tee /etc/logrotate.d/rebash-lab18 >/dev/null << 'EOF'
/var/log/rebash-lab18/*.log {
    daily
    missingok
    rotate 3
    compress
    delaycompress
    notifempty
    copytruncate
}
EOF

# Syntax check (logrotate -d is dry-run debug)
sudo logrotate -d /etc/logrotate.d/rebash-lab18 2>&1 | tee logrotate-debug.txt
grep -Ei 'rebash-lab18|app.log' logrotate-debug.txt
```

Expected output: `app.log` has 200 lines; debug output mentions the lab path; rule file installed.

Task 3 – Force rotation and evidence pack

```
cd ~/rebash-linux/lab18
set -euo pipefail

# Force rotation even if not "daily" yet
sudo logrotate -f /etc/logrotate.d/rebash-lab18
```

```
ls -la /var/log/rebash-lab18/ | tee log-dir-after.txt
# With copytruncate, original app.log remains; a rotated copy appears (name varies)
test -f /var/log/rebash-lab18/app.log
ls /var/log/rebash-lab18/app.log* | tee rotated-names.txt
test "$(ls /var/log/rebash-lab18/app.log* | wc -l)" -ge 2

tar -czf logging-evidence.tgz \
  journalctl-version.txt journal-boot-tail.txt journal-err.txt \
  journal-boots.txt journal-marker.txt \
  app-lines-before.txt logrotate-debug.txt log-dir-after.txt rotated-names.txt
ls -l logging-evidence.tgz | tee evidence-ls.txt
```

Expected output: more than one `app.log*` name after forced rotate; evidence archive exists.

Validation steps

- [ ] `journalctl -t rebash-lab18` shows the marker
- [ ] `/etc/logrotate.d/rebash-lab18` exists
- [ ] Forced rotation created a rotated file alongside `app.log`
- [ ] `logging-evidence.tgz` exists under `~/rebash-linux/lab18`

Common errors and fixes

Error	Cause	Fix
Permission denied on `/var/log`	Need root for system paths	Use `sudo` for mkdir/logrotate
No rotated file after `-f`	Rule path mismatch / `notifempty`	Confirm path; ensure log has content
`journalctl` empty for tag	logger failed / wrong filter	Re-run `logger`; try `journalctl -n 50`
Disk still filling	Journal vacuum not set / other logs	Check `journalctl --disk-usage`; rotate other paths

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `/etc/logrotate.d/rebash-lab18-size` that rotates `/var/log/rebash-lab18/app.log` when it exceeds `1k` (`size 1k`), force it after appending more lines, and save `ls -la` output to `size-rotate.txt`. Remove the extra rule in Cleanup.

Learning outcomes

- Filtered journald by boot, priority, and tag
- Installed a logrotate drop-in and forced rotation
- Proved rotated filenames on disk
- Saved logging evidence for a ticket

Cleanup

```
cd ~/rebash-linux/lab18
set -euo pipefail
sudo rm -f /etc/logrotate.d/rebash-lab18 /etc/logrotate.d/rebash-lab18-size
sudo rm -rf /var/log/rebash-lab18
# Keep logging-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab18/` with evidence files
- [ ] You can explain journald vs text syslog files
- [ ] You know why every long-lived app log needs rotation
- [ ] You can find a service's recent logs with `journalctl -u`

Code Walkthrough

Incident logging path:

1. Identify the unit or log file
2. `journalctl -u … --since …`
3. Check `/var/log` and disk (`df`)

4. Confirm logrotate rules exist and ran
5. Forward critical logs centrally

Security Considerations

- Restrict permissions on auth and application logs
- Do not log secrets (tokens, passwords)
- Limit who can run `journalctl` as root on multi-tenant hosts
- Protect log shipping credentials
- Retain logs long enough for investigations, not forever on the host disk

Common Mistakes

Only watching `/var/log/syslog`

Many units log only to the journal. **Fix:** start with `journalctl -u servicename`.

Rotating without reopening the file

The app may keep writing to a deleted inode. **Fix:** use a proper `postrotate` reload, or carefully use `copytruncate` when appropriate.

No vacuum on journal storage

`/var` fills with journal data. **Fix:** configure `SystemMaxUse=` / vacuum; monitor `journalctl --disk-usage`.

World-readable app logs

Sensitive data leaks. **Fix:** mode `640` / `640`-style ownership to the app group.

Best Practices

- One logrotate file per application under `/etc/logrotate.d/`
- Prefer structured logs and correlation IDs
- Alert on log volume spikes and disk use
- Document where each critical service logs
- Test `logrotate -d` before production changes

Troubleshooting

Symptom	Likely cause	Fix
Empty `journalctl -u`	Wrong unit name / stdout not captured	Check `systemctl status`; unit `StandardOutput`
Log file not rotating	Rule not matched / logrotate not run	`logrotate -d`; check timers/cron
Disk full under `/var/log`	Missing rotate / huge dumps	Rotate, truncate carefully, expand disk
Gaps in logs	Time skew / wiped journal	Fix NTP; check persistence settings
Permission denied reading journal	Non-root / not in `adm` / `systemd-journal`	Use sudo or add user to the right group

Summary

journald holds structured unit logs; syslog files still matter; logrotate keeps text logs from filling disks. Query with purpose, rotate every long-lived file log, and prove it. Next: Host Monitoring — vmstat, iostat, sar.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between journald and classic syslog files?

Answer

journald stores structured, indexed logs from the kernel and systemd units, queried with `journalctl`. **Classic syslog** usually writes plain text files under `/var/log`. Many hosts use both: units → journal, and some apps/syslog rules → text files that **logrotate** manages.

2. How do you show logs for one service since one hour ago?

Answer

Example: `journalctl -u myapp.service --since "1 hour ago" --no-pager`. Add `-p err` to focus on errors, or `-f` to follow. Confirm the exact unit name with `systemctl list-units`.

3. Why must application file logs have logrotate rules?

Answer

Without rotation, logs grow until the filesystem is full, causing outages. logrotate enforces retention (how many copies, compression, when to delete). Every long-lived path under `/var/log` or an app directory needs an owner and a rule.

4. What goes wrong if you rotate a log but the process keeps the old file descriptor open?

Answer

The process keeps writing to the old inode; the new log file stays empty and disk may not free as expected. Fix with a `postrotate` script that signals the app to reopen logs, or use carefully designed `copytruncate` where appropriate.

5. How would you investigate "disk full" when `/var` is the mount that filled?

Answer

Run `df -hT` / `du` on `/var`, check `/var/log` and journal disk use (`journalctl --disk-usage`), identify the largest files, rotate or purge safely, fix the missing retention rule, then confirm `df` recovered. Attach before/after proof.

6. Which journalctl filters do you use most in production?

Answer

Commonly `-u` (unit), `--since` / `--until`, `-b` (boot), `-p` (priority), and `-f` (follow). Tags (`-t`) help for `logger` markers and some apps.

7. How do host logs relate to Kubernetes or container platforms?

Answer

Containers often log to stdout (collected by the runtime) while the node still has journald for kubelet/container runtime and system units. Engineers need both cluster log pipelines and node `journalctl` skills when the node itself is unhealthy.

Host Monitoring — vmstat, iostat, and sar
MODULE 12 · LOGGING & MONITORING

Chapter 19. Host Monitoring — vmstat, iostat, and sar

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/host-monitoring-vmstat-iostat-sar/

Overview

When a host feels "slow", you need numbers for **CPU**, **memory**, and **disk Input/Output (I/O)** — not guesses. Classic Linux tools from the **sysstat** package help: `vmstat` for processes, memory, swap, and CPU; `iostat` for per-disk I/O; `sar` for historical samples (when the sysstat collector is enabled).

Cloud consoles show graphs, but SSH-era tools still matter on jump servers, during outages when agents are down, and in interviews. In this tutorial you will install sysstat if needed, capture `vmstat` / `iostat` / `sar` samples, generate a little controlled load, and save proof under `~/rebash-linux/lab19`.

In production, pair these tools with metrics systems (Prometheus node exporter, CloudWatch, and similar). Use CLI tools to validate what dashboards claim and to dig deeper during incidents.

This is **Tutorial 19** in **Module 12: Logging & Monitoring** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Logging — syslog, journald, and logrotate
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Ability to install `sysstat`

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Read key `vmstat` columns (runnable processes, swap, CPU idle/wait)
- [ ] Use `iostat -xz` to spot high util or await on disks
- [ ] Collect a `sar` sample (or explain when history is empty)
- [ ] Capture before/load/after evidence under `~/rebash-linux/lab19`
- [ ] Relate CLI signals to cloud VM sizing decisions

Architecture

Host monitoring samples kernel counters for CPU, memory, and block I/O so operators can see pressure quickly.

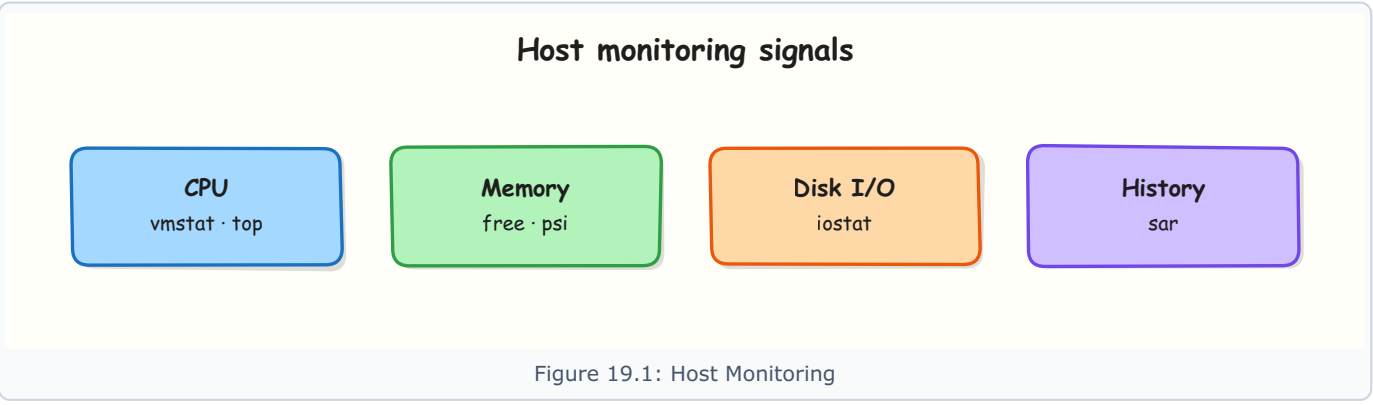

Figure 19.1: Host Monitoring

Theory

What it is

Tool	Focus
`vmstat`	Processes, memory, swap, I/O, CPU
`iostat`	CPU and per-device disk I/O
`sar`	Historical / scheduled activity reports

```
vmstat 1 5
iostat -xz 1 3
sar -u 1 3
```

Why it matters

High **`wa`** (I/O wait) points to storage. High runnable processes (`r` in vmstat) points to CPU contention. Heavy **swap** (`si` / `so`) points to memory pressure. Wrong diagnosis leads to the wrong fix (adding CPU when the disk is the bottleneck).

How it works

1. Take a **baseline** when healthy (or at incident start).
2. Watch a few samples over time (`vmstat 1 10`).
3. Check disks with `iostat -xz`.
4. Use `sar` when sysstat's collector has history (`/var/log/sysstat`).

Signal	Suggests
High `r`, low `id`	CPU busy / contention
High `wa`	Disk or NFS wait
Rising `si` / `so`	Swapping / memory pressure
Device `%util` near 100	Disk saturated

Common pitfalls

- Trusting a single sample (look at trends).
- Ignoring steal time (`st`) on busy hypervisors.
- Assuming `sar` has history when the collector was never enabled.
- Generating heavy load on shared production hosts "to test".

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Install sysstat, capture baseline `vmstat` / `iostat` / `sar` output, create a short controlled CPU+disk load, capture again, and pack evidence under `~/rebash-linux/lab19`.

Prerequisites

- Ubuntu practice VM (not a shared production host)

Lab environment

Workspace: `~/rebash-linux/lab19`

```
mkdir -p ~/rebash-linux/lab19 && cd ~/rebash-linux/lab19
set -euo pipefail
sudo apt-get update -qq
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y sysstat
command -v vmstat; command -v iostat; command -v sar
vmstat -V 2>&1 | head -n 1 | tee sysstat-tools.txt || true
```

Expected output: `vmstat`, `iostat`, and `sar` are on `PATH`.

Real-world scenario

Users say a practice API VM is slow. Before you resize the instance, you capture CPU/memory/disk signals with sysstat tools, create a small reproducible load to see how the counters move, and attach the outputs to the ticket.

Step-by-step tasks

Task 1 – Baseline samples

```
cd ~/rebash-linux/lab19
set -euo pipefail

uptime | tee uptime-before.txt
free -h | tee free-before.txt
vmstat 1 5 | tee vmstat-before.txt
iostat -xz 1 3 | tee iostat-before.txt
sar -u 1 3 | tee sar-u-before.txt
sar -d 1 3 | tee sar-d-before.txt 2>/dev/null || echo 'sar -d unavailable' | tee sar-d-before.txt
```

Expected output: baseline files exist; `vmstat-before.txt` has a header and several data rows.

Task 2 – Controlled load + capture during load

```
cd ~/rebash-linux/lab19
set -euo pipefail

# Short CPU load in background
(timeout 12s bash -c 'while true; do :; done' & echo $! > cpu-load.pid) || true
# Short disk write load
timeout 12s dd if=/dev/zero of=load.bin bs=1M count=256 conv=fdatasync status=none &
echo $! > dd-load.pid || true

sleep 2
vmstat 1 5 | tee vmstat-during.txt
iostat -xz 1 3 | tee iostat-during.txt

# Wait for loads to finish
wait || true
rm -f load.bin
kill "$(cat cpu-load.pid)" 2>/dev/null || true
```

Expected output: `vmstat-during.txt` / `iostat-during.txt` captured while load ran; `load.bin` removed.

Task 3 – After sample + evidence pack

```
cd ~/rebash-linux/lab19
set -euo pipefail

sleep 2
vmstat 1 3 | tee vmstat-after.txt
iostat -xz 1 2 | tee iostat-after.txt
# Optional history location
ls -la /var/log/sysstat 2>/dev/null | tee sysstat-log-dir.txt || echo 'no /var/log/sysstat yet' | tee sysstat-log-dir.txt

# Simple asserts: files non-empty
test -s vmstat-before.txt && test -s vmstat-during.txt && test -s iostat-before.txt

tar -czf hostmon-evidence.tgz \
  sysstat-tools.txt uptime-before.txt free-before.txt \
  vmstat-before.txt iostat-before.txt sar-u-before.txt sar-d-before.txt \
  vmstat-during.txt iostat-during.txt \
  vmstat-after.txt iostat-after.txt sysstat-log-dir.txt
ls -l hostmon-evidence.tgz | tee evidence-ls.txt
```

Expected output: evidence archive exists; during/after samples recorded.

Validation steps

- [ ] `vmstat` and `iostat` produced before and during files
- [ ] You can point to idle (`id`) / wait (`wa`) columns in `vmstat`
- [ ] `sar -u` produced a sample (even if mostly idle)
- [ ] `hostmon-evidence.tgz` exists under `~/rebash-linux/lab19`

Common errors and fixes

Error	Cause	Fix
`iostat: command not found`	sysstat missing	`sudo apt-get install -y sysstat`
`sar` shows little history	Collector disabled	Enable sysstat cron/timer; still use live `sar -u 1 3`
Load made laptop unusable	Loop too aggressive	Shorten `timeout`; skip CPU loop on tiny VMs
Permission denied on `/var/log/sysstat`	Restricted perms	Use `sudo ls` or rely on live samples

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `~/rebash-linux/lab19/quick-host-check.sh` that prints timestamp, `uptime`, one `vmstat 1 3` block, and one `iostat -xz 1 2` block to `quick-host-check.out`, exiting `0` if both tools succeed.

Learning outcomes

- Captured baseline and under-load host signals
- Used vmstat/iostat/sar from sysstat
- Separated CPU vs disk pressure clues
- Saved monitoring evidence for a ticket

Cleanup

```
cd ~/rebash-linux/lab19
set -euo pipefail
rm -f load.bin cpu-load.pid dd-load.pid
# Keep hostmon-evidence.tgz if you want it
# sysstat package may remain installed — that is fine
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab19/` with evidence files
- [ ] You can explain what high `wa` suggests
- [ ] You know `sar` history needs the sysstat collector
- [ ] You would not run stress tests on shared production without approval

Code Walkthrough

Incident host check:

1. `uptime`, `free -h`, `df -hT`
2. `vmstat 1 10` — CPU/memory/swap
3. `iostat -xz 1 5` — disk saturation
4. `ps` / `pidstat` for top consumers
5. Compare with cloud metrics; then change capacity or fix the app

Security Considerations

- Monitoring data can reveal process names and paths — control access
- Do not run destructive stress tools on multi-tenant production
- Protect historical `sar` logs if they include sensitive context
- Prefer least privilege for monitoring agents
- Keep accurate time (chrony) so samples align with other systems

Common Mistakes

Resizing CPU when `wa` is high

The bottleneck may be disk. **Fix:** confirm with `iostat` before changing instance size.

One 1-second sample as truth

Spikes mislead. **Fix:** sample over several seconds or minutes.

Ignoring steal time on VMs

Noisy neighbours reduce your CPU. **Fix:** watch `st` in `vmstat` / `sar`; consider new host/capacity.

Expecting weeks of `sar` history by default

Collector may be off. **Fix:** enable sysstat's timer/cron or rely on a metrics stack.

Best Practices

- Keep a small "first 5 commands" host checklist
- Store baselines for critical VMs
- Alert on saturation (CPU, memory, disk util/latency)
- Correlate with app latency and error rates
- Practise reading vmstat/iostat offline from saved files

Troubleshooting

Symptom	Likely cause	Fix
High load, CPU idle	Uninterruptible I/O / DNS / locks	Check `wa`, `iostat`, app waits
High swap activity	RAM too small / leak	Inspect RSS; add RAM; fix app
Disk `%util` 100%	Storage bottleneck	Faster disk; reduce I/O; cache
Tools missing	Minimal image	Install `sysstat`
Metrics disagree with cloud	Agent lag / wrong host	Confirm instance ID; compare timestamps

Summary

`vmstat`, `iostat`, and `sar` turn "it feels slow" into CPU, memory, or disk evidence. Sample over time, compare before and during load, and fix the real bottleneck. Next: SSH Hardening and Firewalls.

Interview Questions

INTERVIEW PRACTICE

1. What does a high `wa` value in `vmstat` suggest?

Answer

High **I/O wait** means CPUs are idle waiting on block I/O (disk or sometimes network filesystems). Next check `iostat -xz` for saturated devices and review what process is doing heavy reads/writes.

2. How do you use `iostat` to see if a disk is saturated?

Answer

Run `iostat -xz 1 5` and look at `%util`, **await**/latency fields, and throughput. Near-100% util with rising await often means the device is the bottleneck. Confirm which mount sits on that device with `lsblk` / `findmnt`.

3. What is the difference between live `sar -u 1 3` and historical sar reports?

Answer

`sar -u 1 3` samples **live** counters now. Historical reports need the sysstat collector writing under `/var/log/sysstat`. If history is empty, enable the collector or use your metrics platform.

4. Which `vmstat` fields help diagnose memory pressure?

Answer

Watch swap-in/swap-out (`si` / `so`), free memory, and overall CPU. Pair with `free -h` and process RSS from `ps`. Sustained swapping needs RAM or a leak fix, not only "tune swapiness" as a first answer.

5. Why capture a baseline before changing instance size?

Answer

Without before/after numbers you cannot prove the resize helped. Baselines also show whether the problem was CPU, memory, or disk so you pick the right resize dimension.

6. How does steal time (`st`) affect your reading on a cloud VM?

Answer

Steal means the hypervisor scheduled other work instead of your VM. Your app can look slow even when your process list is modest. Consider moving workload or increasing capacity; do not blame the app alone.

7. Where do these tools fit next to Prometheus or CloudWatch?

Answer

Metrics systems are for continuous monitoring and alerting. `vmstat` / `iostat` / `sar` are for **interactive diagnosis**, interviews, and times when agents are broken. Good engineers use both.

SSH Hardening and Firewalls
MODULE 13 · SECURITY

Chapter 20. SSH Hardening and Firewalls

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/ssh-hardening-and-firewalls/

Overview

Most internet-facing Linux breaches start with weak Secure Shell (SSH) or open management ports. **SSH hardening** means tightening `sshd` so only the right people can log in, usually with public keys. A **host firewall** (Uncomplicated Firewall (UFW) on Ubuntu, or **firewalld** on RHEL-family systems) blocks unexpected inbound traffic. Together with a cloud security group, these layers protect jump servers (bastions) and application virtual machines (VMs).

A hardened host is not "passwords off and hope". You must **test config before reload**, keep a second session or serial console open, and **allow SSH in the firewall before you enable default-deny**. One bad `sshd` change or one UFW rule that forgets port 22 can lock you out of a cloud VM. This tutorial teaches safe hardening: keys, drop-in config, `sshd -t`, careful reload, and firewall rules that never cut your own access.

In production, teams also align host firewalls with cloud Network Security Groups (NSGs) / security groups, restrict source Internet Protocol (IP) ranges to office or bastion networks, and monitor auth failures. Fail2ban (next tutorial) can slow password guessing, but it does not fix a wide-open SSH port or a broken `sshd_config`.

This is **Tutorial 20** in **Module 13: Linux Security** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a practice hardening pack you can explain in an interview or change ticket — without locking yourself out.

Prerequisites

- Host Monitoring with vmstat, iostat, and sar
- SSH and Remote Access (keys and basic `ssh` usage)
- A **practice Ubuntu 22.04/24.04 VM** where you already have sudo and SSH access
- Prefer a **second session** open (or cloud serial console) while you change `sshd`
- Do **not** run this lab on a shared production jump server

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain how SSH keys, `sshd` drop-ins, and host firewalls work together
- [ ] Install an authorised key and prove login readiness without disabling password auth yet
- [ ] Add a safe `sshd` drop-in, validate with `sshd -t`, and reload without lockout
- [ ] Allow OpenSSH in UFW (or check firewalld) before any default-deny enable
- [ ] Capture evidence that port 22 still listens after the change

Architecture

SSH hardening sits between the client and the host kernel. Keys authenticate the user. `sshd` enforces policy. The host firewall and cloud security group filter which networks can even reach the SSH port.

SSH Hardening and Firewalls

Figure 20.1: SSH Hardening and Firewalls

Theory

What it is

SSH is the encrypted remote login service. The daemon is `sshd`. Client keys live under `~/.ssh/`; authorised public keys for a user live in `~/.ssh/authorized_keys`. Server settings live in `/etc/ssh/sshd_config` and drop-in files under `/etc/ssh/sshd_config.d/` (Ubuntu).

A **host firewall** decides which local ports accept connections. On Ubuntu, **UFW** is a simple front end to `nftables` / `iptables`. On RHEL-like systems, **firewalld** is common. Cloud security groups are a *separate* layer in front of the VM — both must allow SSH for remote access to work.

```
ss -lntp | grep -E ':22\b|sshd' || true
sudo sshd -T | grep -Ei 'passwordauthentication|permitrootlogin|pubkeyauthentication|maxauthtries'
sudo ufw status verbose 2>/dev/null || sudo firewall-cmd --list-all 2>/dev/null || true
```

Why it matters

SSH is the highest-value remote entry point on most Linux servers. Password guessing, reused passwords, and open management ports drive automated compromise. A hardened `sshd` plus a small firewall reduce attack surface even if someone edits a cloud security group by mistake. Lockout is also a real production risk: a bad config at 02:00 can mean an emergency serial-console recovery.

How it works

1. **Keys** — generate with `ssh-keygen`, install the public key into `authorized_keys`, keep the private key private.
2. **Config** — prefer a drop-in under `sshd_config.d/` instead of rewriting the whole main file.
3. **Test** — always run `sudo sshd -t` (or `sshd -t`) before reload. Fix errors first.
4. **Reload** — `systemctl reload ssh` (Ubuntu) or `systemctl reload sshd` while a second session stays open.
5. **Firewall** — allow OpenSSH / port 22 **before** enabling default-deny. Align with the cloud security group.

Safe hardening knobs for a first change (this lab uses these):

Setting	Safer first step
`MaxAuthTries`	Lower (for example 4) to slow guessing
`ClientAliveInterval` / `ClientAliveCountMax`	Drop idle dead sessions
`LoginGraceTime`	Limit unauthenticated connection time
`X11Forwarding`	`no` on servers that do not need GUI forward
`PasswordAuthentication`	Turn off **only after** key login is proven
`PermitRootLogin`	Prefer `no` or `prohibit-password` when you have a sudo user

Key concepts and comparisons

Control	Example	Role
Auth method	`PubkeyAuthentication yes`	How you prove identity
Identity scope	`AllowUsers deploy`	Who may log in at all
Host firewall	UFW / firewalld allow SSH	Local port filter
Cloud SG / NSG	Source CIDR = bastion/VPN	Network edge filter
Rate limit	fail2ban / equivalent	React to abuse (next tutorial)

Stack	Tool
Ubuntu / Debian	UFW (`ufw allow OpenSSH`)
RHEL family	firewalld (`firewall-cmd`)
Underlying	nftables / iptables

Common pitfalls

- Reloading `sshd` after a bad config with only one session — **lockout**.

- Enabling UFW without allowing OpenSSH first — **lockout**.
- Turning off `PasswordAuthentication` before your key works — **lockout**.
- Opening SSH to `0.0.0.0/0` and calling the host "hardened" because passwords are off.
- Host firewall and cloud security group disagreeing until nobody knows the true policy.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, install a lab SSH key, apply a **safe** `sshd` drop-in (no password disable, no port change), validate with `sshd -t`, reload carefully, and ensure the firewall path cannot cut SSH. Save proof under `~/rebash-linux/lab20`.

Prerequisites

- Ubuntu 22.04/24.04 (or Debian) with an admin user that already has sudo and SSH
- Packages: `openssh-server`, `ufw` (usual on Ubuntu)
- Keep **this SSH session** open until validation passes; open a second session if you can
- Take a VM snapshot before you start, if your hypervisor supports it

Lab environment

Workspace: `~/rebash-linux/lab20`

```
mkdir -p ~/rebash-linux/lab20 && cd ~/rebash-linux/lab20
set -euo pipefail
whoami | tee admin-user.txt
id | tee admin-id.txt
test -n "$(command -v sudo)"
sudo -n true 2>/dev/null || sudo -v
# Prove SSH is listening before we change anything
ss -lntp | tee ss-before.txt
grep -E ':22\b|sshd' ss-before.txt
```

Expected output: `admin-user.txt` and `ss-before.txt` exist; something is listening on port 22 (or your SSH port).

Lockout rules for this lab

Do **not** set `PasswordAuthentication no` until you have proven key login in a second session.
Do **not** change `Port` or `ListenAddress`.
Do **not** run `ufw enable` until `ufw status` shows OpenSSH (or port 22) allowed.
Prefer `systemctl reload` over `restart` for `sshd`.

Real-world scenario

Security asks you to harden a new Ubuntu app VM before it goes on the public internet. They want: (1) a documented key for the deploy user path, (2) safer `sshd` defaults that still allow your current login method, and (3) a host firewall that allows SSH before default-deny. You must leave proof for the change ticket and keep a recovery path (second session / serial console).

Step-by-step tasks

Task 1 – Lab key and current sshd effective settings

Create a key used only for this lab, and capture what `sshd` is currently enforcing.

```
cd ~/rebash-linux/lab20
set -euo pipefail

# Lab-only key (do not overwrite your real id_ed25519)
KEY="$HOME/rebash-linux/lab20/rebash_lab_ed25519"
if [[ ! -f "$KEY" ]]; then
  ssh-keygen -t ed25519 -N '' -f "$KEY" -C "rebash-lab20@$(hostname)"
fi

mkdir -p "$HOME/.ssh"
chmod 700 "$HOME/.ssh"
touch "$HOME/.ssh/authorized_keys"
chmod 600 "$HOME/.ssh/authorized_keys"
grep -Fxf "${KEY}.pub" "$HOME/.ssh/authorized_keys" >/dev/null 2>&1 \
```

```
  || cat "${KEY}.pub" >> "$HOME/.ssh/authorized_keys"

# Effective runtime settings (read-only view)
sudo sshd -T | grep -Ei \
  '^(passwordauthentication|pubkeyauthentication|permitrootlogin|maxauthtries|x11forwarding|logingracetime|clientaliveinterval|clientalivecountmax) ' \
  | tee sshd-T-before.txt

ls -l "$KEY" "${KEY}.pub" | tee key-ls.txt
```

Expected output: `rebash_lab_ed25519` and `.pub` exist; `sshd-T-before.txt` lists effective settings; public key is in `authorized_keys`.

Task 2 – Safe sshd drop-in, test, reload

Add only safe knobs. Do **not** disable passwords in this task.

```
cd ~/rebash-linux/lab20
set -euo pipefail

# Keep a second SSH session open if you can (or serial console)
DROP_IN="/etc/ssh/sshd_config.d/99-rebash-lab20.conf"
TMP="$(mktemp)"
cat > "$TMP" << 'EOF'
# REBASH lab20 — safe hardening (does not disable PasswordAuthentication)
MaxAuthTries 4
LoginGraceTime 60
ClientAliveInterval 300
ClientAliveCountMax 2
X11Forwarding no
EOF

# Install drop-in and validate syntax BEFORE reload
sudo install -m 0644 "$TMP" "$DROP_IN"
rm -f "$TMP"
sudo sshd -t 2>&1 | tee sshd-t.txt
# sshd -t prints nothing on success; confirm exit was 0
test ! -s sshd-t.txt || ! grep -Ei 'error|fatal' sshd-t.txt

# Reload (keeps existing sessions); service name is "ssh" on Ubuntu
sudo systemctl reload ssh 2>/dev/null || sudo systemctl reload sshd

sudo sshd -T | grep -Ei \
  '^(maxauthtries|logingracetime|clientaliveinterval|clientalivecountmax|x11forwarding) ' \
  | tee sshd-T-after.txt

grep -E 'maxauthtries 4|logingracetime 60|x11forwarding no' sshd-T-after.txt
ss -lntp | tee ss-after-reload.txt
grep -E ':22\b|sshd' ss-after-reload.txt
```

Expected output: `sshd -t` succeeds; after reload, effective settings show `maxauthtries 4` and `x11forwarding no`; SSH still listens.

Task 3 – Firewall: allow SSH first, then enable only if safe

This task never enables UFW unless OpenSSH is already allowed.

```
cd ~/rebash-linux/lab20
set -euo pipefail

if command -v ufw >/dev/null 2>&1; then
  sudo ufw status verbose 2>&1 | tee ufw-before.txt

  # Always allow OpenSSH before any enable
  sudo ufw allow OpenSSH comment 'REBASH lab20' 2>&1 | tee ufw-allow.txt || \
    sudo ufw allow 22/tcp comment 'REBASH lab20' 2>&1 | tee ufw-allow.txt

  sudo ufw status verbose 2>&1 | tee ufw-after-allow.txt
  grep -Ei 'OpenSSH|22/tcp' ufw-after-allow.txt

  # Enable only when inactive AND SSH allow is present
  if grep -qi 'Status: inactive' ufw-after-allow.txt; then
    if grep -Ei 'OpenSSH|22/tcp' ufw-after-allow.txt | grep -qi 'ALLOW'; then
      echo 'y' | sudo ufw enable 2>&1 | tee ufw-enable.txt
    else
      echo 'SKIP enable: SSH allow rule not visible' | tee ufw-enable.txt
      exit 1
    fi
  else
    echo 'UFW already active — not forcing re-enable' | tee ufw-enable.txt
  fi

  sudo ufw status verbose 2>&1 | tee ufw-final.txt
else
  # firewalld fallback (RHEL-like practice VMs)
  sudo firewall-cmd --state 2>&1 | tee firewalld-state.txt || true
```

```
  sudo firewall-cmd --list-all 2>&1 | tee firewalld-list.txt || true
  echo 'No ufw — captured firewalld (or none)' | tee ufw-final.txt
fi

# Final proof: listener still up
ss -lntp | tee ss-final.txt
grep -E ':22\b|sshd' ss-final.txt

# Pack whatever evidence files exist (UFW vs firewalld hosts differ)
shopt -s nullglob
EVIDENCE=(
  admin-user.txt admin-id.txt
  ss-before.txt ss-after-reload.txt ss-final.txt
  sshd-T-before.txt sshd-T-after.txt sshd-t.txt
  key-ls.txt ufw-final.txt
  ufw-*.txt firewalld-*.txt
)
tar -czf ssh-firewall-evidence.tgz "${EVIDENCE[@]}"
shopt -u nullglob

ls -l ssh-firewall-evidence.tgz | tee evidence-ls.txt
```

Expected output: OpenSSH/22 allow is listed (on UFW hosts); SSH still listens; `ssh-firewall-evidence.tgz` is not empty.

Validation steps

- [ ] Lab key exists under `~/rebash-linux/lab20/` and public key is in `~/.ssh/authorized_keys`
- [ ] `/etc/ssh/sshd_config.d/99-rebash-lab20.conf` exists
- [ ] `sshd -t` succeeded before reload
- [ ] `sshd -T` shows `maxauthtries 4` after reload
- [ ] `ss` still shows SSH listening on port 22 (or your SSH port)
- [ ] UFW (if present) lists OpenSSH or 22/tcp ALLOW before/when enabled
- [ ] `ssh-firewall-evidence.tgz` exists

Common errors and fixes

Error	Cause	Fix
`sshd: no hostkeys available`	Broken OpenSSH install	Reinstall `openssh-server`; do not reload until fixed
`Reload failed` / connection drop	Bad config or wrong service name	Use serial console; `sshd -t`; fix/remove drop-in; `systemctl reload ssh`
Locked out after UFW enable	SSH not allowed first	Serial/console: `ufw allow OpenSSH` then `ufw reload`
`Permission denied (publickey)` later	You disabled passwords before testing keys	Keep a console; re-enable passwords temporarily; fix `authorized_keys` modes (`700` / `600`)
`ufw: command not found`	Minimal / RHEL image	Use firewalld commands in Task 3 fallback

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write an executable script `~/rebash-linux/lab20/check-ssh-safe.sh` that exits **0** only if: (1) `sshd -t` is clean, (2) `ss` shows a listener on port 22, and (3) either UFW is inactive **or** OpenSSH/22 is ALLOW. Run it and save stdout to `check-ssh-safe.out`. Do not disable password auth in the challenge.

Learning outcomes

- Installed a lab SSH key without breaking existing login
- Applied a safe `sshd` drop-in with `sshd -t` before reload
- Allowed SSH in the firewall before enabling default-deny
- Captured evidence suitable for a change ticket

Cleanup

```
cd ~/rebash-linux/lab20
set -euo pipefail
```

```
# Remove lab sshd drop-in and reload
sudo rm -f /etc/ssh/sshd_config.d/99-rebash-lab20.conf
sudo sshd -t
sudo systemctl reload ssh 2>/dev/null || sudo systemctl reload sshd || true

# Optional: remove only the lab public key line from authorized_keys
# PUB=$(cat ~/rebash-linux/lab20/rebash_lab_ed25519.pub)
# grep -Fvx "$PUB" ~/.ssh/authorized_keys > ~/.ssh/authorized_keys.tmp && mv ~/.ssh/authorized_keys.tmp ~/.ssh/authorized_keys

# UFW: leave rules if you need them; to remove the lab comment rule manually review `ufw status numbered`
# Keep evidence archive if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab20/` with evidence files
- [ ] You can explain why `sshd -t` and a second session matter
- [ ] You can explain the order: allow SSH → then enable firewall
- [ ] You know when it is safe to turn off `PasswordAuthentication`

Code Walkthrough

In real servers, SSH and firewall hardening usually follows this order:

1. **Prove access** — second session or serial console open
2. **Keys first** — install and test key login before disabling passwords
3. **Small drop-ins** — files under `sshd_config.d/` with clear comments
4. **Validate then reload** — `sshd -t`, then `systemctl reload`
5. **Firewall last** — allow OpenSSH, then enable; align with cloud security groups

Configuration management (Ansible, cloud-init) should own these files. People still keep a break-glass console path.

Security Considerations

- Never paste private keys into tickets, chat, or git repositories
- Prefer key-based SSH; disable password auth only after key proof
- Restrict SSH source CIDRs in the cloud security group, not only on the host
- Keep `PermitRootLogin no` (or `prohibit-password`) on internet-facing hosts
- Review auth logs (`journalctl -u ssh` / `/var/log/auth.log`) after changes

Common Mistakes

Enabling UFW before allowing OpenSSH

Default deny drops your SSH session. **Fix:** `ufw allow OpenSSH` first, confirm with `ufw status`, then enable. Keep serial console ready.

Setting PasswordAuthentication no with only one session

A bad key path locks you out. **Fix:** prove key login in a second session, then change auth, then reload.

Editing the whole sshd_config by hand

Package upgrades and typos are harder to review. **Fix:** use a drop-in under `sshd_config.d/` and always run `sshd -t`.

Calling the host hardened while SSH is open to the world

Keys help, but scanners still find the port. **Fix:** tighten cloud security group source ranges to bastion/VPN/office.

Best Practices

- Manage `sshd` and firewall rules with configuration management
- Use short-lived certificates (for example SSH CA) where your platform supports them
- Prefer bastion / jump host patterns over exposing every VM on port 22
- Document recovery: serial console, snapshot, break-glass user

- Test hardening on a practice VM before golden-image bake

Troubleshooting

Symptom	Likely cause	Fix
Connection refused after reload	`sshd` failed to start	Serial console; `sshd -t`; `journalctl -u ssh`
Timeout after UFW enable	Port 22 not allowed	Console: allow OpenSSH; check cloud SG too
`Permission denied (publickey)`	Wrong key / modes / user	Check `authorized_keys`, `chmod 700 ~/.ssh`, `600 authorized_keys`
Settings not applied	Drop-in name/order or typo	`sshd -T`; confirm file under `sshd_config.d/`
Works from office, not CI	Security group / firewall source mismatch	Align SG and host firewall CIDRs

Summary

SSH hardening and host firewalls reduce remote attack surface, but **order and validation prevent lockout**. Install keys, apply small tested drop-ins, reload carefully, and allow SSH before default-deny. Next, learn Mandatory Access Control (MAC), Fail2Ban, auditd, and Pluggable Authentication Modules (PAM) in SELinux, AppArmor, Fail2Ban, Auditd, and PAM.

Interview Questions

INTERVIEW PRACTICE

1. What is the safe order of operations when hardening SSH and enabling a host firewall on a new cloud VM?

Answer

Keep a second session or serial console. Install and prove SSH key login. Apply a small `sshd` drop-in and run `sshd -t` before `systemctl reload`. Allow OpenSSH (or port 22) in UFW/firewalld **before** enabling default-deny. Align the cloud security group. Only then consider disabling password authentication. Interviewers want lockout avoidance, not just a list of settings.

2. Why prefer a drop-in under `/etc/ssh/sshd_config.d/` over rewriting `/etc/ssh/sshd_config`?

Answer

Drop-ins are easier to review, own in configuration management, and remove in an emergency. The main file may be replaced on package upgrade depending on packaging. A named lab/production drop-in makes intent clear and reduces merge mistakes.

3. You set `PasswordAuthentication no` and lost access. How do you recover?

Answer

Use the cloud **serial console**, hypervisor console, or rescue mode. Fix or remove the bad drop-in, restore a working `authorized_keys`, run `sshd -t`, and reload. Temporarily re-enable passwords only if needed, then re-test keys. Prevent by proving key login first and keeping a break-glass path.

4. How do host UFW/firewalld rules relate to cloud security groups?

Answer

They are **two layers**. The security group filters traffic before it reaches the VM Network Interface Card (NIC). The host firewall filters on the guest. Both must allow SSH from the intended source. Hardening only one layer while leaving the other open (or blocked) causes false confidence or mysterious timeouts.

5. Which `sshd` settings would you change first on a practice VM, and which would you delay?

Answer

First: `MaxAuthTries`, idle keepalive, `LoginGraceTime`, `X11Forwarding no`, confirm `PubkeyAuthentication yes`. Delay: `PasswordAuthentication no`, port changes, and aggressive `AllowUsers` until accounts and keys are proven. Always `sshd -t` and keep a second session.

6. How would you prove a hardening change is safe for a change ticket?

Answer

Attach: `sshd -t` success, `sshd -T` before/after for the changed keys, `ss` showing SSH still listening, firewall status showing OpenSSH allowed, and a successful new login test (preferably key-based). Least risk is shown by what you *did not* break.

7. Why is "keys only, SSH open to 0.0.0.0/0" still weak for a public jump server?

Answer

Key-only auth stops password guessing, but the service is still exposed to scanning, user enumeration noise, zero-days, and stolen keys. Prefer bastion patterns, source IP restriction in the security group, monitoring of auth failures, and short-lived credentials where possible.

SELinux, AppArmor, Fail2Ban, Auditd, and PAM
MODULE 13 · SECURITY

Chapter 21. SELinux, AppArmor, Fail2Ban, Auditd, and PAM

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/selinux-apparmor-fail2ban-auditd-pam/

Overview

File modes and sudo are **Discretionary Access Control (DAC)** — the owner decides. Production hosts also need **Mandatory Access Control (MAC)**, authentication policy, and an audit trail. On Ubuntu, **AppArmor** is the usual MAC. On RHEL-family systems, **SELinux** is common. **Fail2Ban** bans addresses after repeated login failures. **auditd** records security-relevant events. **Pluggable Authentication Modules (PAM)** define how login, sudo, and SSH authentication behave.

These tools look mysterious when something fails after a correct `chmod`. A MAC denial, a PAM typo, or a Fail2Ban jail that never matches your logs can waste hours. The safe engineering habit is: **detect what is installed, read status, collect evidence** — do not disable MAC "to make it work" on day one.

In regulated or enterprise environments, auditors ask whether MAC is enforcing, whether auth failures are logged, and whether PAM changes are controlled. Even on a small cloud virtual machine (VM), knowing which stack you have prevents copying RHEL SELinux advice onto an Ubuntu host (or the reverse).

This is **Tutorial 21** in **Module 13: Linux Security** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers. By the end, you will have a **security posture report** built from safe status commands — no MAC disable, no PAM rewrite.

Prerequisites

- SSH Hardening and Firewalls
- A **practice Ubuntu 22.04/24.04 VM** (AppArmor path) or RHEL-like VM (SELinux path) with sudo
- Do **not** edit PAM or set SELinux to Disabled on a shared production host for this lab

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Detect whether SELinux or AppArmor is present and report its mode/status
- [ ] Check auditd and Fail2Ban safely (installed vs active vs missing)
- [ ] Read a PAM stack file without editing it
- [ ] Explain how MAC, PAM, audit, and Fail2Ban layer on a host
- [ ] Produce a posture evidence pack under `~/rebash-linux/lab21`

Architecture

Security layers sit around login and process execution. PAM authenticates. MAC confines what a process may do after it starts. auditd records. Fail2Ban reacts to repeated abuse in logs.

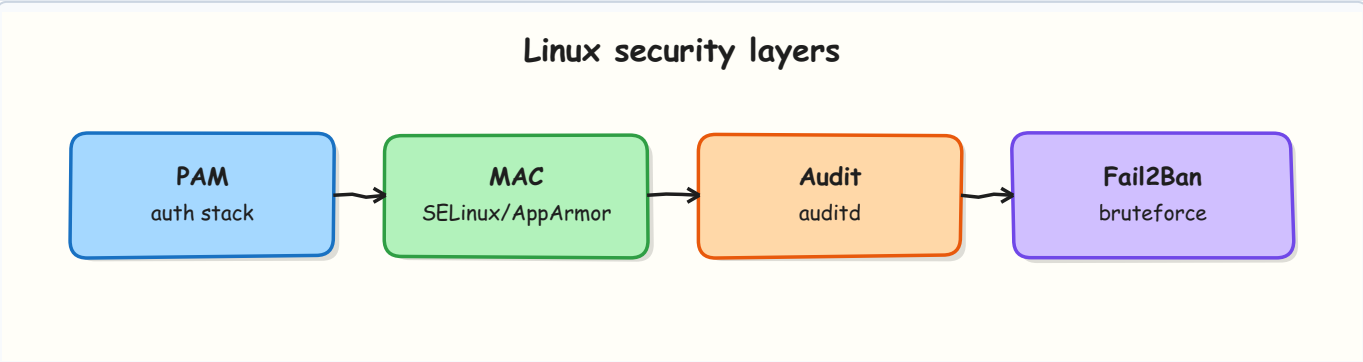

Figure 21.1: SELinux, AppArmor, Fail2Ban, Auditd, and PAM

Theory

What it is

SELinux and **AppArmor** are MAC systems. They confine processes beyond `chmod` / `chown`. SELinux uses labels (`ls -Z`, `getenforce`). AppArmor uses path-based profiles (`aa-status`, `aa-enabled`).

Fail2Ban watches log lines (for example SSH failures) and adds temporary firewall bans for source Internet Protocol (IP) addresses.

auditd is the Linux audit daemon. Rules decide what to record; `ausearch` / `aureport` help you query.

PAM stacks under `/etc/pam.d/` chain modules for `auth`, `account`, `password`, and `session`. SSH, sudo, and login all use PAM.

```
command -v getenforce >/dev/null && getenforce || echo 'no SELinux getenforce'
command -v aa-status >/dev/null && sudo aa-status --enabled; sudo aa-status 2>/dev/null | head || echo 'no AppArmor tools'
systemctl is-active auditd 2>/dev/null || true
systemctl is-active fail2ban 2>/dev/null || true
```

Why it matters

MAC denials often look like random `Permission denied` after DAC looks correct. Disabling SELinux/AppArmor removes a major control — fix labels or profiles instead. Broken PAM can lock out every user. Fail2Ban that points at the wrong log path gives false comfort. Audit trails matter for incidents and compliance.

How it works

1. **Detect MAC** — SELinux: `getenforce` / `sestatus`. AppArmor: `aa-enabled`, `aa-status`.
2. **Modes** — SELinux: Enforcing, Permissive, Disabled. AppArmor profiles: enforce or complain.
3. **Audit** — `systemctl status auditd`; recent boots in `journalctl -u auditd` (or audit logs).
4. **Fail2Ban** — `fail2ban-client status` when installed; check jails carefully.
5. **PAM** — read `/etc/pam.d/sshd` or `common-auth` (Ubuntu); never test by disconnecting all sessions after a blind edit.

These layers work together: PAM decides if you may authenticate; MAC confines the process; audit records; Fail2Ban slows attackers who hammer SSH.

Key concepts and comparisons

Control	Primary job
SELinux / AppArmor	Confine processes (MAC)
DAC (`chmod` / ACL)	Owner-managed access
Fail2Ban	Reactive IP bans on abuse
auditd	Compliance / forensic trail
PAM	Auth stack behaviour

Distro tendency	MAC
RHEL, Rocky, Alma	SELinux
Ubuntu	AppArmor
Many containers	Often unconfined unless configured

Common pitfalls

- Setting SELinux to Disabled permanently instead of fixing contexts.
- Copying SELinux fix steps onto Ubuntu (or AppArmor steps onto RHEL) without detecting the stack.
- Editing PAM and testing only after all sessions disconnect — **lockout**.
- Fail2Ban jails that never match journald/syslog paths.
- Assuming guest MAC equals strong container isolation — runtime config matters.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice VM, **detect and report** MAC (AppArmor and/or SELinux), auditd, Fail2Ban, and one PAM stack — safely. Do not disable MAC. Do not edit PAM. Save a posture report under `~/rebash-linux/lab21`.

Prerequisites

- Ubuntu 22.04/24.04 (typical) or RHEL-like practice VM with sudo
- Optional packages if missing: `apparmor-utils` (Ubuntu), `fail2ban` (optional install only if you choose), `auditd`
- Root/serial console available if you later experiment beyond this lab

Lab environment

Workspace: `~/rebash-linux/lab21`

```
mkdir -p ~/rebash-linux/lab21 && cd ~/rebash-linux/lab21
set -euo pipefail
whoami | tee admin-user.txt
id | tee admin-id.txt
test -n "$(command -v sudo)"
sudo -n true 2>/dev/null || sudo -v
cat /etc/os-release | tee os-release.txt
```

Expected output: identity and OS files exist; sudo works.

Safe lab rules

Do **not** run `setenforce 0` as a "fix".
Do **not** set `SELINUX=disabled` in config.
Do **not** edit files under `/etc/pam.d/` in this lab.
Read status; collect evidence; leave controls as you found them (unless you optionally install Fail2Ban for status only).

Real-world scenario

A security questionnaire asks: "Is MAC enforcing? Is auditd running? Do you rate-limit SSH abuse? Where is PAM configured for SSH?" You must answer from a practice host with command evidence — without weakening the host to get screenshots.

Step-by-step tasks

Task 1 – Detect MAC (AppArmor and SELinux) safely

```
cd ~/rebash-linux/lab21
set -euo pipefail

{
  echo "=== SELinux ==="
  if command -v getenforce >/dev/null 2>&1; then
    getenforce | tee selinux-getenforce.txt
    command -v sestatus >/dev/null 2>&1 && sestatus | tee selinux-sestatus.txt || true
  else
    echo 'SELinux tools not present' | tee selinux-getenforce.txt
```

```
  fi

  echo "=== AppArmor ==="
  if command -v aa-enabled >/dev/null 2>&1; then
    aa-enabled 2>&1 | tee apparmor-enabled.txt || true
  else
    echo 'aa-enabled not present' | tee apparmor-enabled.txt
  fi
  if command -v aa-status >/dev/null 2>&1; then
    sudo aa-status 2>&1 | tee apparmor-status.txt
  else
    echo 'aa-status not present' | tee apparmor-status.txt
  fi

  # At least one MAC stack should be reported on a normal Ubuntu/RHEL server image
  if grep -Eqi 'Enforcing|Permissive|Disabled|Yes|enabled|apparmor' \
      selinux-getenforce.txt apparmor-enabled.txt apparmor-status.txt 2>/dev/null; then
    echo 'MAC detection captured' | tee mac-summary.txt
  else
    echo 'MAC tools missing — note for container/minimal images' | tee mac-summary.txt
  fi
} | tee mac-detect.log
```

Expected output: `mac-detect.log` and status files exist; on Ubuntu you typically see AppArmor profiles; on RHEL-like hosts you see `getenforce` mode.

Task 2 – auditd and Fail2Ban status (detect only)

```
cd ~/rebash-linux/lab21
set -euo pipefail

{
  echo "=== auditd ==="
  systemctl is-enabled auditd 2>&1 | tee auditd-enabled.txt || true
  systemctl is-active auditd 2>&1 | tee auditd-active.txt || true
  systemctl status auditd --no-pager 2>&1 | head -n 20 | tee auditd-status.txt || true
  if command -v ausearch >/dev/null 2>&1; then
    # Safe, small query — may be empty on quiet hosts
    sudo ausearch -m USER_LOGIN -ts recent 2>&1 | head -n 30 | tee ausearch-user-login.txt || \
      echo 'no recent USER_LOGIN matches' | tee ausearch-user-login.txt
  else
    echo 'ausearch not installed' | tee ausearch-user-login.txt
  fi

  echo "=== Fail2Ban ==="
  if command -v fail2ban-client >/dev/null 2>&1; then
    systemctl is-active fail2ban 2>&1 | tee fail2ban-active.txt || true
    sudo fail2ban-client status 2>&1 | tee fail2ban-status.txt || true
  else
    echo 'fail2ban not installed' | tee fail2ban-active.txt
    echo 'fail2ban not installed' | tee fail2ban-status.txt
  fi
} | tee audit-fail2ban.log

# Honest asserts: files exist (services may be inactive on minimal VMs)
test -s auditd-active.txt
test -s fail2ban-status.txt
```

Expected output: status files for auditd and Fail2Ban; either real status or a clear “not installed / inactive” note.

Task 3 – Read PAM (no edits) and build posture report

```
cd ~/rebash-linux/lab21
set -euo pipefail

# Ubuntu common files; fall back to sshd PAM stack
if [[ -f /etc/pam.d/common-auth ]]; then
  sudo sed -n '1,80p' /etc/pam.d/common-auth | tee pam-common-auth.txt
fi
if [[ -f /etc/pam.d/sshd ]]; then
  sudo sed -n '1,80p' /etc/pam.d/sshd | tee pam-sshd.txt
else
  echo 'no /etc/pam.d/sshd' | tee pam-sshd.txt
fi
ls -l /etc/pam.d | tee pam-d-ls.txt

# Machine-readable posture summary
{
  echo "host=$(hostname)"
  echo "user=$(whoami)"
  echo "date_utc=$(date -u +%Y-%m-%dT%H:%M:%SZ)"
  echo "selinux=$(cat selinux-getenforce.txt | tr '\n' ' ' | sed 's/ *$//')"
  echo "apparmor_enabled=$(head -n 1 apparmor-enabled.txt)"
  echo "auditd_active=$(cat auditd-active.txt)"
  echo "fail2ban=$(head -n 1 fail2ban-status.txt)"
  echo "pam_sshd_bytes=$(wc -c < pam-sshd.txt)"
```

```
} | tee posture.env

# Human report
{
  echo "# REBASH lab21 security posture"
  echo
  cat posture.env
  echo
  echo "## Notes"
  echo "- MAC: see mac-detect.log (do not disable to 'fix' apps)"
  echo "- PAM: read-only capture; edit only with console access and change control"
  echo "- Fail2Ban: install/configure only after SSH hardening and log path checks"
} | tee posture-report.md

tar -czf security-posture.tgz \
  admin-user.txt admin-id.txt os-release.txt \
  mac-detect.log mac-summary.txt \
  selinux-*.txt apparmor-*.txt \
  auditd-*.txt ausearch-user-login.txt audit-fail2ban.log \
  fail2ban-*.txt \
  pam-*.txt pam-d-ls.txt \
  posture.env posture-report.md

ls -l security-posture.tgz | tee evidence-ls.txt
test -s security-posture.tgz
```

Expected output: `pam-sshd.txt` (or clear missing note), `posture-report.md`, and non-empty `security-posture.tgz`.

Validation steps

- [ ] `mac-detect.log` shows SELinux and/or AppArmor detection results
- [ ] auditd active/enabled (or honest inactive) captured
- [ ] Fail2Ban status or "not installed" captured
- [ ] PAM files read without modification (`pam-sshd.txt` / `common-auth`)
- [ ] `security-posture.tgz` exists under `~/rebash-linux/lab21`

Common errors and fixes

Error	Cause	Fix
`aa-status: command not found`	Minimal image	`sudo apt-get update && sudo apt-get install -y apparmor-utils` on Ubuntu
`getenforce: command not found`	Not a SELinux distro	Normal on Ubuntu — report AppArmor instead
`Unit auditd not found`	Package not installed	Note as gap; install `auditd` only on practice VMs if required
Empty `ausearch`	Quiet host / no rules	Acceptable — record "no matches"
Tempted to `setenforce 0`	App "fix" culture	Check denials in audit/journal; fix labels/profiles

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `~/rebash-linux/lab21/posture-check.sh` that prints `PASS` or `FAIL` for each of: (1) AppArmor enabled **or** SELinux Enforcing/Permissive detected, (2) `pam-sshd.txt` or `pam-common-auth.txt` non-empty, (3) posture tarball exists. Exit non-zero if any required check fails on your distro. Run it and save output to `posture-check.out`.

Learning outcomes

- Detected the correct MAC stack for the host
- Checked auditd and Fail2Ban without weakening controls
- Read PAM safely
- Built a posture evidence pack for tickets or questionnaires

Cleanup

```
cd ~/rebash-linux/lab21
set -euo pipefail
```

```
# This lab is read-only for MAC/PAM — nothing to revert
# Keep security-posture.tgz for your notes if you want
# rm -f *.txt *.log *.md *.env *.tgz   # only if you want a clean workspace
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab21/` with evidence files
- [ ] You can say which MAC your distro uses and how to check its mode
- [ ] You can explain why disabling MAC is the wrong first fix
- [ ] You know PAM edits need console access and change control

Code Walkthrough

In real incidents and audits, host security work usually follows this order:

1. **Detect** — which MAC, is auditd up, is Fail2Ban installed
2. **Read** — status, recent denials, PAM includes — before changing
3. **Fix forward** — labels/profiles/jails, not "disable everything"
4. **Evidence** — commands and outputs in the ticket
5. **Least privilege** — change one control at a time with rollback

Configuration management should own MAC mode, audit rules, and Fail2Ban jails.

Security Considerations

- Never disable SELinux/AppArmor permanently to ship a feature
- Treat PAM edits as break-glass level changes
- Restrict who can read audit and auth logs
- Fail2Ban is not a substitute for key-only SSH and tight security groups
- Document emergency admin access when central auth (LDAP/IdM) is used

Common Mistakes

Disabling SELinux to fix a permission error

You remove a major control and hide the real label issue. **Fix:** check `ausearch` / audit denials, restore contexts (`restorecon`), or adjust policy properly.

Editing PAM over a single SSH session

A typo can lock out every login. **Fix:** keep a root console; change one line; test a new session before you disconnect.

Assuming Fail2Ban is protecting SSH

Wrong log path or inactive jail means no bans. **Fix:** `fail2ban-client status` and `status sshd`; confirm log source matches journald/syslog.

Applying the wrong MAC playbook for the distro

SELinux steps on Ubuntu (or the reverse) waste time. **Fix:** detect with `getenforce` / `aa-status` first.

Best Practices

- Leave MAC in enforcing mode on production images
- Manage audit rules and Fail2Ban jails as code
- Pair Fail2Ban with SSH hardening from Tutorial 20
- Review PAM changes in pull requests like sudoers changes
- Collect posture evidence during image bake, not only during audits

Troubleshooting

Symptom	Likely cause	Fix
App works after `chmod` still denied	MAC denial	AppArmor logs / SELinux AVC; fix profile or context
`sudo` or SSH rejects everyone	PAM mis-edit	Console recovery; restore PAM from package/backup
Fail2Ban never bans	Jail inactive / wrong log	`fail2ban-client status`; fix `backend` / `logpath`
auditd inactive after reboot	Not enabled	`systemctl enable --now auditd` on practice hosts
Container still "escapes" expectations	Weak runtime isolation	MAC + user namespaces + least privilege — not guest MAC alone

Summary

SELinux/AppArmor, Fail2Ban, auditd, and PAM are layered host controls. Learn to **detect and report** them safely before you change them. Do not disable MAC to pass a demo. Next, see how containers use kernel namespaces and cgroups in Containers — Namespaces, cgroups, OverlayFS, and OCI.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between DAC and MAC on Linux, with one example of each?

Answer

DAC (Discretionary Access Control) is owner-managed access such as `chmod` / `chown` and Access Control Lists (ACLs). **MAC** (Mandatory Access Control) is system policy that can deny access even when DAC allows it — SELinux labels or AppArmor profiles. Example: a web daemon may own its files (DAC) but still be blocked from reading `/etc/shadow` by MAC.

2. How do you check whether AppArmor or SELinux is active on an unknown VM?

Answer

Check both. SELinux: `getenforce` / `sestatus`. AppArmor: `aa-enabled` and `sudo aa-status`. Also read `/etc/os-release` so you know which distro family you are on. Do not assume Ubuntu has SELinux or that RHEL has AppArmor as the primary stack.

3. A junior engineer wants to set SELinux to Disabled to fix a deployment. What do you recommend instead?

Answer

Keep Enforcing (or use Permissive briefly only for diagnosis on non-production). Read AVC denials, fix file contexts (`restorecon`) or policy, and retest. Permanent Disabled removes a major control and often fails audits. Explain the business risk in plain language.

4. What does Fail2Ban actually do, and what does it not do?

Answer

Fail2Ban **reacts** to repeated matching log events by banning source IPs (usually via the firewall) for a time. It does **not** replace key-only SSH, security groups, patching, or MAC. Misconfigured jails can ban nobody — or ban legitimate networks — so status and log paths matter.

5. Why are PAM changes considered high risk?

Answer

PAM sits on the authentication path for SSH, sudo, and local login. One bad line can lock out all users. Changes need a console/break-glass plan, small diffs, and a test login before you close your working session. Treat PAM like sudoers: validate carefully.

6. How would you prove host security posture for an audit using only safe commands?

Answer

Capture OS release, MAC status (`getenforce` / `aa-status`), auditd active/enabled, Fail2Ban status or "not installed", and a read-only excerpt of `/etc/pam.d/sshd`. Pack outputs with timestamps. Do not disable controls to get a "clean" screenshot.

7. An application gets Permission denied but `ls -l` looks correct. What else do you check?

Answer

Check MAC: AppArmor denials in journal/logs, or SELinux `ausearch`/AVC messages and `ls -Z`. Also check ACLs (`getfacl`), mount options (`noexec`, `nosuid`), and whether the process user/group is what you think (`ps`, `id`). DAC success does not mean MAC allows the access.

Containers — Namespaces, cgroups, OverlayFS, and OCI
MODULE 14 · CONTAINERS & CLOUD

Chapter 22. Containers — Namespaces, cgroups, OverlayFS, and OCI

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/containers-namespaces-cgroups-and-oci/

Overview

Kubernetes nodes are Linux. A **container** is not a tiny virtual machine (VM). It is a normal process with an isolated view of the system (**namespaces**) and resource limits (**control groups / cgroups**), usually with a layered root filesystem such as **OverlayFS**. The **Open Container Initiative (OCI)** defines image and runtime standards so engines (Docker, containerd, CRI-O) can build and run portable images.

When a pod is Out-Of-Memory (OOM) killed, when a node disk fills with image layers, or when "it works in Docker but not in the cluster", the real story is often namespaces, cgroups, and filesystems on the host. Brand names sit on top of these kernel features.

In production, Site Reliability Engineering (SRE) and platform engineers debug both the YAML and the node: `lsns`, cgroup paths under `/sys/fs/cgroup`, `findmnt` for overlay mounts, and runtime versions (`runc` / `crun`). Understanding the kernel contract makes Docker and Kubernetes less magical and easier to operate.

This is **Tutorial 22** in **Module 14: Containers & Cloud** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, SREs, and platform engineers. By the end, you will have host-level evidence of namespaces, cgroups, and overlay/runtime facts you can explain in an interview.

Prerequisites

- SELinux, AppArmor, Fail2Ban, Auditd, and PAM
- A **practice Ubuntu 22.04/24.04 VM** with sudo
- Optional: Docker or Podman installed (lab works with kernel tools alone; runtime steps are optional extras)
- Do **not** run heavy container clean-ups on a shared production node

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain how namespaces and cgroups differ (what you see vs what you can consume)
- [ ] List namespaces with `lsns` and read your process cgroup path
- [ ] Detect OverlayFS mounts and relate them to image layers
- [ ] Identify OCI runtime pieces (`runc` / `crun` /engine) when present
- [ ] Capture a container-internals evidence pack under `~/rebash-linux/lab22`

Architecture

Container engines sit above an OCI runtime. The runtime asks the kernel for namespaces, cgroups, and a root filesystem (often OverlayFS). The application process runs with that isolated view and those limits.

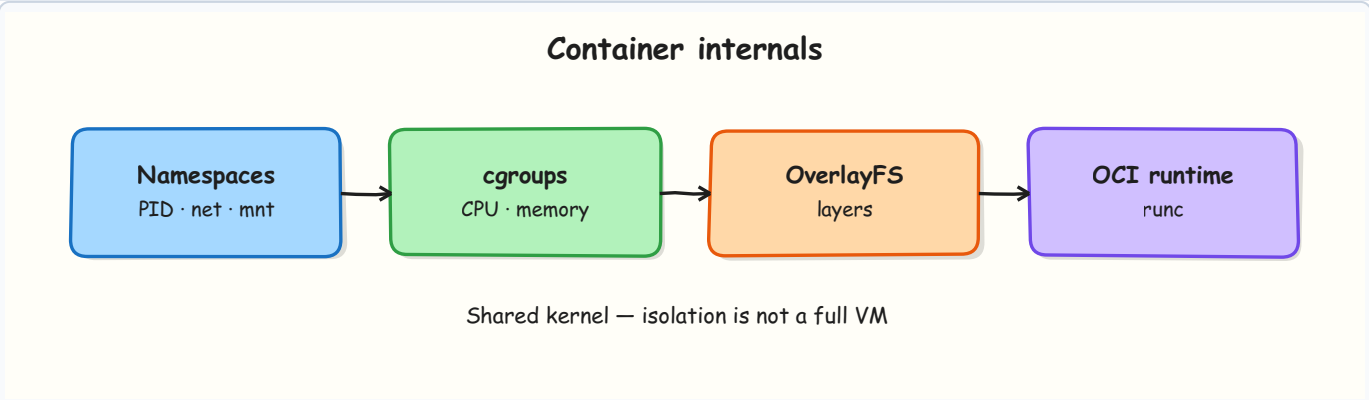

Figure 22.1: Containers — Namespaces, cgroups, OverlayFS, and OCI

Theory

What it is

Namespaces isolate what a process can *see*: process IDs (PID), network, mounts, hostname (UTS), Inter-Process Communication (IPC), user IDs, cgroup view, and time.

cgroups limit and account what a process can *consume*: CPU, memory, I/O, PIDs. Modern systems use **cgroup v2** under `/sys/fs/cgroup`.

OverlayFS stacks read-only image layers under a writable upper layer (copy-on-write).

OCI defines the image format and the runtime lifecycle (`create`, `start`, …). Engines add UX, networking plugins, and distribution.

```
lsns
cat /proc/self/cgroup
findmnt -t overlay 2>/dev/null | head || true
command -v runc; command -v crun; command -v docker; command -v podman
```

Why it matters

Node disk fill, noisy-neighbour CPU, and mysterious OOM kills are kernel-feature issues underneath the container brand. Ops roles debug the host as much as the manifest. Interviewers expect you to say "namespaces isolate; cgroups limit" without confusing them with VMs.

How it works

1. **Engine** pulls/builds an OCI image (layers + config).
2. **Runtime** (`runc` / `crun`) creates namespaces and joins/creates cgroups.
3. **Rootfs** is assembled (often OverlayFS) and the entrypoint starts as PID 1 *inside* the PID namespace.
4. **Host tools** still see the process: `ps`, `lsns`, `systemd-cgls`, `/sys/fs/cgroup`.

You can also explore namespaces without Docker using `unshare` / `nsenter` (careful: practice VM only).

Key concepts and comparisons

Mechanism	Isolates / limits
Namespaces	What the process can *see*
cgroups	What the process can *consume*
OverlayFS	Layered filesystem view
OCI runtime	Standard create/start lifecycle

Stack piece	Role
Engine (Docker / containerd)	Images, API, UX
OCI runtime (`runc` / `crun`)	Kernel plumbing
Kernel	Namespaces, cgroups, OverlayFS

Common pitfalls

- Treating containers as strong security boundaries without MAC, user namespaces, and least privilege.
- Ignoring cgroup OOM kills while chasing application “random exits”.
- Filling the node with image layers and container logs under `/var/lib`.
- Debugging only inside the container when `lsns` / cgroup paths on the host show the truth.
- Assuming PID 1 behaviour inside containers matches a full systemd operating system.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

On a practice Ubuntu VM, inspect namespaces and cgroups, create a short-lived user namespace demo with `unshare`, check for OverlayFS / runtime tools, and save evidence under `~/rebash-linux/lab22`.

Prerequisites

- Ubuntu 22.04/24.04 with `util-linux` (`lsns`, `unshare`, `findmnt`)
- sudo for some cgroup listings
- Optional: Docker or Podman for Task 3 extras

Lab environment

Workspace: `~/rebash-linux/lab22`

```
mkdir -p ~/rebash-linux/lab22 && cd ~/rebash-linux/lab22
set -euo pipefail
whoami | tee admin-user.txt
uname -r | tee kernel.txt
cat /proc/self/cgroup | tee self-cgroup-initial.txt
test -n "$(command -v lsns)"
test -n "$(command -v unshare)"
```

Expected output: kernel and cgroup files exist; `lsns` and `unshare` are available.

Real-world scenario

A Kubernetes node shows high disk use and occasional OOM kills. Before you blame the app YAML, you need host-level proof: which namespaces exist, how cgroups are mounted, whether overlay mounts are piling up, and which OCI runtime the engine uses. You practise that inspection path on a lab VM.

Step-by-step tasks

Task 1 – Namespaces inventory and a safe unshare demo

```
cd ~/rebash-linux/lab22
set -euo pipefail

lsns | tee lsns.txt
lsns -t pid,net,mnt,uts,user 2>/dev/null | tee lsns-filtered.txt || lsns | tee lsns-filtered.txt

# Short-lived UTS namespace: hostname change must NOT affect the host
HOST_BEFORE="$(hostname)"
unshare --uts /bin/bash -c 'hostname rebash-lab22-ns; hostname' | tee unshare-uts.txt
HOST_AFTER="$(hostname)"
echo "host_before=$HOST_BEFORE" | tee hostname-check.txt
echo "host_after=$HOST_AFTER" | tee -a hostname-check.txt
test "$HOST_BEFORE" = "$HOST_AFTER"
grep -F 'rebash-lab22-ns' unshare-uts.txt

# Show your process namespace IDs from /proc
ls -l /proc/self/ns | tee self-ns.txt
```

Expected output: `lsns.txt` lists namespaces; `unshare-uts.txt` shows the temporary hostname; host hostname unchanged.

Task 2 – cgroup v2 paths and controllers

```
cd ~/rebash-linux/lab22
set -euo pipefail

findmnt /sys/fs/cgroup | tee cgroup-mount.txt
```

```
# cgroup.controllers lists available controllers on cgroup v2
if [[ -f /sys/fs/cgroup/cgroup.controllers ]]; then
  cat /sys/fs/cgroup/cgroup.controllers | tee cgroup-controllers.txt
  echo 'cgroup_v2=yes' | tee cgroup-version.txt
else
  echo 'cgroup_v2=no_or_hybrid' | tee cgroup-version.txt
  ls /sys/fs/cgroup | head | tee cgroup-controllers.txt
fi

cat /proc/self/cgroup | tee self-cgroup.txt
# If systemd is present, show a small slice of the tree
systemd-cgls --no-pager 2>/dev/null | head -n 40 | tee systemd-cgls.txt || \
  echo 'systemd-cgls not available' | tee systemd-cgls.txt

# Memory pressure / basic facts (read-only)
grep -E 'MemTotal|MemAvailable' /proc/meminfo | tee meminfo-snip.txt
```

Expected output: cgroup mount and controllers (or honest hybrid note); `self-cgroup.txt` non-empty.

Task 3 – OverlayFS, OCI runtime detection, optional engine

```
cd ~/rebash-linux/lab22
set -euo pipefail

findmnt -t overlay 2>/dev/null | tee overlay-mounts.txt || true
if [[ ! -s overlay-mounts.txt ]]; then
  echo 'no overlay mounts right now (normal without running containers)' | tee overlay-mounts.txt
fi

{
  echo "runc=$(command -v runc || true)"
  echo "crun=$(command -v crun || true)"
  echo "docker=$(command -v docker || true)"
  echo "podman=$(command -v podman || true)"
  echo "containerd=$(command -v containerd || true)"
} | tee runtime-paths.txt

if command -v runc >/dev/null 2>&1; then runc --version 2>&1 | tee runc-version.txt; else echo 'runc absent' | tee runc-version.txt; fi
if command -v docker >/dev/null 2>&1; then
  docker version 2>&1 | head -n 20 | tee docker-version.txt
  # Optional tiny pull-less demo if docker works: run busybox echo (needs network/image)
  # Skip automatic pull in locked-down labs — document only
  echo 'docker present — optional: docker run --rm public.ecr.aws/docker/library/busybox:1.36 echo ok' | tee docker-note.txt
else
  echo 'docker absent' | tee docker-version.txt
  echo 'engine optional for this lab' | tee docker-note.txt
fi

# Disk hint for image stores (common paths)
du -sh /var/lib/docker 2>/dev/null | tee docker-disk.txt || echo 'no /var/lib/docker' | tee docker-disk.txt
du -sh /var/lib/containerd 2>/dev/null | tee containerd-disk.txt || echo 'no /var/lib/containerd' | tee containerd-disk.txt

tar -czf container-internals-evidence.tgz \
  admin-user.txt kernel.txt \
  lsns.txt lsns-filtered.txt unshare-uts.txt hostname-check.txt self-ns.txt \
  cgroup-mount.txt cgroup-controllers.txt cgroup-version.txt self-cgroup.txt \
  systemd-cgls.txt meminfo-snip.txt \
  overlay-mounts.txt runtime-paths.txt runc-version.txt docker-version.txt docker-note.txt \
  docker-disk.txt containerd-disk.txt

ls -l container-internals-evidence.tgz | tee evidence-ls.txt
test -s container-internals-evidence.tgz
```

Expected output: runtime paths file exists; overlay note or mounts listed; evidence tarball non-empty.

Validation steps

- [ ] `lsns.txt` lists namespaces on the host
- [ ] `unshare` UTS demo changed hostname only inside the namespace
- [ ] cgroup mount / controllers captured
- [ ] Overlay and runtime detection files exist (even if “absent”)
- [ ] `container-internals-evidence.tgz` exists under `~/rebash-linux/lab22`

Common errors and fixes

Error	Cause	Fix
`unshare: operation not permitted`	Restricted environment	Use a full VM; some shared hosts block namespace creation
No overlay mounts	No containers running	Expected — keep the note file
`docker: permission denied`	User not in `docker` group	Use `sudo docker` on practice VM, or skip engine extras
Confusing cgroup v1 paths	Older/hybrid setup	Record what you see; prefer documenting v2 `cgroup.controllers` when present

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `~/rebash-linux/lab22/ns-report.sh` that prints: count of namespaces from `lsns`, whether `/sys/fs/cgroup/cgroup.controllers` exists, and whether any of `runc` / `crun` / `docker` / `podman` is on `PATH`. Save output to `ns-report.out` and exit 0 on success.

Learning outcomes

- Separated “isolation” (namespaces) from “limits” (cgroups)
- Used `unshare` safely for a UTS demo
- Inspected cgroup v2 and overlay/runtime signals on the host
- Packed evidence for node-level container debugging

Cleanup

```
cd ~/rebash-linux/lab22
set -euo pipefail
# No persistent namespaces from the UTS demo
# If you started containers manually, remove them yourself
# Keep container-internals-evidence.tgz if useful
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab22/` with evidence files
- [ ] You can explain namespaces vs cgroups in one clear sentence each
- [ ] You can relate OverlayFS to image layers and node disk growth
- [ ] You know where OCI runtimes fit under Docker/Kubernetes

Code Walkthrough

Host-level container debugging usually follows this order:

1. **See isolation** — `lsns`, `/proc/<pid>/ns`
2. **See limits** — `/proc/<pid>/cgroup`, `/sys/fs/cgroup`, OOM logs
3. **See filesystem** — `findmnt -t overlay`, disk under `/var/lib/...`
4. **See runtime** — engine + `runc` / `crun` versions
5. **Change least first** — fix requests/limits and image GC before exotic kernel tweaks

Security Considerations

- Containers are not a full security boundary by themselves
- Prefer non-root containers, read-only rootfs, and drop capabilities
- Combine with MAC (AppArmor/SELinux) and seccomp profiles
- Limit who can talk to the Docker/containerd socket (root-equivalent)
- Scan images and pin digests in production pipelines

Common Mistakes

Calling a container a lightweight VM

It shares the host kernel. **Fix:** say "isolated process with namespaces and cgroups"; use VMs when you need a separate kernel.

Debugging only inside the container

Node pressure and overlay growth are host facts. **Fix:** check `df`, `lsns`, cgroup OOM, and runtime disk paths on the node.

Ignoring cgroup memory limits

The kernel OOM-kills the cgroup; the app looks "random". **Fix:** read events under the cgroup, `dmesg` / `journalctl -k`, and right-size limits.

Leaving docker.sock world-accessible

Socket access is effectively root. **Fix:** restrict group membership; prefer rootless or moderated APIs in platforms.

Best Practices

- Standardise on cgroup v2-capable node images
- Set resource requests/limits with evidence from monitoring
- Garbage-collect images and manage container log size
- Know your runtime chain: kubelet → CRI → containerd → `runc` / `crun`
- Practise host inspection before the first production page-out

Troubleshooting

Symptom	Likely cause	Fix
OOMKilled	cgroup memory limit / node pressure	Raise limit carefully or fix leak; check node `MemAvailable`
Disk full on node	Images, overlays, logs	`du` on `/var/lib/containerd` or docker; prune safely
Network weird in pod	netns / CNI / NetworkPolicy	Debug from host `ip netns` / CNI docs — not only `curl` in pod
`operation not permitted`	seccomp/AppArmor/caps	Inspect securityContext and MAC profiles
Slow pulls	Registry / disk I/O	Check `iostat`, mirror registry, slim images

Summary

Containers are kernel features plus an OCI runtime contract. Namespaces isolate views; cgroups limit resources; OverlayFS layers filesystems; engines make it usable. Next, practise a disciplined host debugging loop in Troubleshooting Linux Systems.

Interview Questions

INTERVIEW PRACTICE

1. In one sentence each, what do namespaces and cgroups do?

Answer

Namespaces change what a process can *see* (PID list, network stack, mounts, hostname, and more). **cgroups** limit and account what a process can *consume* (CPU, memory, I/O, PIDs). Interviewers listen for this see-vs-consume split.

2. Why is a container not a virtual machine?

Answer

A VM runs a guest kernel on a hypervisor. A container is a process on the **host kernel** with isolation and limits applied. That is why kernel CVEs and host tuning affect all containers on the node.

3. How does OverlayFS relate to Docker/container image layers?

Answer

Image layers are usually stacked as read-only lower layers with a writable upper layer. OverlayFS makes that stack look like one root filesystem. Deleted data and unused images can still consume disk until garbage collection — node `df` matters.

4. A pod is OOMKilled. Where do you look on the Linux node?

Answer

Check the container/pod cgroup memory events, node `journalctl -k` / `dmesg` for OOM, `free` / `MemAvailable`, and whether limits in the manifest are too low or the app leaked. Do not restart forever without reading cgroup evidence.

5. What is the OCI runtime's job compared to Docker/containerd?

Answer

Docker/containerd handle images, API, and higher-level lifecycle. An OCI runtime such as `runc` or `crun` performs the low-level create/start work: namespaces, cgroups, rootfs, and exec of the entrypoint. Kubernetes talks CRI to a runtime that ultimately uses OCI.

6. How can you demonstrate a namespace on a host without Docker?

Answer

Use `unshare` (for example `--uts`) to change hostname inside a new namespace and show the host hostname is unchanged. Use `lsns` and `/proc/self/ns` to list namespace inodes. Keep demos on a practice VM.

7. Why can access to `docker.sock` be dangerous?

Answer

Clients of the Docker socket can often run privileged containers and mount the host filesystem — effectively **root on the host**. Restrict membership, prefer rootless where possible, and treat socket access as highly privileged in audits.

Troubleshooting Linux Systems
MODULE 15 · TROUBLESHOOTING

Chapter 23. Troubleshooting Linux Systems

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/troubleshooting-linux-systems/

Overview

Troubleshooting is not random command typing. It is a **method**: define the symptom, gather facts, narrow the scope, make one change at a time, and prove recovery with evidence. On Linux hosts the first facts usually come from **time**, **recent changes**, **failed units**, **disk**, **memory**, **network listen ports**, and **logs**.

In Cloud and DevOps work you troubleshoot jump servers, Continuous Integration (CI) runners, Kubernetes nodes, and application VMs. The tools differ slightly; the method stays the same. In this tutorial you will practise a checklist, break and fix a small systemd unit on purpose, and save an incident-style evidence pack under `~/rebash-linux/lab23`.

In production, communicate blast radius (how much is affected) in plain language, avoid untracked changes, and write down what you tried. Guessing without evidence extends outages.

This is **Tutorial 23** in **Module 15: Troubleshooting** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Containers — Namespaces, cgroups, and OCI
- Comfort with systemd Services and journalctl
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Apply a fact-first troubleshooting checklist on a Linux host
- [ ] Use `systemctl --failed`, `journalctl`, `df`, and `ss` as early signals
- [ ] Diagnose a broken systemd unit from status + journal
- [ ] Fix forward with a reversible change and prove recovery
- [ ] Pack incident evidence under `~/rebash-linux/lab23`

Architecture

Troubleshooting walks from user symptom → host signals → component logs → targeted fix → validation.

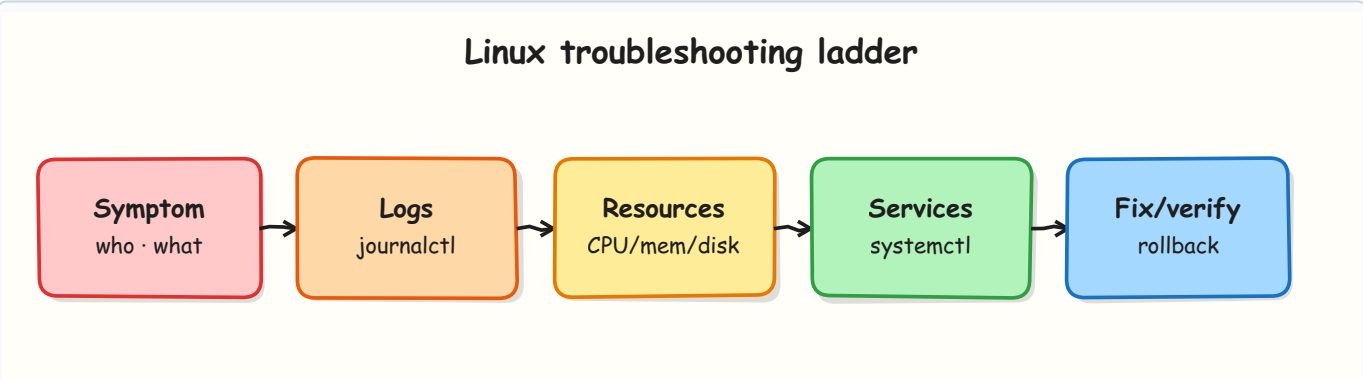

Figure 23.1: Troubleshooting Linux Systems

Theory

What it is

A practical loop:

1. **Symptom** — what fails, since when, for whom?
2. **Scope** — one host, one service, or many?
3. **Facts** — status, logs, resources, recent changes
4. **Hypothesis** — one likely cause
5. **Action** — smallest safe change
6. **Proof** — symptom gone; evidence saved

```
systemctl --failed
journalctl -xe --no-pager | tail
df -hT
ss -lntu
```

Why it matters

Unstructured troubleshooting causes longer outages and new failures. Interviewers and incident commanders look for method and evidence, not memorised trivia.

How it works

Area	First commands
Services	`systemctl status`, `--failed`, `journalctl -u`
Capacity	`df -hT`, `df -i`, `free -h`
CPU/I/O	`vmstat`, `iostat`
Network	`ip -br a`, `ss -lntu`
Auth/SSH	`journalctl -u ssh`, auth logs

Common pitfalls

- Changing three things at once.
- Rebooting as the first step without capturing logs.
- Fixing a symptom on the wrong host.
- No before/after proof in the ticket.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Run a host fact pack, deploy a systemd unit that fails on purpose, diagnose it with `systemctl`/`journalctl`, fix it, prove it is active, and save evidence under `~/rebash-linux/lab23`.

Prerequisites

- Ubuntu with `sudo` and systemd

Lab environment

Workspace: `~/rebash-linux/lab23`

```
mkdir -p ~/rebash-linux/lab23 && cd ~/rebash-linux/lab23
set -euo pipefail
date -Is | tee incident-start.txt
whoami | tee operator.txt
```

Expected output: timestamp and operator files exist.

Real-world scenario

A practice “health writer” service should create `/var/tmp/rebash-lab23.ok` every time it runs. After a bad config change it fails. You are on call: gather host facts, find the failed unit, fix the ExecStart path, and attach proof that the unit is active again.

Step-by-step tasks

Task 1 – Host fact pack

```
cd ~/rebash-linux/lab23
set -euo pipefail

uname -a | tee uname.txt
cat /etc/os-release | tee os-release.txt
uptime | tee uptime.txt
df -hT | tee df.txt
free -h | tee free.txt
systemctl is-system-running | tee systemd-state.txt || true
systemctl --failed --no-pager | tee failed-before.txt || true
ss -lntu | head -n 30 | tee ss.txt
ip -br a | tee ip.txt
```

Expected output: fact files created; systemd state captured (may be `running` or `degraded`).

Task 2 – Break a unit on purpose and diagnose

```
cd ~/rebash-linux/lab23
set -euo pipefail

# Broken unit: ExecStart points to a missing script
sudo tee /etc/systemd/system/rebash-lab23.service >/dev/null << 'EOF'
[Unit]
Description=REBASH lab23 health writer (intentionally broken first)
After=network.target

[Service]
Type=oneshot
ExecStart=/usr/local/bin/rebash-lab23-missing.sh
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl daemon-reload
set +e
sudo systemctl start rebash-lab23.service
set -e
systemctl status rebash-lab23.service --no-pager -l | tee status-broken.txt || true
journalctl -u rebash-lab23.service -n 30 --no-pager | tee journal-broken.txt
grep -Ei 'No such file|failed|status=' journal-broken.txt status-broken.txt
systemctl is-failed rebash-lab23.service | tee is-failed.txt
test "$(cat is-failed.txt)" = "failed"
```

Expected output: unit is `failed`; journal/status mention the missing ExecStart path.

Task 3 – Fix forward, prove recovery, pack evidence

```
cd ~/rebash-linux/lab23
set -euo pipefail

sudo tee /usr/local/bin/rebash-lab23-health.sh >/dev/null << 'EOF'
#!/bin/bash
set -euo pipefail
date -Is > /var/tmp/rebash-lab23.ok
EOF
sudo chmod 755 /usr/local/bin/rebash-lab23-health.sh

sudo tee /etc/systemd/system/rebash-lab23.service >/dev/null << 'EOF'
[Unit]
Description=REBASH lab23 health writer
After=network.target

[Service]
Type=oneshot
ExecStart=/usr/local/bin/rebash-lab23-health.sh
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl daemon-reload
```

```
sudo systemctl reset-failed rebash-lab23.service || true
sudo systemctl start rebash-lab23.service
systemctl is-active rebash-lab23.service | tee is-active.txt
test "$(cat is-active.txt)" = "active"
test -f /var/tmp/rebash-lab23.ok
cat /var/tmp/rebash-lab23.ok | tee health-ok.txt
systemctl status rebash-lab23.service --no-pager -l | tee status-fixed.txt
journalctl -u rebash-lab23.service -n 20 --no-pager | tee journal-fixed.txt

tar -czf troubleshooting-evidence.tgz \
  incident-start.txt operator.txt uname.txt os-release.txt \
  uptime.txt df.txt free.txt systemd-state.txt failed-before.txt ss.txt ip.txt \
  status-broken.txt journal-broken.txt is-failed.txt \
  is-active.txt health-ok.txt status-fixed.txt journal-fixed.txt
ls -l troubleshooting-evidence.tgz | tee evidence-ls.txt
```

Expected output: unit `active`; `/var/tmp/rebash-lab23.ok` exists; evidence archive not empty.

Validation steps

- [ ] Fact pack files exist under `~/rebash-linux/lab23`
- [ ] Broken state showed `failed` with a clear journal reason
- [ ] Fixed unit is `active` and wrote the ok file
- [ ] `troubleshooting-evidence.tgz` exists

Common errors and fixes

Error	Cause	Fix
`Unit not found`	Forgot `daemon-reload`	`sudo systemctl daemon-reload`
Still failed after fix	Old failure cached / wrong path	`reset-failed`; verify ExecStart exists
Cannot write `/etc/systemd/system`	No sudo	Use a practice VM with admin rights
`is-active` is `inactive`	oneshot without RemainAfterExit	Keep `RemainAfterExit=yes` as in the lab

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Extend the unit into a simple **restarting service** (`Type=simple`) that loops `sleep 30` after rewriting the ok file, enable it, prove `active (running)`, then stop and disable it. Save `systemctl status` to `challenge-status.txt`.

Learning outcomes

- Ran a repeatable host fact pack
- Diagnosed a failed unit from status + journal
- Fixed forward and proved recovery
- Built an incident evidence archive

Cleanup

```
cd ~/rebash-linux/lab23
set -euo pipefail
sudo systemctl disable --now rebash-lab23.service 2>/dev/null || true
sudo rm -f /etc/systemd/system/rebash-lab23.service
sudo rm -f /usr/local/bin/rebash-lab23-health.sh
sudo systemctl daemon-reload
sudo rm -f /var/tmp/rebash-lab23.ok
# Keep troubleshooting-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab23/` with evidence files
- [ ] You can list the first five fact commands you run on a sick host
- [ ] You know why rebooting first can destroy evidence
- [ ] You can explain failed → fixed with journal proof

Code Walkthrough

Production incident habit:

1. Announce symptom and scope
2. Capture facts (do not reboot yet)
3. Form one hypothesis
4. Change one thing; watch logs
5. Confirm user symptom cleared; attach evidence

Security Considerations

- Prefer read-only gathering before privileged changes
- Do not paste secrets from logs into tickets
- Use break-glass admin accounts carefully (emergency admin)
- Record who changed what during the incident
- Limit SSH access while you work on internet-facing hosts

Common Mistakes

Rebooting before collecting logs

Volatile evidence disappears. **Fix:** `journalctl`, `systemctl status`, and resource snapshots first.

Changing config and restarting three services at once

You cannot tell what fixed it. **Fix:** one change, then re-check.

Troubleshooting the wrong machine

Load balancers hide targets. **Fix:** confirm hostname/IP/instance ID with the reporter.

Declaring victory without a user-visible check

Unit active ≠ feature works. **Fix:** hit the real health URL or canary file users care about.

Best Practices

- Keep a personal one-page checklist
- Prefer reversible changes and feature flags
- Use configuration management after emergency hotfixes
- Write a short timeline in the ticket
- Practise failure drills on lab VMs

Troubleshooting

Symptom	Likely cause	Fix
Service failed	Bad ExecStart/config	status + journal; fix path/config
Disk full	Logs/containers	`df` / `du`; rotate; expand
No listen port	App/crash/firewall	`ss`, journal, security groups
High load	CPU/I/O/memory	vmstat/iostat/free; top PIDs
Intermittent failure	Race/deps/time	journals across boots; chrony

Summary

Method beats memory. Gather facts, narrow scope, fix forward, prove recovery, and keep evidence. Next: Production Hardening and Performance.

Interview Questions

INTERVIEW PRACTICE

1. What are the first five commands you run on an unfamiliar sick Linux VM?

Answer

A solid set is: `uptime` (load/time since boot), `df -hT` (+ `df -i`), `free -h`, `systemctl --failed`, and `journalctl -xe` / `journalctl -u <service>`. Add `ss -lntu` and `ip -br a` when the symptom is network. Explain *why* each command, not only the names.

2. A service is `failed`. How do you proceed?

Answer

`systemctl status name -l`, then `journalctl -u name --since …`, fix the root cause (path, permissions, config), `daemon-reload` if units changed, `reset-failed` if needed, start, and prove with `is-active` plus an application check. Avoid reboot as step one.

3. Why is "reboot the server" a weak first answer in interviews?

Answer

Reboot may clear symptoms without understanding cause, destroy volatile evidence, and hide recurring bugs. Prefer capturing logs and status first; reboot only when justified (kernel deadlock, exhausted resources with a plan).

4. How do you decide if the problem is disk vs memory vs CPU?

Answer

Use `df` / `df -i` for disk, `free` / `vmstat` swap fields for memory, and `vmstat` / `top` runnable vs idle for CPU. High `wa` points to I/O. Correlate with the application symptom and recent changes.

5. What evidence belongs in an incident ticket?

Answer

Timeline, scope, commands run, key outputs (`status`, journal snippets, `df`), changes made, and proof of recovery. Redact secrets. Before/after is stronger than "we restarted it".

6. How does troubleshooting a Kubernetes node differ from an app VM?

Answer

You still use host facts (`df`, journal for kubelet/runtime), but you also check node conditions, pods on the node, and cluster events. Decide whether the failure is node-level (disk pressure) or workload-level. Do not apply random `kubectl delete` without scope.

7. What does "fix forward" mean?

Answer

Make the smallest safe change that restores service (correct config, free disk, restart one unit) while recording evidence — rather than only rolling back blindly or changing many things. Rollback is still valid when it is the safest path; either way, prove the user symptom is gone.

Production Hardening and Performance
MODULE 16 · PRODUCTION LINUX

Chapter 24. Production Hardening and Performance

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/production-linux-hardening-and-performance/

Overview

Production Linux hosts need two habits at once: **hardening** (reduce attack surface and limit blast radius) and **performance baselines** (know normal CPU, memory, disk, and network behaviour). Hardening without observability creates brittle hosts. Performance tuning without security creates fast vulnerable hosts.

Practical baselines include: time sync (chrony), kernel parameters via **sysctl**, user/process **resource limits**, unattended security updates policy, and regular audit of listening ports and failed units. In this tutorial you will inspect current baselines, apply a **lab-scoped** sysctl drop-in and limits drop-in, verify them, and save proof under `~/rebash-linux/lab24`. Changes stay namespaced to REBASH lab files so Cleanup is safe.

In real estates, prefer golden images and configuration management over one-off SSH edits. Test sysctl changes on practice VMs — some settings can break apps if copied blindly from blog posts.

This is **Tutorial 24** in **Module 16: Production Linux** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Troubleshooting Linux Systems
- A **practice Ubuntu 22.04/24.04 VM** with `sudo`
- Do **not** apply experimental kernel tuning on shared production without change control

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain a minimal production baseline (time, updates, limits, sysctl, audit)
- [ ] Add a sysctl drop-in and prove it with `sysctl`
- [ ] Add a limits drop-in and prove the configured values
- [ ] Capture listening ports and failed units as an audit snapshot
- [ ] Pack evidence under `~/rebash-linux/lab24`

Architecture

Hardening and performance controls sit across identity, network exposure, kernel parameters, resource limits, and monitoring feedback loops.

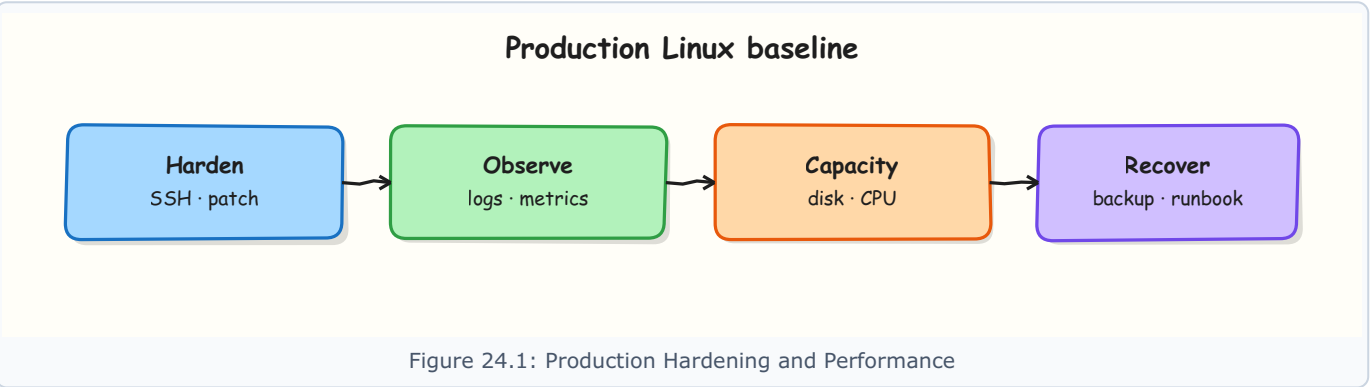

Figure 24.1: Production Hardening and Performance

Theory

What it is

Area	Examples
Identity & access	sudo least privilege, SSH keys
Network exposure	firewall, few listen ports
Kernel/OS	sysctl, chrony, automatic security updates
Resources	ulimits, cgroup limits on services
Feedback	metrics, logs, failed-unit alerts

```
timedatectl
sysctl net.ipv4.ip_forward
ss -lntu
systemctl --failed
```

Why it matters

Open SSH with password auth, no time sync, and unlimited processes are common root causes of incidents and audit failures. Performance regressions often start as “we never recorded a baseline”.

How it works

1. Measure current state
2. Apply small, named drop-in files
3. Verify with read-back commands
4. Monitor impact
5. Codify in images/config management

Knob	Careful note
`fs.file-max` / nofile limits	Apps may need higher file descriptors
`vm.swappiness`	Workload-dependent
IP forwarding	Only when the host is a router
Automatic reboots	Coordinate maintenance windows

Common pitfalls

- Copy-pasting huge sysctl “performance” lists without testing.
- Raising limits globally instead of per-service.
- Disabling swap or firewalls “for speed”.
- No chrony — TLS and logs become confusing.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Capture a production-style audit snapshot, install lab sysctl and limits drop-ins, verify read-back, and save evidence under `~/rebash-linux/lab24`.

Prerequisites

- Ubuntu practice VM with `sudo`

Lab environment

Workspace: `~/rebash-linux/lab24`

```
mkdir -p ~/rebash-linux/lab24 && cd ~/rebash-linux/lab24
set -euo pipefail
sudo apt-get update -qq
```

```
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y chrony
timedatectl | tee timedatectl.txt
```

Expected output: chrony present; `timedatectl.txt` shows clock sync fields.

Real-world scenario

Security and platform teams ask for a baseline on a new Ubuntu app VM: time sync on, a documented sysctl drop-in, higher file descriptor limits for the app user group, and a snapshot of listening ports. You implement lab-scoped files and keep proof for the hardening ticket.

Step-by-step tasks

Task 1 – Audit snapshot

```
cd ~/rebash-linux/lab24
set -euo pipefail

uname -a | tee uname.txt
cat /etc/os-release | tee os-release.txt
timedatectl show | tee timedatectl-show.txt
sysctl fs.file-max net.ipv4.ip_forward vm.swappiness | tee sysctl-before.txt
ulimit -n | tee ulimit-n-before.txt
ss -lntu | tee ss-listen.txt
systemctl --failed --no-pager | tee failed-units.txt || true
df -hT | tee df.txt
free -h | tee free.txt
```

Expected output: audit files exist; sysctl values captured before change.

Task 2 – Lab sysctl drop-in

```
cd ~/rebash-linux/lab24
set -euo pipefail

# Conservative lab-only example: raise file-max modestly and keep ip_forward=0
AFTER_MAX=$(( $(sysctl -n fs.file-max) + 1000 ))
echo "$AFTER_MAX" | tee target-file-max.txt

sudo tee /etc/sysctl.d/99-rebash-lab24.conf >/dev/null << EOF
# REBASH lab24 - remove in cleanup
fs.file-max = ${AFTER_MAX}
net.ipv4.ip_forward = 0
EOF

sudo sysctl --system 2>&1 | tee sysctl-apply.txt
sysctl fs.file-max net.ipv4.ip_forward | tee sysctl-after.txt
test "$(sysctl -n fs.file-max)" -eq "$AFTER_MAX"
test "$(sysctl -n net.ipv4.ip_forward)" -eq 0
```

Expected output: `sysctl-after.txt` shows the new `fs.file-max` and `ip_forward = 0`.

Task 3 – Limits drop-in + evidence pack

```
cd ~/rebash-linux/lab24
set -euo pipefail

sudo tee /etc/security/limits.d/99-rebash-lab24.conf >/dev/null << 'EOF'
# REBASH lab24 - remove in cleanup
* soft nofile 4096
* hard nofile 8192
EOF

# limits.d applies to new login sessions; prove the file content and pam path exist
cat /etc/security/limits.d/99-rebash-lab24.conf | tee limits-file.txt
grep -n 'pam_limits.so' /etc/pam.d/common-session | tee pam-limits.txt || \
  grep -rn 'pam_limits.so' /etc/pam.d | head | tee pam-limits.txt

# Performance-oriented snapshot after changes
vmstat 1 3 | tee vmstat.txt
iostat -xz 1 2 2>/dev/null | tee iostat.txt || echo 'install sysstat for iostat' | tee iostat.txt

tar -czf production-evidence.tgz \
  timedatectl.txt timedatectl-show.txt uname.txt os-release.txt \
  sysctl-before.txt sysctl-after.txt sysctl-apply.txt target-file-max.txt \
  ulimit-n-before.txt limits-file.txt pam-limits.txt \
  ss-listen.txt failed-units.txt df.txt free.txt vmstat.txt iostat.txt
ls -l production-evidence.tgz | tee evidence-ls.txt
```

Expected output: limits file installed; pam_limits referenced; evidence archive exists.

Validation steps

- [ ] Time sync status captured with `timedatectl`
- [ ] `/etc/sysctl.d/99-rebash-lab24.conf` applied and verified
- [ ] `/etc/security/limits.d/99-rebash-lab24.conf` exists
- [ ] `production-evidence.tgz` exists under `~/rebash-linux/lab24`

Common errors and fixes

Error	Cause	Fix
sysctl did not change	Typo / not loaded	`sysctl --system`; check file under `/etc/sysctl.d/`
`ulimit -n` unchanged in current shell	limits apply on new sessions	Open a new login SSH session to see new soft limit
chrony not syncing	Network/NTP blocked	Check security groups; `chronyc tracking`
`iostat` missing	sysstat not installed	Optional install; not required for pass

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Create a systemd **service drop-in** directory for an existing unit you are allowed to edit in the lab (or a lab unit you create) that sets `LimitNOFILE=8192`, then show `systemctl show -p LimitNOFILE …` in `service-limit.txt`. Remove it in Cleanup.

Learning outcomes

- Captured a host hardening/performance audit snapshot
- Applied and verified a sysctl drop-in
- Added limits.d configuration with proof files
- Packed production baseline evidence

Cleanup

```
cd ~/rebash-linux/lab24
set -euo pipefail
sudo rm -f /etc/sysctl.d/99-rebash-lab24.conf
sudo rm -f /etc/security/limits.d/99-rebash-lab24.conf
sudo sysctl --system >/dev/null
# Keep production-evidence.tgz if you want it
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab24/` with evidence files
- [ ] You can explain why drop-in files beat editing primary configs blindly
- [ ] You know limits often need a new login session
- [ ] You treat sysctl changes as change-controlled production work

Code Walkthrough

Production rollout pattern:

1. Audit current host
2. Propose small drop-ins in git
3. Test on a twin VM
4. Apply via config management
5. Watch metrics and failed units

Security Considerations

- Least privilege sudo and key-only SSH on internet-facing hosts
- Minimise listening ports; firewall default deny inbound
- Keep automatic security updates or a patch pipeline
- Do not disable security modules “for performance” without review

- Protect sysctl/limits change rights

Common Mistakes

Huge blog sysctl packs on day one

Untested settings break apps. **Fix:** change one parameter, measure, then keep.

Global ulimit raises for everyone

Masks leaks and surprises other users. **Fix:** prefer systemd `LimitNOFILE` on the app unit.

No time synchronisation

Certificates and log correlation fail. **Fix:** install/enable chrony; monitor sync.

Calling a host "hardened" after one sysctl

Hardening is layered. **Fix:** cover SSH, patches, users, firewall, audit, backups.

Best Practices

- Golden images + config as code
- Baselines and alerts for saturation
- Document every non-default sysctl
- Separate OS and data disks
- Regular restore and failover tests (next tutorial)

Troubleshooting

Symptom	Likely cause	Fix
App "too many open files"	nofile too low	Raise unit LimitNOFILE; verify
Sysctl reverts after reboot	File not in sysctl.d	Place under `/etc/sysctl.d/`
Clock offset	chrony/NTP blocked	Fix network; restart chrony
Performance worse after tune	Bad parameter	Roll back drop-in; retest
Audit fails on open ports	Unexpected listeners	`ss -lntup`; remove/stop service

Summary

Production hosts need layered hardening and honest performance baselines. Use small drop-in files, verify read-back, and codify what works. Next: Backup, Disaster Recovery, and Capacity.

Interview Questions

INTERVIEW PRACTICE

1. What belongs in a minimal Linux production baseline?

Answer

Patch process, time sync, least-privilege access, firewall/SSH hardening, logging/metrics, resource limits for apps, backups/restore tests, and documented kernel/sysctl exceptions. Exact tools vary; the categories should not.

2. Why prefer `/etc/sysctl.d/` drop-ins over editing `/etc/sysctl.conf` only?

Answer

Drop-ins are easier to own per team/role, review in git, and remove cleanly. They reduce merge conflicts and make intent obvious (`99-app.conf` vs a giant shared file).

3. A process still sees the old `ulimit -n` after you edited limits.d. Why?

Answer

limits.d values apply to **new** login sessions (via PAM). Existing shells keep old limits. Systemd services should set `LimitNOFILE=` on the unit for reliable service limits.

4. How do hardening and performance conflict, and how do you balance them?

Answer

Example: verbose audit logging adds I/O; very low timeouts can break slow clients. Balance with measured SLOs: keep security defaults, raise specific limits for known apps, and watch metrics after each change.

5. What sysctl change is dangerous to copy from the internet without context?

Answer

Anything affecting networking (forwarding, rp_filter, connection tracking) or virtual memory behaviour can break routing or latency. Always test on a practice VM and know the rollback file.

6. How would you prove a hardening change in a ticket?

Answer

Show the drop-in contents, `sysctl` / `systemctl show` read-back, listening port snapshot, and that critical services still pass health checks. Before/after matters.

7. Where does SSH hardening fit relative to this tutorial?

Answer

SSH and firewalls are a major hardening slice covered in dedicated modules. This tutorial focuses on host baselines (time, sysctl, limits, audit snapshots) that complement network access controls.

Backup, Disaster Recovery, and Capacity
MODULE 16 · PRODUCTION LINUX

Chapter 25. Backup, Disaster Recovery, and Capacity

Online lab & updates
Scan or open the live tutorial (includes the hands-on lab): https://rebash.in/linux/backup-disaster-recovery-and-capacity/

Overview

A backup you have never restored is only a hope. **Backup** copies data so you can recover from deletion, corruption, or bad deploys. **Disaster Recovery (DR)** is the plan and practise to restore service after a larger failure (lost VM, lost region, lost disk). **Capacity** planning watches disk, CPU, and memory so you grow before you break.

Two numbers appear in every serious DR conversation: **Recovery Point Objective (RPO)** — how much data you can afford to lose (time since last good backup), and **Recovery Time Objective (RTO)** — how long recovery may take. In this tutorial you will create sample data, back it up with `tar` and `rsync`, destroy the original on purpose, restore, verify checksums, and capture capacity signals under `~/rebash-linux/lab25`.

In production, use platform tools too (cloud snapshots, database native backups, object storage). Host-level `tar` / `rsync` skills still matter for configs, small app trees, and proving you understand restore.

This is **Tutorial 25** in **Module 16: Production Linux** of the REBASH Academy **Linux for Cloud & DevOps Engineers** series. It is written for Linux administrators, DevOps engineers, Site Reliability Engineering (SRE), and platform engineers.

Prerequisites

- Production Hardening and Performance
- A **practice Ubuntu 22.04/24.04 VM** with write space under `$HOME`
- Package `rsync` (install if missing)

Learning Objectives

By the end of this tutorial, you will be able to:

- [ ] Explain RPO and RTO in plain language
- [ ] Create a `tar` archive backup and restore it
- [ ] Mirror a directory with `rsync` and prove a restore after deletion
- [ ] Verify restored data with checksums
- [ ] Capture capacity signals (`df`, `du`) with the backup evidence under `~/rebash-linux/lab25`

Architecture

Backups copy data to safe storage; DR restores service within RPO/RTO; capacity monitoring prevents “no space for backups” failures.

Backup, Disaster Recovery, and Capacity

Figure 25.1: Backup, Disaster Recovery, and Capacity

Theory

What it is

Term	Meaning
Backup	Copy of data for restore
Restore test	Proving the copy works
RPO	Max acceptable data loss window
RTO	Max acceptable downtime to recover
Capacity	Headroom for growth and for backup storage

```
tar -czf backup.tgz data/
rsync -aH data/ backup-mirror/
df -hT
```

Why it matters

Ransomware, bad migrations, and accidental `rm` are normal risks. Cloud snapshots help, but application consistency and restore drills matter. Disks that are 95% full often fail the backup job first.

How it works

1. Identify critical data and RPO/RTO
2. Copy off-box when possible
3. Automate schedules (cron/timer)
4. **Restore regularly** into a scratch path
5. Watch capacity of source and backup target

Tool	Good for
`tar`	Portable archives of trees
`rsync`	Incremental mirrors
Snapshots	Whole disk/VM points in time
DB dumps	Consistent database recovery

Common pitfalls

- Backups on the same disk as the source only.
- Never testing restore.
- No room left for the backup target.
- Backing up without application quiesce when consistency matters.

Hands-on Lab

TRY IT YOURSELF — LAB

Objective

Create sample app data, back it up with `tar` and `rsync`, delete the live data, restore from both methods, verify checksums, and pack evidence under `~/rebash-linux/lab25`.

Prerequisites

- Ubuntu with `tar`, `rsync`, `sha256sum`

Lab environment

Workspace: `~/rebash-linux/lab25`

```
mkdir -p ~/rebash-linux/lab25 && cd ~/rebash-linux/lab25
set -euo pipefail
sudo apt-get update -qq
```

```
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y rsync
df -hT . | tee df-before.txt
```

Expected output: `rsync` available; capacity snapshot stored.

Real-world scenario

A small app keeps critical config and upload files under one directory. Leadership asks for an RPO of "last hourly backup" and proof that restore works. You rehearse archive + mirror backups, destroy the live tree, restore, and attach checksum proof to the DR document.

Step-by-step tasks

Task 1 – Create data and checksum manifest

```
cd ~/rebash-linux/lab25
set -euo pipefail

rm -rf appdata backups restore
mkdir -p appdata/conf appdata/uploads backups restore

printf 'role=api\nversion=1\n' > appdata/conf/app.env
printf 'user-upload-1\n' > appdata/uploads/a.txt
printf 'user-upload-2\n' > appdata/uploads/b.txt
# Stable manifest of contents
( cd appdata && find . -type f -print0 | sort -z | xargs -0 sha256sum ) | tee checksums-before.txt
test -s checksums-before.txt
```

Expected output: `checksums-before.txt` lists hashes for the three files.

Task 2 – Backup with tar and rsync

```
cd ~/rebash-linux/lab25
set -euo pipefail

tar -czf backups/appdata.tgz -C appdata .
rsync -aH --delete appdata/ backups/appdata-mirror/
tar -tzf backups/appdata.tgz | tee tar-listing.txt
find backups/appdata-mirror -type f | sort | tee mirror-listing.txt
test -f backups/appdata.tgz
test -f backups/appdata-mirror/conf/app.env
```

Expected output: archive and mirror both contain `conf/app.env`; listings captured.

Task 3 – Destroy, restore, verify, capacity + evidence

```
cd ~/rebash-linux/lab25
set -euo pipefail

# Disaster: live data wiped
rm -rf appdata
test ! -d appdata

# Restore from tar
mkdir -p restore/from-tar
tar -xzf backups/appdata.tgz -C restore/from-tar
( cd restore/from-tar && find . -type f -print0 | sort -z | xargs -0 sha256sum ) | tee checksums-from-tar.txt
cmp checksums-before.txt checksums-from-tar.txt

# Restore from rsync mirror
mkdir -p restore/from-rsync
rsync -aH backups/appdata-mirror/ restore/from-rsync/
( cd restore/from-rsync && find . -type f -print0 | sort -z | xargs -0 sha256sum ) | tee checksums-from-rsync.txt
cmp checksums-before.txt checksums-from-rsync.txt

# Put live data back from tar (simulating recovery)
mkdir -p appdata
tar -xzf backups/appdata.tgz -C appdata
df -hT . | tee df-after.txt
du -sh appdata backups restore | tee du-trees.txt

# Simple RPO/RTO notes for the ticket
cat > dr-notes.txt << 'EOF'
Lab RPO: since last successful backup in backups/
Lab RTO: time to untar/rsync + checksum verify on this VM
Off-box note: copy backups/ to another disk or object storage in real DR
EOF

tar -czf backup-dr-evidence.tgz \
  df-before.txt df-after.txt du-trees.txt \
  checksums-before.txt checksums-from-tar.txt checksums-from-rsync.txt \
```

```
  tar-listing.txt mirror-listing.txt dr-notes.txt
ls -l backup-dr-evidence.tgz | tee evidence-ls.txt
```

Expected output: both `cmp` checks succeed; live `appdata` restored; evidence archive exists.

Validation steps

- [ ] `checksums-before.txt` matches both restore checksum files
- [ ] `backups/appdata.tgz` and `backups/appdata-mirror/` exist
- [ ] Live `appdata` restored after deletion
- [ ] `backup-dr-evidence.tgz` exists under `~/rebash-linux/lab25`

Common errors and fixes

Error	Cause	Fix
`cmp` fails	Different find order / paths	Use the same `find … sort` pipeline
`rsync: command not found`	Package missing	`sudo apt-get install -y rsync`
Archive empty	Wrong `-C` path	`tar -tzf` to inspect before delete
No space for backups	Capacity ignored	Free space; separate backup disk

Challenge exercise

TRY IT YOURSELF — CHALLENGE

Write `~/rebash-linux/lab25/backup-appdata.sh` that creates a timestamped `backups/appdata-YYYYMMDD-HHMMSS.tgz`, writes `backups/latest-checksums.txt`, and exits non-zero if `df` free space on `.` is under 100M. Run it once and keep the new archive.

Learning outcomes

- Defined lab RPO/RTO in plain notes
- Backed up with tar and rsync
- Restored after destructive loss and verified hashes
- Linked capacity checks to backup success

Cleanup

```
cd ~/rebash-linux/lab25
set -euo pipefail
# Keep evidence; remove bulky trees if needed:
# rm -rf appdata backups restore
```

Validation

- [ ] Lab finished under `~/rebash-linux/lab25/` with evidence files
- [ ] You can explain RPO vs RTO with an example
- [ ] You treat restore tests as mandatory
- [ ] You watch capacity on both source and backup target

Code Walkthrough

Production DR habit:

1. Classify data and set RPO/RTO
2. Automate backups off-box
3. Schedule restore drills
4. Monitor backup job success **and** free space
5. Document who runs DR and where credentials live

Security Considerations

- Encrypt backups that contain personal or secret data
- Restrict who can read backup storage

- Do not leave world-readable archives in `/tmp`
- Protect backup credentials like production admin access
- Test restores in isolated networks when data is sensitive

Common Mistakes

Never testing restore

Backups can be corrupt or incomplete. **Fix:** restore to a scratch path on a schedule; keep checksum proof.

Backups only on the same disk

Disk failure loses source and backup together. **Fix:** copy off-box (another disk, another account, object storage).

Ignoring backup target capacity

Jobs fail silently when the target is full. **Fix:** alert on target `df`; retention policies.

Confusing VM snapshot with application backup

Snapshots may be crash-consistent only. **Fix:** use DB-native backups / app quiesce when RPO is strict.

Best Practices

- Write RPO/RTO per service in plain language
- Automate + alert on backup failures
- Quarterly restore drills with notes
- Separate backup accounts and least privilege
- Keep capacity headroom for growth and backups

Troubleshooting

Symptom	Likely cause	Fix
Restore checksum mismatch	Incomplete backup / bitrot	Re-backup; check disk health
`tar` permission errors	Not reading all files	Run with appropriate user; note exclusions
rsync deleted too much	`--delete` misuse	Dry-run (`-n`) first
Backup job OOM/timeout	Huge trees	Incremental strategy; exclude caches
Cannot meet RTO	Slow restore path	Parallel restore; warmer standby

Summary

Backups matter only when restores work. Practise tar/rsync restore, state RPO/RTO clearly, and watch capacity so backup jobs can succeed. This completes the core Linux production module sequence — return to the Linux overview for related labs and interview prep.

Interview Questions

INTERVIEW PRACTICE

1. What is the difference between RPO and RTO?

Answer

RPO (Recovery Point Objective) is how much data you can afford to lose — tied to backup frequency. **RTO (Recovery Time Objective)** is how long recovery may take before the business accepts the outage. Example: RPO 1 hour, RTO 4 hours.

2. Why is a restore test more important than “backup success” green checks?

Answer

Jobs can archive the wrong path, skip files, or write corrupt data while still exiting zero. Only a **restore + verification** (checksums, app start) proves usefulness.

3. When would you choose `rsync` over `tar` for host backups?

Answer

`rsync` is strong for incremental mirrors and efficient repeats. `tar` is strong for portable point-in-time archives. Many designs use both: frequent rsync plus periodic tar/snapshot to object storage.

4. How does capacity planning relate to backups?

Answer

Backup targets need free space and retention room. Source disks that are nearly full may also fail snapshots. Monitor `df` on both sides and alert before jobs fail.

5. Are cloud VM snapshots enough for database DR?

Answer

Snapshots are useful but may be **crash-consistent** only. Databases often need native dumps/backups or filesystem freeze/agent integration for a stricter RPO. Know your consistency requirements.

6. What belongs in a one-page DR runbook?

Answer

Critical systems list, RPO/RTO, where backups live, how to restore (commands/links), who to call, and how to verify success. Keep it short enough to use under stress.

7. How would you prove backup success in a change ticket for this lab?

Answer

Show archive/mirror listings, a deliberate delete, restore commands, and matching `sha256sum` manifests before vs after. That is stronger than “tar completed”.

Glossary

Key terms used in this course. Acronyms are expanded on first use in the chapters as well.

ACL	Access Control List; extra file permissions beyond owner/group/other.
AppArmor	Linux Security Module that confines programmes with path-based profiles (common on Ubuntu).
apt	Debian/Ubuntu package manager front end (`apt` / `apt-get`).
BIOS	Basic Input/Output System; older firmware that starts the boot process.
bootloader	Software (often GRUB) that loads the kernel and initramfs.
cgroup	Control group; kernel feature that limits and accounts CPU, memory, and I/O for process groups.
CI/CD	Continuous Integration / Continuous Delivery (or Deployment).
cron	Time-based job scheduler for recurring commands.
DAC	Discretionary Access Control; classic Unix owner/group/other permissions.
df	Report free disk space on mounted filesystems (blocks and, with `-i`, inodes).
distribution (distro)	A complete Linux OS: kernel plus user space, package manager, and release policy.
dnf	Package manager used on many RHEL-family systems.
DNS	Domain Name System; maps names to IP addresses.
DR	Disaster Recovery; plan and practise to restore service after a major failure.
FHS	Filesystem Hierarchy Standard; conventional layout of `/etc`, `/var`, `/usr`, and related paths.
firewall	Host or network filter that allows or denies traffic (for example UFW or nftables).
GID	Group ID; numeric identity of a group.
GRUB	GRand Unified Bootloader; common bootloader on Ubuntu and many servers.
hard link	Another directory entry for the same inode on one filesystem.
HTTP / HTTPS	Hypertext Transfer Protocol (and its TLS-secured form) used by web services.
I/O	Input/Output; disk or network data movement.
IaC	Infrastructure as Code.
initramfs	Initial RAM filesystem loaded with the kernel for early boot drivers and root mount.
inode	Filesystem object that stores metadata and points to a file's data blocks.
journald	systemd's logging service; query with `journalctl`.
kernel	Core of Linux that manages CPU, memory, devices, and system calls.
LVM	Logical Volume Manager; flexible disk volumes above physical disks/partitions.
MAC	Mandatory Access Control (SELinux or AppArmor), enforced by policy beyond DAC.
mount point	Directory where a filesystem is attached into the tree.
namespace	Kernel isolation boundary (PID, mount, network, and others) used by containers.
OCI	Open Container Initiative; standard image and runtime formats.
PAM	Pluggable Authentication Modules; login and auth stack configuration.
PATH	Environment variable listing directories searched for executable commands.
PID	Process ID; number identifying a running process.
pipe	Connects stdout of one command to stdin of the next (`|`).
process	A running instance of a programme.
PTY / TTY	Pseudo-terminal / terminal device connecting a session to a shell.
RBAC	Role-Based Access Control.
RHEL	Red Hat Enterprise Linux and compatible distributions.
root	UID 0 superuser account, or the filesystem root `/`.
RPO	Recovery Point Objective; how much data loss (in time) is acceptable.
RTO	Recovery Time Objective; how quickly service must be restored.
SELinux	Security-Enhanced Linux; label-based MAC common on RHEL-family systems.
shell	Programme that reads and runs commands (for example `bash`).
SRE	Site Reliability Engineering.

SSH	Secure Shell; encrypted remote login and file transfer.
sticky bit	Directory mode bit so users can delete only their own files (typical on `/tmp`).
sudo	Run a command as another user (usually root) under policy.
symlink	Symbolic link; a path that points to another path (may cross filesystems).
systemd	Common init system and service manager (PID 1 on most modern servers).
target	systemd unit that groups other units into a desired system state (replaces old runlevels).
TCP	Transmission Control Protocol; reliable stream transport used by SSH and HTTP.
timer	systemd unit that activates another unit on a schedule.
UEFI	Unified Extensible Firmware Interface; modern firmware replacing classic BIOS.
UFW	Uncomplicated Firewall; simple front end to nftables/iptables on Ubuntu.
UID	User ID; numeric identity of a user account.
umask	Mask that controls default permissions for newly created files.
unit	systemd configuration object (service, timer, mount, target, and so on).
UUID	Universally Unique Identifier; stable disk/filesystem id for `/etc/fstab`.
VM	Virtual machine.
zombie process	Process that has exited but still has an entry until its parent reaps it.

Index

Commands and topics with chapter page numbers.

A

aa-status 156
acl 61
apk 19
apparmor 156
apt 19 122
apt-get 122
ar- 19 26 33 40 47 54 61 68 75 82 89 96 102 108 115 122 128 136 142 148 156 164 171 177 183
chi-
tec-
ture
at 128
attributes 47
auditd 156
awk 68

B

backup 183
bash 19
blkid 96
boot 26 89

C

capacity 183
cgroups 164
chgrp 61
chmod 61 156
chown 61 156
cli 33
Code 19 26 33 40 47 54 61 68 75 82 89 96 102 108 115 122 128 136 142 148 156 164 171 177 183
Walk-
through
commands 33
cron 19 128
crontab 128
curl 108 164
cut 68

D

df 40 47 102 136 171 183
dig 108
disaster recovery 183
distributions 19
dnf 19 122
dns 108
docker 164
dpkg 122
du 26 47 164 171 183

F

fail2ban 156
fdisk 96
fhs 26
files 33
filesystem 40
findmnt 26 40 47 96 102 164
firewalld 148
free 102
fundamentals 19

G

getenforce 156
getent 54
grep 68
groupadd 54
groups 54

H

hardening 148 177
host 108 115
hostnamectl 19

I

id 19 54 61 142
identity 54
incident 171
initramfs 26
inodes 40
iostat 142
ip 108
iptables 148

J

jobs 75
journalctl 54 82 89 128 136 171
journald 136
jq 122

K

kernel 19
keys 115
kill 75

L

links 40
linux 19 26 33 40 47 54 61 68 75 82 89 96 102 108 115 122 128 136 142
ln 40
logrotate 136
ls 33 40
lsblk 96 102
lvcreate 102
lvm 102

M

mkfs 96 102
monitoring 102
mount 40 61 96
mounts 40

N

namespaces 164
navigation 33
nc 108
netstat 108
networking 108
nice 75
nohup 75 82
nsenter 164
nslookup 108

O

oci 164
overlayfs 164

P

packages 122

pam 156
parted 96
passwd 54
paths 40
performance 177
ping 108
pipelines 68
pkill 75
podman 164
production 177
ps 75 142 164
pvcreate 102

R

read 19
remote 115
rsync 115 183

S

sort 47 68
ss 108 148 171
ssh 108 115 148
sshd 19 115 148
sudo 19 47 54 61 82 89 96 102 108 115 122 128 136 142 156 171 177
systemctl 54 82 89 128 171
systemd 26 82 128
systemd-analyze 26 89

T

tee 68 108
Theory 19 26 33 40 47 54 61 68 75 82 89 96 102 108 115 122 128 136 142
timers 89 128

Y

yum 19 122

Z

zypper 19 122