Tertiary Infotech Academy Pte Ltd

UEN: 201200696W

LEARNER GUIDE

For

CompTIA Certified Linux+ Training (XK0-006)

TGS Ref No: TGS-2024048316

Conducted by

Tertiary Infotech Academy Pte Ltd

UEN: 201200696W

Version v3

DOCUMENT VERSION CONTROL RECORD

Version Number	Effective Date of Release	Summary of Included Changes	Author
v1	1 Jul 2026	Initial release — aligned to CompTIA Linux+ XK0-006 V8 exam domains and the 30 hands-on labs.	Dr. Alfred Ang
v2	18 Aug 2026	Aligned to the v2 slide deck (legacy teaching chapters merged and re-ordered onto the five exam domains); labs restructured one-folder-per-lab; exam registration and practice-exam guidance added.	Dr. Alfred Ang
v3	18 Aug 2026	Aligned to the v3 standard-format deck — condensed, objective-by-objective (XK0-006 blueprint), full lab step coverage.	Dr. Alfred Ang

TABLE OF CONTENTS

How to Use This Guide

This Learner Guide accompanies CompTIA Certified Linux+ Training (XK0-006) (TGS-2024048316). It is organised by the five CompTIA Linux+ XK0-006 V8 exam domains and contains all 30 hands-on labs, each with a goal, the commands you run, and a way to verify your work. Every lab runs on the free Killercoda Ubuntu Playground — no local install, virtual machine or credit card is required.

- Open the Killercoda Ubuntu Playground: https://killercoda.com/playgrounds/scenario/ubuntu
- Work through the labs in order; each maps to one exam sub-objective.
- Reset the playground between labs that change kernel, firewall or systemd state.
- Type the commands yourself rather than copy-pasting — muscle memory helps in the exam.
- Download the slides and this guide, and take the assessment, on the LMS: https://lms-tms.tertiaryinfotech.com/

What you need before starting:

- A laptop with a modern web browser and internet access.
- Basic comfort with a command line (the labs teach the rest).
- A Tertiary Infotech LMS account for courseware and the assessment.

Domain 1 — System Management (23% of the exam)

This domain covers exam objectives: 1.1 Explain basic Linux concepts; 1.2 Summarize Linux device management concepts and tools; 1.3 Given a scenario, manage storage in a Linux system; 1.4 Given a scenario, manage network services and configurations; 1.5 Given a scenario, manage a Linux system using common shell operations; 1.6 Given a scenario, perform backup and restore operations; 1.7 Summarize virtualization on Linux systems.

Lab 1 — Boot Process & Filesystem Hierarchy

Exam objective: 1.1 — Explain basic Linux concepts

Goal: Inspect a running Linux system's boot configuration, kernel command line, initramfs, and the standard Filesystem Hierarchy to understand how GRUB hands off to the kernel and where FHS directories live.

What you'll build: Identify the distro and kernel, inspect GRUB, the kernel cmdline, and initramfs, then tour the FHS and boot-related directories.

Key concepts: /etc/os-release, GRUB2, /proc/cmdline, initramfs, lsinitramfs, FHS, systemd targets, usrmerge

Step-by-step

Step 1 — Identify distribution and architecture

```
cat /etc/os-release
```

```
uname -a
uname -m
uname -r
hostnamectl 2>/dev/null || true
lsb_release -a
```
Reads the standard distro-ID file and kernel/architecture details.

Step 2 — Inspect GRUB configuration

```
cat /etc/default/grub 2>/dev/null || true
ls /boot 2>/dev/null
find / -name "grub.cfg" 2>/dev/null | head
```
Views the human-edited GRUB defaults and the generated grub.cfg.

Step 3 — Inspect the kernel command line

```
cat /proc/cmdline
cat /proc/version
ls /proc/sys/kernel/ | head
cat /proc/sys/kernel/hostname
```
Shows the exact arguments GRUB passed to the running kernel.

Step 4 — Explore the initramfs

```
ls /boot/initrd* 2>/dev/null || echo "no initrd image present"
lsinitramfs /boot/initrd.img-$(uname -r) 2>/dev/null | head -30
```
Lists the contents of the initramfs cpio archive without extracting it.

Step 5 — Tour the Filesystem Hierarchy (FHS)

```
for d in /bin /boot /dev /etc /home /lib /proc /sbin /sys /tmp /usr /var
/run /opt /srv; do
  printf "%-8s -> " "$d"
  ls -ld "$d" 2>/dev/null | awk '{print $1, $NF}'
done
ls /usr
ls /var
file /bin /sbin /lib 2>/dev/null
```
Walks each top-level FHS directory and reveals the usrmerge symlinks.

Step 6 — Walk through key boot-related dirs

```
ls /etc/systemd/system/ | head
systemctl list-units --type=target --no-pager 2>/dev/null | head
ls /lib/modules/$(uname -r) 2>/dev/null | head || ls /lib/modules | head
cat /proc/mounts | head
```
Shows systemd targets (successors to runlevels) and the kernel module tree.

Test it / key takeaway: You can identify the running distro/kernel/arch and explain how GRUB, the kernel cmdline, and initramfs bring up userspace.

Lab 2 — Kernel Modules & Device Management

Exam objective: 1.2 — Summarize Linux device management concepts and tools

Goal: List, inspect, load, and unload kernel modules and use the standard hardware-enumeration tools to understand modprobe vs insmod, module dependencies, and reading dmesg.

What you'll build: Install discovery tools, inspect and load/unload a module, examine module dependencies, read the kernel ring buffer, and enumerate hardware.

Key concepts: lsmod, modinfo, modprobe, insmod/rmmod, depmod, dmesg, lspci/lsusb/lscpu/lsblk, dracut vs mkinitramfs

Step-by-step

Step 1 — Install hardware inspection tools

```
apt-get update -qq
apt-get install -y pciutils usbutils hwinfo lshw dmidecode kmod >/dev/null
```

Installs discovery utilities and kmod, which provides the module tools.

Step 2 — List loaded modules and inspect one

```
lsmod | head -20
lsmod | wc -l
MOD=$(lsmod | awk 'NR==2 {print $1}')
modinfo "$MOD" | head -20
```

Lists currently loaded modules from /proc/modules and shows one module's metadata.

Step 3 — Load and unload a module

```
modprobe dummy 2>&1 || modprobe loop 2>&1 || echo "module load blocked"
lsmod | grep -E "dummy|loop" | head
ip link add dummy0 type dummy 2>/dev/null && ip link show dummy0
ip link del dummy0 2>/dev/null
modprobe -r dummy 2>&1 || true
```

Loads a module with dependency resolution then removes it with modprobe -r.

Step 4 — Module dependencies and depmod

```
cat /lib/modules/$(uname -r)/modules.dep 2>/dev/null | head -5
depmod -a 2>&1 | head || echo "depmod requires module tree"
modprobe --show-depends loop 2>&1 | head
```

Shows the modules.dep dependency map that depmod builds and modprobe uses.

Step 5 — Read the kernel ring buffer

```
dmesg | tail -20 2>/dev/null || journalctl -k --no-pager | tail -20
dmesg -T 2>/dev/null | tail -5 || true
```

Reads kernel messages where module load and hardware probe events appear.

Step 6 — Enumerate hardware

```
lscpu
lsmem 2>/dev/null | head || free -h
lspci | head
lsusb 2>/dev/null | head
lsblk
lshw -short 2>/dev/null | head -20
```
Uses each ls* tool to read a different kernel interface for hardware discovery.

Step 7 — initramfs builders overview

```
which dracut 2>/dev/null || echo "dracut on RHEL/Fedora/SUSE"
which mkinitramfs && mkinitramfs --help 2>&1 | head
which update-initramfs && update-initramfs --help 2>&1 | head
```
Contrasts Debian/Ubuntu mkinitramfs with the RHEL-family dracut.

Test it / key takeaway: You can pick the right ls* tool for a hardware question and explain insmod vs modprobe vs depmod.

Lab 3 — Storage with LVM, Partitions & Filesystems

Exam objective: 1.3 — Manage storage in a Linux system

Goal: Build a complete LVM stack on loopback-backed disks, format with ext4/xfs/btrfs, mount with hardening options, persist in fstab, and grow a filesystem online.

What you'll build: Create loopback disks, partition, build PVs/VG/LVs, make filesystems, mount with hardening options into fstab, then extend an LV and its filesystem live.

Key concepts: losetup, parted/GPT, pvcreate/vgcreate/lvcreate, mkfs.ext4/xfs/btrfs, blkid/UUID, /etc/fstab, mount hardening (nodev/nosuid/noexec), lvextend/resize2fs/xfs_growfs

Step-by-step

Step 1 — Install tooling and create loopback disks

```
apt-get install -y lvm2 xfsprogs btrfs-progs parted util-linux >/dev/null
mkdir -p /lab/disks && cd /lab/disks
for i in 1 2 3; do
  truncate -s 512M disk${i}.img
  losetup -fP disk${i}.img
done
losetup -a
```
Creates three sparse image files and attaches them as loopback block devices.

Step 2 — Partition the first loopback disk

```
LOOP1=$(losetup -j /lab/disks/disk1.img | cut -d: -f1)
parted -s "$LOOP1" mklabel gpt
parted -s "$LOOP1" mkpart primary 1MiB 100%
parted -s "$LOOP1" set 1 lvm on
partprobe "$LOOP1"
```
Writes a GPT label and one LVM-flagged partition spanning the disk.

Step 3 — Create PVs, a VG, and LVs

```
LOOP2=$(losetup -j /lab/disks/disk2.img | cut -d: -f1)
LOOP3=$(losetup -j /lab/disks/disk3.img | cut -d: -f1)
pvcreate "${LOOP1}p1" "$LOOP2" "$LOOP3"
vgcreate labvg "${LOOP1}p1" "$LOOP2" "$LOOP3"
lvcreate -L 200M -n lv_ext4 labvg
lvcreate -L 200M -n lv_xfs  labvg
lvcreate -L 200M -n lv_btrfs labvg
```

Writes LVM metadata on the devices, pools them into a VG, and carves out LVs.

Step 4 — Make filesystems

```
mkfs.ext4  -F /dev/labvg/lv_ext4
mkfs.xfs   -f /dev/labvg/lv_xfs
mkfs.btrfs -f /dev/labvg/lv_btrfs
blkid /dev/labvg/lv_ext4 /dev/labvg/lv_xfs /dev/labvg/lv_btrfs
```

Formats each LV and reads back the UUID/TYPE used by fstab.

Step 5 — Mount with hardening options and persist in fstab

```
mkdir -p /mnt/ext4 /mnt/xfs /mnt/btrfs
mount -o noatime,nodev,nosuid,noexec /dev/labvg/lv_ext4  /mnt/ext4
mount -o noatime,nodev,nosuid         /dev/labvg/lv_xfs   /mnt/xfs
mount -o noatime,compress=zstd        /dev/labvg/lv_btrfs /mnt/btrfs
UUID=$(blkid -s UUID -o value /dev/labvg/lv_ext4)
echo "UUID=$UUID  /mnt/ext4  ext4  noatime,nodev,nosuid,noexec  0 2" >>
/etc/fstab
df -hT /mnt/ext4 /mnt/xfs /mnt/btrfs
```

Mounts each filesystem with security options and adds a persistent fstab entry by UUID.

Step 6 — Grow a logical volume and its filesystem online

```
lvextend -L +100M /dev/labvg/lv_ext4
resize2fs /dev/labvg/lv_ext4
lvextend -L +100M /dev/labvg/lv_xfs
xfs_growfs /mnt/xfs
lvresize -L +50M -r /dev/labvg/lv_btrfs
df -hT /mnt/ext4 /mnt/xfs /mnt/btrfs
```

Extends each LV and grows the filesystem live; lvresize -r does both at once.

Test it / key takeaway: You can go from raw block device to a mounted, UUID-referenced, online-expandable filesystem; note XFS grows but cannot shrink.

Lab 4 — Network Configuration

Exam objective: 1.4 — Manage network services and configurations on a Linux server

Goal: Inspect interfaces, addresses, and routes with ip; walk the name-resolution stack; validate a netplan YAML; and use the standard diagnostic toolbox to read a Linux network stack top-to-bottom.

What you'll build: Install network tools, inspect interfaces/routes, walk the resolution stack, run DNS diagnostics, validate netplan, check sockets/connectivity, and capture packets/scan ports.

Key concepts: ip -br link/address/route, /etc/nsswitch.conf, /etc/resolv.conf, getent, dig/nslookup, netplan generate, ss, tcpdump/nmap

Step-by-step

Step 1 — Install tools

```
apt-get update -qq
apt-get install -y iproute2 iputils-ping dnsutils curl mtr-tiny tcpdump
traceroute nmap network-manager netplan.io >/dev/null
```

Installs the full networking toolbox plus netplan and NetworkManager.

Step 2 — Interfaces, addresses, routes

```
ip -br link
ip -br address
ip route
ip -6 route
cat /proc/net/dev | head
```

Shows compact interface state/addresses and the IPv4/IPv6 routing tables.

Step 3 — Name resolution stack

```
cat /etc/hosts
cat /etc/resolv.conf
cat /etc/nsswitch.conf | grep -E "^hosts"
getent hosts localhost
getent hosts killercoda.com
```

Walks the resolution path and tests it via the full NSS stack with getent.

Step 4 — DNS diagnostics

```
dig +short killercoda.com
dig +short MX gmail.com
dig +trace example.com 2>&1 | head -20
nslookup example.com
```

Queries DNS directly, including a root-down +trace of the resolution path.

Step 5 — NetworkManager and netplan syntax

```
nmcli device status 2>&1 | head
ls /etc/netplan/
cat > /etc/netplan/99-lab.yaml <<'EOF'
network:
  version: 2
  renderer: networkd
  ethernets:
    lo:
      addresses:
```

```
        - 127.0.0.2/32
EOF
chmod 600 /etc/netplan/99-lab.yaml
netplan generate
```
Writes a netplan YAML and validates it with netplan generate without applying.

Step 6 — Sockets and connectivity tests

```
ss -tulnp | head
ss -s
ping -c 3 127.0.0.1
curl -sI https://killercoda.com | head -5
traceroute -n -m 5 1.1.1.1
```
Lists listening sockets and runs ping/curl/traceroute connectivity checks.

Step 7 — Packet capture and port scan

```
timeout 3 tcpdump -i any -nn -c 5 'icmp or port 53' 2>&1 &
sleep 1
ping -c 2 1.1.1.1 >/dev/null
dig +short example.com >/dev/null
wait
nmap -sT -p 1-1024 127.0.0.1 | head -20
```
Captures live ICMP/DNS packets with tcpdump and TCP-connect scans localhost with nmap.

Test it / key takeaway: You can read the resolution path (nsswitch -> hosts/resolv.conf -> DNS) and pick the right diagnostic: ss, dig/getent, tcpdump, nmap.

Lab 5 — Shell Operations & Text Processing

Exam objective: 1.5 — Manage a Linux system using common shell operations

Goal: Work with shell environment variables, login vs interactive startup files, every common redirection form, and the classic text-processing pipeline to assemble non-trivial one-liners.

What you'll build: Explore env vars and PATH, paths and startup files, all redirection operators, then grep/cut/sort/uniq, sed/awk, tr/xargs, and scripted editors.

Key concepts: export/env/unset, login vs interactive shells, redirection > >> < << <<< | tee, grep/cut/sort/uniq/wc, sed/awk, tr/xargs, shell scripting

Step-by-step

Step 1 — Environment variables and PATH

```
echo "USER=$USER  HOME=$HOME  SHELL=$SHELL"
echo "PATH=$PATH"
env | head
export LABVAR="hello linux+"
env | grep LABVAR
unset LABVAR
```
Shows the exported environment and demonstrates export/unset.

Step 2 — Absolute vs relative paths and startup files

```
mkdir -p /tmp/lab5 && cd /tmp/lab5
pwd
ls ./
ls ../tmp
cat ~/.bashrc 2>/dev/null | head || echo "no .bashrc"
cat /etc/profile | head
```

Contrasts absolute/relative paths and the login vs interactive startup files.

Step 3 — Redirection operators

```
echo "first line"  > out.txt
echo "second line" >> out.txt
wc -l < out.txt
cat <<EOF >> out.txt
line three
line four
EOF
grep line <<<"line via here-string"
ls /etc | head | tee dirlist.txt | wc -l
```

Demonstrates truncate/append, stdin redirect, here-doc, here-string, and tee.

Step 4 — grep, cut, sort, uniq, wc

```
cp /etc/passwd .
wc -l passwd
grep -c bash passwd
cut -d: -f1,7 passwd | head
cut -d: -f7 passwd | sort | uniq -c | sort -rn
```

Builds a login-shell frequency report from /etc/passwd.

Step 5 — sed and awk

```
sed -n '1,5p' passwd
sed 's|/bin/bash|/bin/zsh|' passwd | head -3
awk -F: '$3 >= 1000 {print $1, $3, $7}' passwd
awk -F: 'BEGIN{n=0} /nologin/{n++} END{print n" nologin accounts"}' passwd
```

Uses sed for stream editing and awk for field-based filtering and counting.

Step 6 — head, tail, tr, xargs

```
head -3 passwd
tail -3 passwd
echo "MIXED case STRING" | tr 'a-z' 'A-Z'
echo "one two three" | tr ' ' '\n'
ls /etc | head -5 | xargs -I{} echo "found: {}"
```

Slices files, translates characters with tr, and turns stdin into args with xargs.

Step 7 — Editors: nano and vim, scripted

```
apt-get install -y vim nano >/dev/null
cat > script.sh <<'EOF'
#!/bin/bash
```

```
echo "hello from $(whoami) on $(hostname)"
EOF
chmod +x script.sh
./script.sh
vim -e -s -c 'normal Goappended line' -c 'wq' script.sh
cat script.sh
```
Writes and runs a script, then edits it non-interactively with vim Ex commands.

Test it / key takeaway: You can compose grep/awk/sed/cut/sort/uniq/tr/xargs into ad-hoc pipelines and know which startup file each shell type sources.

Lab 6 — Backup & Restore

Exam objective: 1.6 — Perform backup and restore operations for a Linux server

Goal: Create tar/cpio/dd archives with gzip/bzip2/xz compression, image and recover disks with dd and ddrescue, mirror trees with rsync, and verify integrity with sha256sum.

What you'll build: Make sample data, archive with tar and cpio, dd a disk image, recover a damaged one with ddrescue, mirror with rsync --delete, then verify checksums and read compressed files in place.

Key concepts: tar (czf/xzf/tzf), cpio, dd, ddrescue, rsync --delete, sha256sum -c, zcat/zgrep/gzip -l, gzip/bzip2/xz tradeoffs

Step-by-step

Step 1 — Install tooling and create sample data

```
apt-get install -y tar cpio gzip bzip2 xz-utils rsync gddrescue coreutils
>/dev/null
mkdir -p /lab6/src/{etc,logs,docs}
cp -r /etc/hostname /etc/os-release /lab6/src/etc/
for i in 1 2 3 4 5; do
  dd if=/dev/urandom of=/lab6/src/logs/log${i}.bin bs=1K count=20 status=none
done
echo "important document" > /lab6/src/docs/notes.txt
```
Builds a small fake tree of configs, binary logs, and a text file to archive.

Step 2 — tar with gzip, bzip2, and xz

```
cd /lab6
tar -czvf src.tar.gz  src/ 2>&1 | tail -5
tar -cjvf src.tar.bz2 src/ 2>&1 | tail -2
tar -cJvf src.tar.xz  src/ 2>&1 | tail -2
tar -tzf src.tar.gz | head -5
tar -xzvf src.tar.gz -C restore/ 2>&1 | tail -3
diff -r src restore/src && echo "restore matches source"
```
Creates gzip/bzip2/xz tarballs, lists one, extracts it, and verifies against the source.

Step 3 — cpio (the older archiver)

```
cd /lab6
find src -depth -print | cpio -ov > src.cpio 2>/dev/null
mkdir cpio-restore && cd cpio-restore
cpio -idv < ../src.cpio 2>&1 | tail -3
cd /lab6
```
Archives via find|cpio and restores it, the same format used by initramfs.

Step 4 — dd a disk image to a loopback file

```
cd /lab6
truncate -s 64M srcdisk.img
mkfs.ext4 -F srcdisk.img >/dev/null
LOOP=$(losetup -fP --show srcdisk.img)
mount "$LOOP" mnt
echo "payload" > mnt/file.txt
umount mnt; losetup -d "$LOOP"
dd if=srcdisk.img of=backup.img bs=1M status=progress
sha256sum srcdisk.img backup.img
```
Block-copies a disk image with dd and confirms a bit-for-bit match via sha256sum.

Step 5 — ddrescue from a damaged image

```
cd /lab6
cp backup.img damaged.img
dd if=/dev/zero of=damaged.img bs=1 count=1024 seek=10240 conv=notrunc
status=none
ddrescue -d -r3 damaged.img recovered.img recovered.log 2>&1 | tail -5
cat recovered.log | head
```
Simulates bad sectors then recovers what it can with ddrescue, logging failures.

Step 6 — rsync local-to-local with delete

```
mkdir -p /lab6/mirror
rsync -avz /lab6/src/ /lab6/mirror/ | tail -5
echo "new" > /lab6/src/docs/new.txt
rm /lab6/src/logs/log1.bin
rsync -avz --delete /lab6/src/ /lab6/mirror/ | tail -8
diff -r /lab6/src /lab6/mirror && echo "mirror in sync"
```
Mirrors a tree and uses --delete to prune files removed from the source.

Step 7 — Integrity and reading compressed files

```
cd /lab6
sha256sum src.tar.gz src.tar.bz2 src.tar.xz > checksums.txt
sha256sum -c checksums.txt
zcat src.tar.gz | tar -t | head -5
gzip -l src.tar.gz
```
Validates checksums and reads compressed archives without a separate decompress step.

Test it / key takeaway: You can choose tar/cpio/dd/ddrescue/rsync appropriately and know sha256sum -c and rsync trailing-slash/--delete semantics.

Lab 7 — Virtualization with QEMU/KVM & libvirt

Exam objective: 1.7 — Summarize virtualization on Linux systems

Goal: Install QEMU and libvirt, manipulate virtual disk images with qemu-img, list libvirt domains/networks with virsh, and boot a tiny guest under QEMU to discuss hypervisors, image formats, snapshots, and network modes.

What you'll build: Install QEMU/libvirt, create and inspect disk images, convert/resize them, take qcow2 snapshots, list libvirt domains/networks, boot a guest under TCG, and review the three network modes.

Key concepts: QEMU vs KVM, /dev/kvm vs TCG, qemu-img (qcow2/raw), qemu-img convert/resize, qcow2 snapshots, virsh list/net-list, bridged/NAT/host-only networks

Step-by-step

Step 1 — Install QEMU and libvirt

```
apt-get install -y qemu-system-x86 qemu-utils libvirt-clients libvirt-daemon-system bridge-utils >/dev/null
ls -l /dev/kvm 2>/dev/null && echo "KVM available" || echo "KVM NOT available - use TCG"
qemu-system-x86_64 --version | head -1
virsh --version
```

Installs the emulator and management daemon and detects KVM vs TCG.

Step 2 — Create and inspect disk images

```
mkdir -p /lab7 && cd /lab7
qemu-img create -f qcow2 disk.qcow2 1G
qemu-img create -f raw   disk.raw   256M
qemu-img info disk.qcow2
qemu-img info disk.raw
```

Creates a sparse qcow2 and a flat raw image and inspects their metadata.

Step 3 — Convert and resize images

```
cd /lab7
qemu-img convert -O raw disk.qcow2 disk-from-qcow.raw
qemu-img convert -O qcow2 disk.raw disk-from-raw.qcow2
qemu-img resize disk.qcow2 +512M
qemu-img info disk.qcow2 | grep -E "virtual size|disk size"
```

Interconverts qcow2/raw and grows the virtual size of the qcow2 image.

Step 4 — Snapshots inside a qcow2

```
cd /lab7
qemu-img snapshot -c clean disk.qcow2
qemu-img snapshot -l disk.qcow2
qemu-img snapshot -c after-config disk.qcow2
qemu-img snapshot -d clean disk.qcow2
qemu-img snapshot -l disk.qcow2
```

Creates, lists, and deletes internal qcow2 snapshots.

Step 5 — libvirt: domains and networks

```
systemctl start libvirtd 2>/dev/null || service libvirtd start 2>/dev/null ||
true
virsh list --all
virsh net-list --all
virsh net-dumpxml default 2>/dev/null | head -20
virsh capabilities 2>&1 | head -20
```

Starts libvirtd and lists defined domains and the default NAT network.

Step 6 — Boot a minimal guest under QEMU (TCG fallback)

```
cd /lab7
ACCEL="tcg"
[ -e /dev/kvm ] && ACCEL="kvm"
echo "Using accel=$ACCEL"
timeout 8 qemu-system-x86_64 \
  -machine accel=$ACCEL \
  -m 128 \
  -nographic \
  -serial mon:stdio \
  -drive file=disk.qcow2,if=virtio,format=qcow2 \
  -boot order=c 2>&1 | tail -20 || true
```

Boots QEMU headless under TCG (or KVM if available) to prove the emulator works.

Step 7 — Discuss network modes (concept)

```
cat <<'EOF'
Bridged   : guest is a peer on the LAN with its own DHCP IP.
NAT       : guest on virbr0; host masquerades outbound (libvirt default).
Host-only : isolated bridge, no upstream; host<->guest only.
EOF
ip link show type bridge 2>/dev/null
```

Summarizes bridged, NAT, and host-only guest network modes.

Test it / key takeaway: You can distinguish QEMU (emulator) from KVM (accelerator), qcow2 from raw, internal vs libvirt snapshots, and the three guest network modes.

Domain 2 — Services and User Management (20% of the exam)

This domain covers exam objectives: 2.1 Given a scenario, manage files and directories; 2.2 Given a scenario, perform local account management; 2.3 Given a scenario, manage processes and jobs; 2.4 Given a scenario, configure and manage software; 2.5 Given a scenario, manage Linux using systemd; 2.6 Given a scenario, manage applications in a container.

Lab 8 — File & Directory Management

Exam objective: 2.1 — Manage files and directories on a Linux system

Goal: Practice core file and directory operations: navigating, creating, copying, searching, linking, and inspecting special device files with standard Linux utilities.

What you'll build: Create a lab tree, then copy, find, link, diff, and inspect files and /dev device nodes.

Key concepts: ls -la / stat / file, find predicates (-name, -mtime, -size), locate & updatedb, lsof, diff & sdiff, hard vs symbolic links, block/character device files

Step-by-step

Step 1 — Navigate and list

```
pwd
cd /tmp
mkdir -p lab08/{src,dst,archive}
cd lab08
ls -la
```

pwd shows the cwd; ls -la lists hidden files with full metadata.

Step 2 — Create, copy, move, touch

```
touch src/file1.txt src/file2.log
echo "hello" > src/file1.txt
cp -a src/file1.txt dst/
mv src/file2.log archive/
stat dst/file1.txt
file dst/file1.txt
```

cp -a preserves attributes; stat shows inode details and file identifies content type.

Step 3 — Find files by name, time, and size

```
dd if=/dev/zero of=src/big.bin bs=1M count=5
find /tmp/lab08 -name "*.txt"
find /tmp/lab08 -mtime -1
find /tmp/lab08 -size +1M
find /tmp/lab08 -name "*.bin" -exec ls -lh {} \;
```

find walks the tree with predicates; -exec runs a command per match.

Step 4 — locate and lsof

```
apt update -y && apt install -y mlocate lsof
updatedb
locate file1.txt | head
sleep 300 &
lsof -p $! | head
```

locate queries a prebuilt DB; lsof lists open files held by a process.

Step 5 — diff and sdiff

```
echo -e "alpha\nbravo\ncharlie" > a.txt
echo -e "alpha\nbravo2\ncharlie\ndelta" > b.txt
diff a.txt b.txt
```

```
sdiff a.txt b.txt
```
diff shows line differences; sdiff gives a side-by-side view with markers.

Step 6 — Hard links vs symbolic links

```
echo "original" > target.txt
ln target.txt hardlink.txt
ln -s target.txt symlink.txt
ls -li target.txt hardlink.txt symlink.txt
rm target.txt
cat hardlink.txt
cat symlink.txt || echo "symlink broken"
```
Hard links share an inode and survive deletion; symlinks break when the target is gone.

Step 7 — Device files in /dev

```
ls -l /dev/sda /dev/null /dev/tty1 2>/dev/null
echo "discarded" > /dev/null
head -c 16 /dev/urandom | xxd
```
The ls -l first column marks block (b) vs character (c) devices; /dev/null discards writes.

Test it / key takeaway: ls -li confirms hardlink shares the target's inode while the symlink breaks after the target is removed.

Lab 9 — Local Account & Group Management

Exam objective: 2.2 — Perform local account management in a Linux environment

Goal: Create and manage local users and groups, set passwords and aging policies, change shells, and distinguish user, system, and service accounts.

What you'll build: Create users/groups, set passwords and aging, inspect identity databases, then remove the accounts.

Key concepts: /etc/passwd, /etc/shadow, /etc/group, /etc/skel, useradd / adduser / usermod -aG, chpasswd, passwd -S, chage, chsh, getent, id, UID>=1000 user vs UID<1000 system vs service (-r), userdel -r / groupdel

Step-by-step

Step 1 — Inspect account databases

```
head -5 /etc/passwd
head -5 /etc/shadow
head -5 /etc/group
ls /etc/skel -la
head -20 /etc/profile
```
passwd holds metadata, shadow holds hashed passwords, and skel templates new home dirs.

Step 2 — Create users and groups

```
groupadd developers
```

```
useradd -m -s /bin/bash -G developers alice
adduser --disabled-password --gecos "" bob
usermod -aG developers bob
id alice
groups bob
```
useradd is low-level, adduser is the Debian wrapper; usermod -aG appends groups without dropping existing ones.

Step 3 — Passwords and aging

```
echo "alice:Passw0rd!" | chpasswd
passwd -S alice
chage -l alice
chage -M 90 -W 7 alice
chage -l alice
```
chage sets aging policy: -M max age in days, -W the warning window.

Step 4 — Change shell and identify accounts

```
chsh -s /bin/sh bob
getent passwd alice bob
whoami
id
sudo -u alice id
```
chsh updates the login shell; sudo -u shows the effective UID/GID when running as another user.

Step 5 — User vs system vs service accounts

```
awk -F: '$3 >= 1000 && $3 < 65534 {print "USER  ", $1, $3}' /etc/passwd
awk -F: '$3 < 1000 {print "SYSTEM", $1, $3}' /etc/passwd | head
getent passwd www-data
useradd -r -s /usr/sbin/nologin svc_app
getent passwd svc_app
```
Users are UID>=1000, system accounts UID<1000, service accounts use -r plus nologin.

Step 6 — Login history and active sessions

```
last | head
lastlog | head
who
w
```
last reads wtmp history, lastlog shows last login per user, w/who show current sessions.

Step 7 — Remove users and groups

```
userdel -r bob
userdel -r svc_app
groupdel developers
getent passwd bob || echo "bob removed"
```
userdel -r removes the account plus home and mail spool; remove users before their primary group.

Test it / key takeaway: getent passwd bob returns nothing after userdel -r, confirming the account and home directory are gone.

Lab 10 — Processes, Jobs & Scheduling

Exam objective: 2.3 — Manage processes and jobs in a Linux environment

Goal: Inspect processes, manipulate shell jobs, adjust priorities, send signals, and schedule tasks with at, cron, and anacron.

What you'll build: Examine processes via ps/proc/strace, control background jobs, renice, signal, and schedule jobs.

Key concepts: ps auxf / pstree / pidstat / mpstat, /proc/<PID> and lsof -p, process states (R,S,D,Z,T) & strace, jobs: &, Ctrl+Z, bg, fg, nohup, disown, nice / renice priority, kill / pkill / killall signals, at, cron, anacron

Step-by-step

Step 1 — Inspect running processes

```
ps auxf | head -20
pstree -p | head -20
apt update -y && apt install -y htop sysstat lsof strace at
pidstat 1 3
mpstat 1 2
```

ps auxf shows a process forest; pidstat/mpstat sample per-process and per-CPU stats.

Step 2 — Explore /proc and lsof

```
sleep 600 &
SPID=$!
ls /proc/$SPID/
cat /proc/$SPID/status | head
cat /proc/$SPID/cmdline; echo
lsof -p $SPID | head
```

Each process has a /proc/<PID> dir; lsof -p lists all its open file descriptors.

Step 3 — Process states and strace

```
ps -o pid,stat,cmd -p $SPID
strace -p $SPID -e trace=nanosleep -o /tmp/strace.out &
sleep 2
kill %2 2>/dev/null
head /tmp/strace.out
```

STAT codes show process state; strace -p attaches to a live PID to trace system calls.

Step 4 — Background, foreground, jobs

```
sleep 300 &
sleep 400 &
jobs
```

```
fg %1
# Ctrl+Z to suspend, then:
bg %1
jobs
nohup sleep 500 > /tmp/nohup.out 2>&1 &
disown
jobs
```
& backgrounds, Ctrl+Z suspends, bg/fg resume, and nohup+disown detach a job to survive logout.

Step 5 — Priority with nice and renice

```
nice -n 10 sleep 200 &
NPID=$!
ps -o pid,ni,cmd -p $NPID
renice -n 5 -p $NPID
ps -o pid,ni,cmd -p $NPID
```
Nice ranges -20 (highest) to 19 (lowest); only root can raise priority.

Step 6 — Signals: kill, pkill, killall

```
sleep 1000 &
TPID=$!
kill -HUP $TPID 2>/dev/null; echo "HUP sent"
sleep 1000 &
kill -15 $!
sleep 1000 &
kill -9 $!
sleep 1000 & sleep 1000 &
pkill -f "sleep 1000"
killall sleep 2>/dev/null
```
-15 (TERM) asks graceful exit, -9 (KILL) forces it, -1 (HUP) reloads; pkill matches pattern, killall by name.

Step 7 — Scheduling with at and cron

```
service atd start 2>/dev/null || systemctl start atd
echo "date > /tmp/at-fired.txt" | at now + 1 minute
atq
( crontab -l 2>/dev/null; echo "* * * * * date >> /tmp/cron.log" ) | crontab -
crontab -l
ls /etc/anacrontab && head /etc/anacrontab
```
at schedules a one-shot job, cron runs recurring five-field jobs, anacron catches missed jobs.

Test it / key takeaway: atq lists the queued at job and crontab -l shows the recurring entry, confirming both schedulers accepted the tasks.

Lab 11 — Software & Package Management

Exam objective: 2.4 — Configure and manage software in a Linux environment

Goal: Manage Debian packages with apt/dpkg, add a signed third-party repo, run dnf/rpm in a container, and use language and sandboxed package managers.

What you'll build: Install/remove apt packages, add a GPG-signed repo, demo dnf/rpm, then pip/npm/cargo and snap/flatpak.

Key concepts: apt update/install/remove/purge/autoremove, apt-cache policy, dpkg -l / -L, signed-by GPG keyrings, dnf / rpm -q / rpm -V, pip, npm -g, cargo, update-alternatives, snap & flatpak sandboxes

Step-by-step

Step 1 — apt basics

```
apt update
apt install -y tree jq
apt-cache policy jq
dpkg -l | grep -E "tree|jq"
dpkg -L tree | head
```

apt manages packages/deps; dpkg -l lists installed packages and dpkg -L lists a package's files.

Step 2 — Add a third-party repo with GPG key

```
apt install -y curl gnupg lsb-release
install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://apt.releases.hashicorp.com/gpg | gpg --dearmor -o
/etc/apt/keyrings/hashicorp.gpg
echo "deb [signed-by=/etc/apt/keyrings/hashicorp.gpg]
https://apt.releases.hashicorp.com $(lsb_release -cs) main" >
/etc/apt/sources.list.d/hashicorp.list
apt update
apt-cache policy terraform | head
```

Modern Debian/Ubuntu uses per-repo signed-by keyrings instead of deprecated apt-key.

Step 3 — Remove packages

```
apt remove -y tree
apt purge -y jq
apt autoremove -y
```

remove keeps config, purge deletes it too, autoremove cleans orphaned dependencies.

Step 4 — RPM world inside a container

```
apt install -y docker.io
systemctl start docker
docker run --rm rockylinux:9 bash -c "dnf -y install nano > /dev/null && rpm -q
nano && rpm -V nano || true && rpm -qa | head"
```

dnf is the Red Hat package manager; rpm -q queries install status and rpm -V verifies integrity.

Step 5 — Language-specific package managers

```
apt install -y python3-pip nodejs npm cargo
pip3 install --user --break-system-packages requests
```

```
python3 -c "import requests; print(requests.__version__)"
npm install -g cowsay
cowsay "Linux+"
cargo --version
```
pip installs Python libs, npm -g installs Node CLIs globally, cargo is Rust's build/package tool.

Step 6 — update-alternatives

```
update-alternatives --display editor
update-alternatives --list editor
```
update-alternatives manages symlinks for commands with multiple providers (editor, java, etc.).

Step 7 — Sandboxed package formats

```
which snap && snap list 2>/dev/null || echo "snap not preinstalled on this
image"
apt install -y flatpak
flatpak --version
flatpak remotes
```
Snap and Flatpak ship apps with bundled dependencies in confined sandboxes.

Test it / key takeaway: apt-cache policy terraform shows a candidate from the signed HashiCorp repo, confirming the GPG-verified source is trusted.

Lab 12 — Service Management with systemd

Exam objective: 2.5 — Manage Linux using systemd

Goal: Inspect and control systemd units, author custom service/timer/mount units, analyze boot performance, and configure host/time/DNS via systemd helpers.

What you'll build: Manage nginx, write a service + timer + mount unit, analyze boot time, and set hostname/timezone/DNS.

Key concepts: systemctl list-units / list-unit-files / --failed, start/stop/restart/reload, enable/disable, mask/unmask, custom .service (Type=oneshot) + daemon-reload, .timer units (OnUnitActiveSec), .mount units, systemd-analyze blame / critical-chain, hostnamectl / timedatectl / resolvectl, journalctl -u

Step-by-step

Step 1 — List and inspect units

```
systemctl list-units --type=service | head -20
systemctl list-unit-files --type=service | head -20
systemctl --failed
```
list-units shows loaded units; list-unit-files shows all installed units and their enabled state.

Step 2 — Control nginx

```
apt update -y && apt install -y nginx
systemctl status nginx --no-pager
```

```
systemctl stop nginx; systemctl start nginx; systemctl restart nginx; systemctl
reload nginx
systemctl disable nginx; systemctl enable nginx
systemctl mask nginx && systemctl start nginx 2>&1 | head
systemctl unmask nginx
```
enable/disable toggles autostart, reload re-reads config live, and mask makes a unit unstartable.

Step 3 — Author a custom service

```
cat > /usr/local/bin/lab-hello.sh <<'EOF'
#!/usr/bin/env bash
echo "lab-hello fired at $(date)" >> /var/log/lab-hello.log
EOF
chmod +x /usr/local/bin/lab-hello.sh
cat > /etc/systemd/system/lab-hello.service <<'EOF'
[Unit]
Description=Lab Hello one-shot service
[Service]
Type=oneshot
ExecStart=/usr/local/bin/lab-hello.sh
EOF
systemctl daemon-reload
systemctl start lab-hello.service
cat /var/log/lab-hello.log
```
daemon-reload re-parses unit files; Type=oneshot suits short scripts that exit when done.

Step 4 — Add a timer

```
cat > /etc/systemd/system/lab-hello.timer <<'EOF'
[Unit]
Description=Run lab-hello every minute
[Timer]
OnBootSec=30s
OnUnitActiveSec=1min
Unit=lab-hello.service
[Install]
WantedBy=timers.target
EOF
systemctl daemon-reload
systemctl enable --now lab-hello.timer
systemctl list-timers --no-pager | head
```
Timer units are the systemd-native replacement for cron, with journal integration.

Step 5 — Mount unit example

```
mkdir -p /mnt/labdata
cat > /etc/systemd/system/mnt-labdata.mount <<'EOF'
[Unit]
Description=Tmpfs at /mnt/labdata
[Mount]
What=tmpfs
Where=/mnt/labdata
Type=tmpfs
Options=size=16M
```

```
[Install]
WantedBy=multi-user.target
EOF
systemctl daemon-reload
systemctl start mnt-labdata.mount
mount | grep labdata
```
Mount unit filenames must match the path with / replaced by - (/mnt/labdata -> mnt-labdata.mount).

Step 6 — Boot performance

```
systemd-analyze
systemd-analyze blame | head
systemd-analyze critical-chain | head -20
```
blame ranks units by start time; critical-chain traces the longest dependency delay path.

Step 7 — Host, time, and DNS

```
hostnamectl
hostnamectl set-hostname linuxplus-lab
timedatectl
timedatectl set-timezone Asia/Singapore
resolvectl status | head -20
```
These systemd front ends manage /etc/hostname, the clock/timezone, and the stub DNS resolver.

Test it / key takeaway: systemctl list-timers shows lab-hello.timer scheduled and journalctl -u lab-hello records each firing, confirming the custom units work.

Lab 13 — Containers with Docker & Podman

Exam objective: 2.6 — Manage applications in a container on a Linux server

Goal: Install Docker and Podman, pull/run images, manage volumes and networks, build an image from a Dockerfile, and contrast rootful vs rootless engines.

What you'll build: Run and inspect nginx containers, bind-mount volumes, build an image, wire a user-defined network, and try rootless Podman.

Key concepts: docker.io vs podman (daemonless/rootless), docker pull/run -d -p --name, docker ps, logs / exec -it / inspect / stats, bind-mount volumes (-v host:container:ro), Dockerfile: FROM/RUN/COPY/USER/ENTRYPOINT/CMD, docker network create (built-in DNS), rootless Podman & --privileged risks

Step-by-step

Step 1 — Install Docker and Podman

```
apt update -y
apt install -y docker.io podman
systemctl start docker
```

```
docker --version
podman --version
```
docker.io is Ubuntu's Docker Engine; podman is a daemonless, rootless-friendly alternative.

Step 2 — Pull and run a container

```
docker pull nginx
docker images
docker run -d -p 8080:80 --name web nginx
docker ps
curl -s -o /dev/null -w "%{http_code}\n" http://localhost:8080
```
-d detaches, -p publishes a port, --name assigns a friendly identifier.

Step 3 — Inspect, exec, logs

```
docker logs web | head
docker exec -it web bash -c "nginx -v; ls /usr/share/nginx/html"
docker inspect web | head -40
docker stats --no-stream web
```
exec runs a new process inside a running container; inspect returns its full JSON config.

Step 4 — Volumes for persistent data

```
mkdir -p /data
echo "<h1>Lab 13</h1>" > /data/index.html
docker run -d -p 8081:80 -v /data:/usr/share/nginx/html:ro --name web2 nginx
curl -s http://localhost:8081
```
Bind mounts expose host paths in the container; :ro makes the mount read-only.

Step 5 — Build a tiny image

```
mkdir -p /tmp/hello && cd /tmp/hello
cat > app.sh <<'EOF'
#!/bin/sh
echo "Hello from $(hostname) as $(id -un)"
EOF
chmod +x app.sh
cat > Dockerfile <<'EOF'
FROM alpine:3.20
RUN adduser -D appuser
COPY app.sh /usr/local/bin/app.sh
USER appuser
ENTRYPOINT ["/usr/local/bin/app.sh"]
CMD []
EOF
docker build -t hello:1 .
docker run --rm hello:1
```
FROM sets the base, USER drops root, ENTRYPOINT is the fixed command, CMD its default args.

Step 6 — Networks and port mapping

```
docker network create labnet
docker run -d --network labnet --name api nginx
```

```
docker run --rm --network labnet alpine sh -c "apk add --no-cache curl
>/dev/null && curl -s -o /dev/null -w '%{http_code}\n' http://api"
docker network inspect labnet | head -30
```
User-defined bridge networks provide built-in DNS so containers reach each other by name.

Step 7 — Rootless Podman and privileged note

```
podman run --rm alpine echo "hello from podman"
podman ps -a
echo "--privileged disables most isolation: device access, capabilities,
AppArmor."
echo "Prefer unprivileged + specific --cap-add over --privileged."
```
Podman runs daemonless and, as a non-root user, in a rootless user namespace for defense in depth.

Test it / key takeaway: curl against the published ports returns 200 and the alpine container reaches http://api by name, confirming containers, volumes, and the user-defined network work.

Domain 3 — Security (18% of the exam)

This domain covers exam objectives: 3.1 Summarize authorization, authentication, and accounting methods; 3.2 Given a scenario, configure and implement firewalls; 3.3 Given a scenario, apply OS hardening techniques; 3.4 Explain account hardening techniques and best practices; 3.5 Explain cryptographic concepts and technologies; 3.6 Explain the importance of compliance and audit procedures.

Lab 14 — AAA: sudo, PAM, and Polkit

Exam objective: 3.1 — Summarize authorization, authentication, and accounting methods

Goal: Configure sudo authorization, inspect PAM authentication stacks, and enable auditd accounting to record security events. Finish with a conceptual look at centralized identity via LDAP, Kerberos, and SSSD.

What you'll build: Install sudo, grant a user privileges, inspect PAM, read auth logs, and enable auditd rules.

Key concepts: sudo / visudo / sudoers.d, PAM stacks (auth, account, session), auditctl / ausearch, journalctl / rsyslog / logrotate, pkexec (Polkit), SSSD / LDAP / Kerberos

Step-by-step

Step 1 — Install sudo and create a user

```
apt update && apt install -y sudo
useradd -m -s /bin/bash alice
echo "alice:Passw0rd!" | chpasswd
id alice
```
Creates user alice with a home directory and bash shell.

Step 2 — Grant sudo via group and drop-in file

```
usermod -aG sudo alice
EDITOR=tee visudo -f /etc/sudoers.d/lab <<'EOF'
alice ALL=(ALL) NOPASSWD: /usr/bin/systemctl status *, /usr/bin/apt update
EOF
chmod 440 /etc/sudoers.d/lab
visudo -c
sudo -l -U alice
```

A validated sudoers drop-in grants two NOPASSWD commands without editing the master file.

Step 3 — Switch identity with sudo and su

```
su - alice -c 'sudo -n systemctl status ssh || true'
su - alice -c 'whoami; sudo -i whoami'
```

sudo -i opens a root login shell; su - becomes another user with a full login environment.

Step 4 — Inspect PAM stacks

```
ls /etc/pam.d/
grep -vE '^\s*#|^$' /etc/pam.d/sshd
grep -vE '^\s*#|^$' /etc/pam.d/common-auth
apt show sssd 2>/dev/null | head -n 20
```

PAM chains auth/account/password/session modules; SSSD integrates Linux with LDAP/Kerberos/AD.

Step 5 — Read auth logs via journald and rsyslog

```
apt install -y rsyslog
systemctl enable --now rsyslog
journalctl _COMM=sshd --no-pager | tail -n 20 || true
tail -n 20 /var/log/auth.log 2>/dev/null || journalctl -u ssh --no-pager | tail
logrotate -d /etc/logrotate.conf 2>&1 | tail -n 20
```

journalctl filters by process name; logrotate -d dry-runs rotation without changing files.

Step 6 — Enable auditd and watch /etc/passwd

```
apt install -y auditd audispd-plugins
systemctl enable --now auditd
auditctl -w /etc/passwd -p wa -k passwd_changes
auditctl -l
echo "# test comment" >> /etc/passwd
ausearch -k passwd_changes | tail -n 20
```

The watch records write and attribute changes to /etc/passwd, queried by key with ausearch.

Step 7 — Persistent audit rules and Polkit note

```
cat >/etc/audit/rules.d/lab.rules <<'EOF'
-w /etc/passwd -p wa -k passwd_changes
-w /etc/shadow -p wa -k shadow_changes
-w /etc/sudoers -p wa -k sudoers_changes
EOF
augenrules --load
auditctl -l
```

```
which pkexec && pkexec --version
```
Rules in /etc/audit/rules.d/ survive reboot; pkexec is the Polkit equivalent of sudo.

Test it / key takeaway: ausearch -k passwd_changes returns the recorded write event, confirming accounting is active.

Lab 15 — Firewalls: ufw, firewalld, iptables, nftables, ipset

Exam objective: 3.2 — Configure and implement firewalls on a Linux system

Goal: Configure firewalls at every layer of the Linux netfilter stack, from the ufw wrapper down to iptables and nftables, and understand stateful inspection. Explore zones, rich rules, ipset, and SNAT/DNAT.

What you'll build: Configure ufw, firewalld zones, nftables, ipset, NAT rules, and a stateful conntrack pattern.

Key concepts: ufw allow/deny/enable, firewalld zones and rich rules, iptables / nftables chains, ipset hash:ip, SNAT / DNAT / MASQUERADE, conntrack states (NEW/ESTABLISHED/RELATED)

Step-by-step

Step 1 — Install and configure ufw

```
apt update && apt install -y ufw
ufw allow 22/tcp
ufw deny 23/tcp
yes | ufw enable
ufw status verbose
```
ufw is a friendly netfilter wrapper; verbose status shows defaults, logging, and rule order.

Step 2 — Explore iptables backing ufw

```
iptables -L -n -v
iptables -t nat -L -n -v
```
ufw writes rules into the filter table ufw-* chains; the nat table stays empty until NAT rules are added.

Step 3 — firewalld zones and rich rules in Rocky

```
apt install -y docker.io
systemctl start docker
docker run --rm --privileged -it rockylinux:9 bash -c '
  dnf install -y firewalld iproute && \
  firewall-offline-cmd --zone=public --add-service=https && \
  firewall-offline-cmd --zone=public --add-rich-rule="rule family=ipv4 source
address=10.0.0.0/24 service name=ssh accept" && \
  firewall-offline-cmd --list-all --zone=public
'
```
firewall-offline-cmd configures zones without the daemon; rich rules add source-based ACLs.

Step 4 — nftables: table, chain, and a rule

```
apt install -y nftables
nft add table inet lab
nft 'add chain inet lab input { type filter hook input priority 0 ; policy
accept ; }'
nft add rule inet lab input tcp dport 23 drop
nft list ruleset
```

nftables is the modern netfilter front end; the inet family covers both IPv4 and IPv6.

Step 5 — ipset for high-performance IP lists

```
apt install -y ipset
ipset create badguys hash:ip
ipset add badguys 192.0.2.10
ipset add badguys 198.51.100.5
ipset list badguys
iptables -I INPUT -m set --match-set badguys src -j DROP
```

ipset stores many addresses in an O(1) hash referenced by a single iptables rule.

Step 6 — SNAT vs DNAT and masquerade

```
sysctl -w net.ipv4.ip_forward=1
iptables -t nat -A POSTROUTING -o eth0 -s 10.10.0.0/24 -j MASQUERADE
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8080 -j DNAT --to-
destination 10.10.0.50:80
iptables -t nat -L -n -v
```

SNAT/MASQUERADE rewrites source addresses outbound; DNAT rewrites destinations for port forwarding.

Step 7 — Stateful inspection demo

```
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -m conntrack --ctstate NEW -j ACCEPT
iptables -L INPUT -n -v
```

The canonical stateful pattern: allow return traffic, then permit only new SSH connections.

Test it / key takeaway: ufw status and nft list ruleset show the loaded rules; conntrack states drive stateful acceptance.

Lab 16 — OS Hardening: Permissions, SELinux, SSH, Fail2ban

Exam objective: 3.3 — Apply operating system hardening techniques on a Linux system

Goal: Apply layered OS hardening from file permissions and immutability through SELinux labeling, then harden OpenSSH and add brute-force protection. Inventory SUID binaries and legacy cleartext services.

What you'll build: Set special permission bits, ACLs, and immutability, demo SELinux, harden SSH, and run fail2ban.

Key concepts: chmod setuid/setgid/sticky bit, umask / chattr immutability, POSIX ACLs (setfacl/getfacl), SELinux chcon/restorecon/audit2allow, OpenSSH hardening (PermitRootLogin), fail2ban, SUID hunting (find -perm -4000)

Step-by-step

Step 1 — chmod, chown, special bits

```
useradd -m alice 2>/dev/null
mkdir -p /srv/lab && cd /srv/lab
cat >run.sh <<'EOF'
#!/bin/bash
id
EOF
chmod 750 run.sh
chmod u+s run.sh
chmod g+s .
chmod +t /tmp
chown root:alice run.sh
ls -l run.sh /tmp -d .
```

setuid runs a file as its owner, setgid on a dir inherits the group, sticky bit protects /tmp.

Step 2 — umask, chattr, lsattr

```
umask 027
touch /srv/lab/newfile && ls -l /srv/lab/newfile
chattr +i /etc/resolv.conf
lsattr /etc/resolv.conf
chattr -i /etc/resolv.conf
```

umask 027 yields 640 files; chattr +i makes a file immutable even to root until cleared.

Step 3 — POSIX ACLs

```
apt install -y acl
touch /tmp/file
setfacl -m u:alice:r /tmp/file
getfacl /tmp/file
ls -l /tmp/file
```

ACLs extend owner/group/other; the trailing + in ls -l flags extra ACL entries.

Step 4 — SELinux demo inside Rocky

```
apt install -y docker.io && systemctl start docker
docker run --rm -it rockylinux:9 bash -c '
  dnf install -y policycoreutils selinux-policy-targeted policycoreutils-
python-utils setools-console && \
  getenforce || echo "SELinux disabled in container"; \
  ls -Z /etc/passwd; \
  echo "Workflow: chcon -t httpd_sys_content_t /web; semanage fcontext -a ...;
restorecon -Rv /web"; \
  echo "Denials: ausearch -m AVC | audit2allow -M mypol"
'
```

Ubuntu uses AppArmor, so SELinux labeling and audit2allow are shown in a Rocky container.

Step 5 — Harden OpenSSH

```
apt install -y openssh-server
sed -i 's/^#\?PermitRootLogin.*/PermitRootLogin no/' /etc/ssh/sshd_config
sed -i 's/^#\?PasswordAuthentication.*/PasswordAuthentication no/'
/etc/ssh/sshd_config
sshd -t && systemctl restart ssh
ssh-keygen -t ed25519 -N '' -f /root/.ssh/id_ed25519
mkdir -p /home/alice/.ssh
cat /root/.ssh/id_ed25519.pub >> /home/alice/.ssh/authorized_keys
chmod 700 /home/alice/.ssh && chmod 600 /home/alice/.ssh/authorized_keys
```

Disabling root login and password auth forces key-based access; sshd -t validates before restart.

Step 6 — fail2ban

```
apt install -y fail2ban
cat >/etc/fail2ban/jail.local <<'EOF'
[sshd]
enabled = true
maxretry = 5
bantime = 600
EOF
systemctl enable --now fail2ban
fail2ban-client status
fail2ban-client status sshd
```

fail2ban tails auth logs and bans IPs exceeding maxretry failures within findtime.

Step 7 — Hunt SUID binaries and review legacy services

```
find / -perm -4000 -type f 2>/dev/null | head -n 20
dpkg -l | grep -Ei 'telnet|tftp|vsftpd|rsh' || echo "Clean: no legacy daemons
installed"
systemctl list-unit-files | grep -Ei 'telnet|tftp|ftp' || true
```

SUID binaries are escalation targets; cleartext telnet/FTP/TFTP should be replaced with SSH/SFTP/HTTPS.

Test it / key takeaway: fail2ban-client status sshd shows the active jail and sshd -t confirms the hardened config is valid.

Lab 17 — Account Hardening

Exam objective: 3.4 — Explain account hardening techniques and best practices

Goal: Harden user accounts through password quality and aging policies, brute-force lockouts, and restricted or non-interactive shells. Explore TOTP MFA and Have I Been Pwned k-anonymity lookups.

What you'll build: Tune login.defs, enforce pwquality, set aging with chage, add pam_faillock, and lock down shells.

Key concepts: /etc/login.defs (PASS_MAX_DAYS), pam_pwquality / pwscore, chage aging, passwd -l/-u, pam_faillock lockout, nologin / rbash restricted shells, Google Authenticator TOTP MFA, HIBP k-anonymity API

Step-by-step

Step 1 — login.defs and a test user

```
apt update && apt install -y libpam-pwquality
useradd -m -s /bin/bash alice
echo "alice:Passw0rd!" | chpasswd
sed -i 's/^PASS_MAX_DAYS.*/PASS_MAX_DAYS   90/' /etc/login.defs
sed -i 's/^PASS_MIN_DAYS.*/PASS_MIN_DAYS   1/' /etc/login.defs
grep -E '^PASS_' /etc/login.defs
```

/etc/login.defs sets system-wide defaults applied to newly created accounts.

Step 2 — Password quality with pwquality

```
cat >/etc/security/pwquality.conf <<'EOF'
minlen = 14
dcredit = -1
ucredit = -1
lcredit = -1
ocredit = -1
retry = 3
EOF
pwscore <<<"short"
pwscore <<<"Tr0ub4dor&3LongerPhrase!"
```

Negative credits require each character class; pwscore rates a candidate password 0-100.

Step 3 — Password aging and lock/unlock

```
chage -M 90 -m 1 -W 7 alice
chage -l alice
passwd -l alice
passwd -S alice
passwd -u alice
passwd -S alice
```

chage -l lists aging; passwd -l locks by prefixing ! to the hash, -u unlocks.

Step 4 — pam_faillock lockout

```
cat >/etc/security/faillock.conf <<'EOF'
deny = 5
unlock_time = 900
fail_interval = 900
EOF
grep -l pam_faillock /etc/pam.d/* 2>/dev/null || echo "Add pam_faillock
preauth/authfail lines to common-auth"
faillock --user alice
```

pam_faillock (successor to pam_tally2) locks after 5 failures; --reset clears the counter.

Step 5 — Service accounts with nologin and rbash

```
useradd -r -s /usr/sbin/nologin svcacct
grep svcacct /etc/passwd
ln -sf /bin/bash /usr/local/bin/rbash
useradd -m -s /usr/local/bin/rbash boxed
echo 'PATH=/usr/local/rbin' >> /home/boxed/.bashrc
mkdir -p /usr/local/rbin && ln -sf /bin/ls /usr/local/rbin/ls
su - boxed -c 'cd /tmp 2>&1 || echo "rbash blocks cd"'
```
nologin blocks interactive daemon logins; rbash prevents cd, PATH reassignment, and redirection.

Step 6 — MFA with Google Authenticator (walkthrough)

```
apt install -y libpam-google-authenticator
echo "Run as the target user, not root:"
echo "  google-authenticator -t -d -f -r 3 -R 30 -W"
echo "Then add 'auth required pam_google_authenticator.so' to /etc/pam.d/sshd"
echo "and set ChallengeResponseAuthentication yes in /etc/ssh/sshd_config."
```
Interactive setup prints a QR code; the TOTP secret is stored in ~/.google_authenticator.

Step 7 — Pwned Passwords k-anonymity lookup

```
apt install -y curl
PW='password123'
HASH=$(printf "%s" "$PW" | sha1sum | awk '{print toupper($1)}')
PREFIX=${HASH:0:5}; SUFFIX=${HASH:5}
curl -s "https://api.pwnedpasswords.com/range/$PREFIX" | grep -i "^$SUFFIX" \
  && echo "BREACHED" || echo "Not found"
```
Only the first 5 SHA-1 chars are sent; the client filters returned suffixes locally.

Test it / key takeaway: chage -l alice and passwd -S alice confirm aging and lock state; the HIBP query flags breached passwords.

Lab 18 — Cryptography: GPG, LUKS, OpenSSL, WireGuard

Exam objective: 3.5 — Explain cryptographic concepts and technologies in a Linux environment

Goal: Practice core Linux cryptography: GPG key generation and signing, LUKS2 disk encryption, OpenSSL certificates, hashing and HMAC, and WireGuard keys. Learn to identify weak TLS versions and ciphers.

What you'll build: Generate GPG keys, encrypt/sign, build a LUKS2 volume, create an OpenSSL cert, and make WireGuard keys.

Key concepts: GPG batch key generation, encrypt/sign, LUKS2 (cryptsetup, loopback), OpenSSL genrsa / req -x509, hashing vs HMAC, WireGuard Curve25519 keys, TLS version/cipher inspection (s_client)

Step-by-step

Step 1 — GPG key generation in batch mode

```
apt update && apt install -y gnupg
mkdir -p /root/.gnupg && chmod 700 /root/.gnupg
cat >/tmp/gpgparams <<'EOF'
%no-protection
Key-Type: RSA
Key-Length: 4096
Name-Real: Lab
Name-Email: lab@example.com
Expire-Date: 0
%commit
EOF
gpg --batch --generate-key /tmp/gpgparams
gpg --list-keys
```
%no-protection skips passphrase prompts for unattended use; Expire-Date: 0 is non-expiring.

Step 2 — Encrypt, decrypt, and detached-sign

```
echo "top secret payload" > /tmp/msg.txt
gpg --yes --trust-model always -r lab@example.com -e /tmp/msg.txt
gpg --yes -d /tmp/msg.txt.gpg
gpg --yes --detach-sign /tmp/msg.txt
gpg --verify /tmp/msg.txt.sig /tmp/msg.txt
```
A detached .sig verifies integrity without altering the original file.

Step 3 — LUKS2 on a loopback device

```
apt install -y cryptsetup
dd if=/dev/zero of=/tmp/luks.img bs=1M count=64
LOOP=$(losetup -f --show /tmp/luks.img)
echo -n "Passw0rd!" | cryptsetup luksFormat --type luks2 "$LOOP" -
echo -n "Passw0rd!" | cryptsetup open "$LOOP" labvol -
mkfs.ext4 /dev/mapper/labvol
mkdir -p /mnt/lab && mount /dev/mapper/labvol /mnt/lab
echo "data at rest" > /mnt/lab/file
umount /mnt/lab && cryptsetup close labvol && losetup -d "$LOOP"
```
LUKS2 carries key slots and Argon2 params; closing returns the file to opaque ciphertext.

Step 4 — OpenSSL keys and self-signed certificate

```
apt install -y openssl
openssl genrsa -out /tmp/key.pem 2048
openssl req -x509 -newkey rsa:2048 -nodes -keyout /tmp/srv.key \
  -out /tmp/cert.pem -days 365 -subj "/CN=lab.example.com"
openssl x509 -in /tmp/cert.pem -noout -text | head -n 25
```
req -x509 produces a self-signed cert in one step; x509 -text decodes issuer, subject, and validity.

Step 5 — Hashing and HMAC

```
sha256sum /tmp/msg.txt
openssl dgst -sha256 /tmp/msg.txt
openssl dgst -sha256 -hmac "secret-key" /tmp/msg.txt
```
A hash gives integrity only; an HMAC binds the hash to a shared secret for authentication too.

Step 6 — WireGuard key pair and sample config

```
apt install -y wireguard-tools
umask 077
wg genkey | tee /etc/wireguard/priv | wg pubkey > /etc/wireguard/pub
cat >/etc/wireguard/wg0.conf <<EOF
[Interface]
PrivateKey = $(cat /etc/wireguard/priv)
Address = 10.20.0.1/24
ListenPort = 51820

[Peer]
PublicKey = REPLACE_WITH_PEER_PUBKEY
AllowedIPs = 10.20.0.2/32
EOF
cat /etc/wireguard/wg0.conf
```
WireGuard uses Curve25519 keys; wg-quick up wg0 would raise the tunnel given a peer.

Step 7 — TLS version and cipher inspection

```
echo | openssl s_client -connect example.com:443 -tls1_2 2>/dev/null \
  | openssl x509 -noout -issuer -subject -dates
echo "TLS 1.0 and 1.1 are deprecated; prefer TLS 1.2 and 1.3."
echo "Avoid RC4, 3DES, MD5, and any 'EXPORT' cipher suites."
```
s_client prints the negotiated cert chain; disable legacy versions and weak ciphers.

Test it / key takeaway: gpg --verify confirms the signature and gpg -d recovers plaintext; the LUKS volume mounts only with the passphrase.

Lab 19 — Compliance, Auditing, and File Integrity

Exam objective: 3.6 — Explain the importance of compliance and audit procedures

Goal: Establish file-integrity monitoring and run compliance and audit tooling to detect tampering, rootkits, and misconfigurations. Cover secure deletion, login banners, and OpenSCAP/CIS benchmarks.

What you'll build: Baseline files with AIDE, scan with rkhunter/lynis/nmap, verify packages, and set banners.

Key concepts: AIDE file-integrity baseline, rkhunter rootkit scan, debsums / rpm -V package verification, lynis system audit, nmap -sV service detection, shred secure deletion, OpenSCAP / CIS benchmarks / banners

Step-by-step

Step 1 — AIDE baseline and tamper check

```
apt update && apt install -y aide
aideinit -y -f 2>/dev/null || aide --init
mv /var/lib/aide/aide.db.new /var/lib/aide/aide.db
echo "tampered" >> /etc/hostname
aide --check | head -n 40
```

aide --init builds the cryptographic baseline; --check reports any change against it.

Step 2 — Rootkit hunting with rkhunter

```
apt install -y rkhunter
rkhunter --update || true
rkhunter --propupd -q
rkhunter --check --sk --rwo
```

--sk skips prompts, --rwo shows only warnings, --propupd resyncs the property DB.

Step 3 — Package verification: debsums and rpm -V

```
apt install -y debsums
debsums -c | head -n 20 || echo "No changed files detected"
apt install -y docker.io && systemctl start docker
docker run --rm rockylinux:9 bash -c 'rpm -V coreutils || echo "rpm -V flags
S/M/5/D/L/U/G/T/P"'
```

debsums -c reports modified Debian-packaged files; rpm -V is the RPM equivalent.

Step 4 — System audit with lynis

```
apt install -y lynis
lynis audit system --quick --no-colors 2>&1 | tail -n 40
ls /var/log/lynis*.log
```

lynis runs hundreds of hardening checks and outputs a hardening index and suggestions.

Step 5 — nmap service detection

```
apt install -y nmap openssh-server
systemctl start ssh
nmap -sV -p22,80 localhost
```

-sV probes service banners to identify versions, the first step of vulnerability assessment.

Step 6 — Secure deletion and wipe

```
echo "very sensitive" > /tmp/secret.txt
shred -vn 3 -u /tmp/secret.txt
ls /tmp/secret.txt 2>/dev/null || echo "Shredded"
dd if=/dev/zero of=/tmp/wipe.img bs=1M count=10
LOOP=$(losetup -f --show /tmp/wipe.img)
dd if=/dev/urandom of="$LOOP" bs=1M count=10 status=progress
losetup -d "$LOOP" && rm -f /tmp/wipe.img
```

shred overwrites 3 times and unlinks; on SSDs use full-disk encryption due to wear leveling.

Step 7 — Banners and OpenSCAP / CIS pointers

```
echo "Authorized use only. Activity is monitored." > /etc/issue
cp /etc/issue /etc/issue.net
echo "Welcome to Lab Host" > /etc/motd
apt install -y libopenscap8 || apt install -y openscap-scanner
oscap --version | head -n 5
echo "CIS Benchmarks: https://www.cisecurity.org/cis-benchmarks"
```

/etc/issue shows on TTY, issue.net on network logins, motd after login; OpenSCAP evaluates SCAP/XCCDF content.

Test it / key takeaway: aide --check flags the tampered /etc/hostname, proving the integrity baseline detects unauthorized change.

Domain 4 — Automation, Orchestration, and Scripting (17% of the exam)

This domain covers exam objectives: 4.1 Summarize automation and orchestration use cases and techniques; 4.2 Given a scenario, perform automated tasks using shell scripting; 4.3 Summarize Python basics used for Linux system administration; 4.4 Given a scenario, implement version control using Git; 4.5 Summarize best practices and responsible uses of AI.

Lab 20 — Infrastructure as Code with Ansible

Exam objective: 4.1 — Summarize the use cases and techniques of automation and orchestration

Goal: Learn Infrastructure as Code using Ansible as the primary agentless configuration management tool, and recognize the wider IaC ecosystem (Puppet, OpenTofu, cloud-init, Kubernetes ConfigMaps). Build an idempotent playbook that deploys and configures nginx.

What you'll build: Install Ansible, write an inventory, run ad-hoc commands, and author an idempotent playbook with variables, templates, and handlers.

Key concepts: Ansible, inventory, playbook, idempotency, handlers/templates, ansible-galaxy, GitOps, cloud-init

Step-by-step

Step 1 — Install Ansible and prepare workspace

```
apt update
apt install -y ansible
mkdir -p /root/lab && cd /root/lab
ansible --version
```

Installs Ansible and creates the working directory for the inventory and playbook.

Step 2 — Create inventory pointing at localhost

```
cat > /root/lab/inventory.ini <<'EOF'
[web]
localhost ansible_connection=local
EOF

ansible -i /root/lab/inventory.ini all -m ping
```

ansible_connection=local runs modules directly without SSH; the ping module confirms connectivity.

Step 3 — Run ad-hoc commands

```
cd /root/lab
ansible -i inventory.ini all -m apt -a "name=tree state=present" -b
ansible -i inventory.ini all -m command -a "tree -L 1 /etc"
```
Ad-hoc commands handle one-off tasks; -b is become (sudo).

Step 4 — Write a playbook with variable, template, and handler

```
cat > /root/lab/site.yml <<'EOF'
- name: Deploy a simple nginx site
  hosts: web
  become: true
  vars:
    site_title: "Linux+ XK0-006"
  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present
        update_cache: true
    - name: Ensure nginx is running
      service:
        name: nginx
        state: started
        enabled: true
    - name: Deploy index page from template
      template:
        src: index.html.j2
        dest: /var/www/html/index.html
      notify: Reload nginx
  handlers:
    - name: Reload nginx
      service:
        name: nginx
        state: reloaded
EOF

cat > /root/lab/index.html.j2 <<'EOF'
<h1>{{ site_title }}</h1>
<p>Managed by Ansible on {{ ansible_hostname }}.</p>
EOF

ansible-playbook -i inventory.ini site.yml --check
ansible-playbook -i inventory.ini site.yml
curl -s localhost | head
```
--check is a dry run; handlers fire only when a notifying task changes, keeping runs idempotent.

Step 5 — Install a Galaxy collection and peek at other IaC tools

```
ansible-galaxy collection install community.general
ansible-galaxy collection list | head
apt install -y puppet
puppet apply -e 'notice("hello from puppet")'
mkdir -p /root/lab/tofu && cd /root/lab/tofu
```

```
cat > main.tf <<'EOF'
resource "null_resource" "hello" {
  provisioner "local-exec" {
    command = "echo hello-from-opentofu"
  }
}
EOF
tofu init && tofu apply -auto-approve
```

Galaxy hosts community modules; Puppet uses a declarative DSL and OpenTofu provisions via providers.

Step 6 — GitOps, cloud-init, and Kubernetes ConfigMap samples

```
cat > /root/lab/cloud-init.yaml <<'EOF'
#cloud-config
package_update: true
packages:
  - nginx
write_files:
  - path: /var/www/html/index.html
    content: "Provisioned by cloud-init"
runcmd:
  - systemctl enable --now nginx
EOF

cat > /root/lab/configmap.yaml <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  SITE_TITLE: "Linux+ XK0-006"
  LOG_LEVEL: "info"
EOF

ls /root/lab
```

GitOps stores declarative files in Git; cloud-init runs at first boot and ConfigMaps decouple config from images.

Test it / key takeaway: Idempotent playbooks converge state without unnecessary restarts; Ansible, Puppet, OpenTofu, cloud-init, and ConfigMaps each occupy a distinct IaC niche.

Lab 21 — Bash Scripting

Exam objective: 4.2 — Perform automated tasks using shell scripting

Goal: Practice core Bash scripting constructs (variables, conditionals, loops, functions, option parsing, regex, file tests) and assemble a defensive backup script, then audit it with shellcheck.

What you'll build: Write a series of Bash scripts exercising each language feature, build a real /etc backup script, and lint everything with shellcheck.

Key concepts: shebang, parameter expansion, arrays, test operators [[ ]], getopts, IFS, set -euo pipefail, shellcheck

Step-by-step

Step 1 — Set up a scripts directory

```
mkdir -p /root/scripts && cd /root/scripts
apt update && apt install -y shellcheck
bash --version
```

Creates the workspace and installs shellcheck, the standard static analyser for shell scripts.

Step 2 — Variables, command substitution, parameter expansion

```
cat > /root/scripts/basics.sh <<'EOF'
#!/usr/bin/env bash
name="Linux+"
today=$(date +%F)
greeting="${1:-hello}"
echo "$greeting $name on $today"

fruits=(apple banana cherry)
echo "first=${fruits[0]} count=${#fruits[@]}"
echo "all: ${fruits[*]}"
EOF
chmod +x /root/scripts/basics.sh
/root/scripts/basics.sh "hi"
```

${1:-hello} supplies a default; arrays use () and index with ${arr[i]}.

Step 3 — Conditionals, case, and test operators

```
cat > /root/scripts/checks.sh <<'EOF'
#!/usr/bin/env bash
num=${1:-0}
path=${2:-/etc/hostname}

if [[ $num -eq 0 ]]; then echo "zero"
elif [[ $num -lt 10 ]]; then echo "small"
else echo "big"; fi

case "$path" in
  /etc/*) echo "system config" ;;
  /home/*) echo "user data" ;;
  *)      echo "other" ;;
esac

[[ -f $path ]] && echo "$path is a regular file"
s="hello123"
if [[ $s =~ ^[a-z]+[0-9]+$ ]]; then echo "matches regex"; fi
EOF
```

```
chmod +x /root/scripts/checks.sh
/root/scripts/checks.sh 42 /etc/hostname
```
Numeric tests use -eq/-lt; strings use ==, !=, and =~ (regex) inside [[ ]].

Step 4 — Loops, functions, and exit codes

```
cat > /root/scripts/loops.sh <<'EOF'
#!/usr/bin/env bash
log() { echo "[$(date +%T)] $*"; }

for f in /etc/hosts /etc/hostname /etc/nonexistent; do
  if [[ -f $f ]]; then log "found $f"; else log "missing $f"; fi
done

i=0
while [[ $i -lt 3 ]]; do log "while i=$i"; ((i++)); done

grep -q root /etc/passwd
echo "grep exit code: $?"
EOF
chmod +x /root/scripts/loops.sh
/root/scripts/loops.sh
```
$? holds the previous command's exit code; zero means success.

Step 5 — getopts and IFS-based CSV parsing

```
cat > /root/scripts/opts.sh <<'EOF'
#!/usr/bin/env bash
verbose=0
target=""
while getopts ":vt:" opt; do
  case $opt in
    v) verbose=1 ;;
    t) target=$OPTARG ;;
    *) echo "usage: $0 [-v] [-t target]"; exit 2 ;;
  esac
done
echo "verbose=$verbose target=$target"

row="alice,30,admin"
IFS=',' read -r user age role <<<"$row"
echo "user=$user age=$age role=$role"
EOF
chmod +x /root/scripts/opts.sh
/root/scripts/opts.sh -v -t web01
```
getopts is the POSIX option parser; temporarily changing IFS splits CSV without external tools.

Step 6 — Build a real backup script

```
cat > /root/scripts/backup-etc.sh <<'EOF'
#!/usr/bin/env bash
set -euo pipefail
```

```
SRC="/etc"
DEST="/root/backups"
MAX_AGE_DAYS="${1:-7}"
STAMP=$(date +%Y%m%d-%H%M%S)
ARCHIVE="$DEST/etc-$STAMP.tar.gz"

mkdir -p "$DEST"
latest=$(find "$DEST" -maxdepth 1 -name 'etc-*.tar.gz' -printf '%T@ %p\n'
2>/dev/null | sort -n | tail -1 | awk '{print $2}')

if [[ -n "${latest:-}" ]]; then
  age_days=$(( ( $(date +%s) - $(stat -c %Y "$latest") ) / 86400 ))
  if [[ $age_days -lt $MAX_AGE_DAYS ]]; then
    echo "Recent backup exists; skipping."; exit 0
  fi
fi

tar -czf "$ARCHIVE" "$SRC"
echo "Created $ARCHIVE"
EOF
chmod +x /root/scripts/backup-etc.sh
/root/scripts/backup-etc.sh 7
ls -lh /root/backups
```

set -euo pipefail fails fast; the script only tars /etc if the newest archive is older than N days.

Step 7 — Lint with shellcheck

```
shellcheck /root/scripts/*.sh || true
shellcheck -s bash /root/scripts/backup-etc.sh
```

shellcheck catches common bugs like unquoted variables and using [ ] instead of [[ ]].

Test it / key takeaway: A robust script uses set -euo pipefail, quotes variables, and passes shellcheck cleanly.

Lab 22 — Python for System Administration

Exam objective: 4.3 — Summarize Python basics used for Linux system administration

Goal: Use Python for common Linux admin tasks: create a virtual environment, exercise core data types, walk the filesystem, parse JSON, call a REST API with requests, and enforce PEP 8 with black and flake8.

What you'll build: Set up a venv, write scripts covering data types, os.walk, JSON parsing, and HTTP requests, then organize code into a package and lint it.

Key concepts: venv/pip, core types (bool/int/float/str/list/dict), type hints, os.walk, json module, requests, black/flake8/PEP 8

Step-by-step

Step 1 — Check Python and create a virtual environment

```
python3 --version
apt update && apt install -y python3-venv python3-pip
python3 -m venv ~/lab-venv
source ~/lab-venv/bin/activate
which python
pip install --upgrade pip
```
A venv isolates project packages so pip install cannot break OS tools.

Step 2 — Install dependencies

```
pip install requests black flake8
pip list
mkdir -p ~/pylab && cd ~/pylab
```
requests is the HTTP client; black formats code and flake8 checks PEP 8 style.

Step 3 — Exercise core data types

```
cat > ~/pylab/types_demo.py <<'EOF'
"""Demonstrate the built-in Python data types."""

flag: bool = True
count: int = 42
ratio: float = 3.14
name: str = "linuxplus"
items: list = ["nginx", "ssh", "cron"]
meta: dict = {"distro": "ubuntu", "version": 22.04}

print(type(flag), flag)
print(type(name), name.upper())
print(type(items), items, "len=", len(items))
print(type(meta), meta["distro"])
EOF
python ~/pylab/types_demo.py
```
Type hints (flag: bool) are optional but help readers and tools like mypy catch bugs early.

Step 4 — Walk /etc, count files, emit JSON

```
cat > ~/pylab/scan_etc.py <<'EOF'
"""Walk /etc and print a JSON summary."""
import json
import os
from pathlib import Path



def scan(root: str) -> dict:
    total_files = 0
    total_dirs = 0
    by_ext: dict[str, int] = {}
    for dirpath, dirnames, filenames in os.walk(root):
        total_dirs += len(dirnames)
        total_files += len(filenames)
        for f in filenames:
            ext = Path(f).suffix or "(none)"
```

```
            by_ext[ext] = by_ext.get(ext, 0) + 1
    return {"root": root, "files": total_files, "dirs": total_dirs}



if __name__ == "__main__":
    print(json.dumps(scan("/etc"), indent=2))
EOF
python ~/pylab/scan_etc.py
```

os.walk yields (dirpath, dirnames, filenames) tuples; json.dumps serialises the dict for downstream tools.

Step 5 — Parse a JSON file with the json module

```
cat > ~/pylab/sample.json <<'EOF'
{
  "users": [
    {"name": "alice", "role": "admin"},
    {"name": "bob",   "role": "dev"}
  ]
}
EOF

cat > ~/pylab/read_json.py <<'EOF'
import json
from pathlib import Path

data = json.loads(Path("/root/pylab/sample.json").read_text())
for u in data["users"]:
    print(f"{u['name']} -> {u['role']}")
EOF
python ~/pylab/read_json.py
```

json.loads parses a string, json.load reads a file object; f-strings give readable formatting.

Step 6 — Call a public API with requests

```
cat > ~/pylab/api_call.py <<'EOF'
"""Call a public test API and print selected fields."""
import json
import requests



def fetch(url: str) -> dict:
    resp = requests.get(url, timeout=10)
    resp.raise_for_status()
    return resp.json()



if __name__ == "__main__":
    payload = fetch("https://httpbin.org/get")
    print("URL :", payload["url"])
    print("UA  :", payload["headers"].get("User-Agent"))
EOF
```

```
python ~/pylab/api_call.py
```
raise_for_status turns 4xx/5xx responses into exceptions so failures are loud, not silent.

Step 7 — Use a package module and apply PEP 8

```
mkdir -p ~/pylab/helpers
cat > ~/pylab/helpers/__init__.py <<'EOF'
"""Tiny helper package."""
EOF
cat > ~/pylab/helpers/fs.py <<'EOF'
from pathlib import Path



def count_files(root: str) -> int:
    return sum(1 for _ in Path(root).rglob("*") if _.is_file())
EOF

cat > ~/pylab/use_pkg.py <<'EOF'
from helpers.fs import count_files

print("files in /etc:", count_files("/etc"))
EOF

cd ~/pylab && python use_pkg.py
black ~/pylab
flake8 ~/pylab --max-line-length=100
```
black rewrites code to a canonical style; flake8 reports remaining PEP 8 issues.

Test it / key takeaway: A venv isolates dependencies, requests loudly fails on bad HTTP, and black+flake8 enforce PEP 8 automatically.

Lab 23 — Git Version Control

Exam objective: 4.4 — Implement version control using Git

Goal: Work through Git end to end: init, commit, branch, merge, resolve conflicts, stash, tag, rewrite history with reset and rebase, and push/pull against a local bare remote.

What you'll build: Create a working repo and a bare origin repo, then practice the full Git workflow from first commit through conflict resolution, rebasing, and remote push/pull.

Key concepts: git init/config, branch/merge --no-ff, conflict resolution, stash/tag, reset --soft/rebase, bare remote, push/pull --rebase, .gitignore

Step-by-step

Step 1 — Install Git and set identity

```
apt update && apt install -y git
git config --global user.name  "Linux Plus Learner"
git config --global user.email "learner@example.com"
```

```
git config --global init.defaultBranch main
git config --global --list
```
user.name/email appear in every commit; init.defaultBranch makes new repos start on main.

Step 2 — Initialise a repo and make the first commit

```
mkdir -p /root/lab && cd /root/lab
git init lab && cd lab
cat > README.md <<'EOF'
# Lab repo
EOF
cat > .gitignore <<'EOF'
*.log
*.tmp
__pycache__/
EOF
git add README.md .gitignore
git commit -m "initial commit: readme and gitignore"
git log --oneline --graph --decorate
```
.gitignore keeps build artifacts out of the repo; log --oneline --graph visualises history.

Step 3 — Branch, modify, merge

```
git switch -c feature/banner
echo "Welcome to the lab" > banner.txt
git add banner.txt
git commit -m "feat: add welcome banner"

git switch main
git merge --no-ff feature/banner -m "merge feature/banner"
git log --oneline --graph --all
```
--no-ff forces a merge commit so the branch topology stays visible in history.

Step 4 — Create and resolve a conflict

```
git switch -c feature/edit
sed -i 's/Welcome to the lab/Welcome, learner/' banner.txt
git commit -am "tweak banner wording"

git switch main
sed -i 's/Welcome to the lab/Hello from main/' banner.txt
git commit -am "main edits banner"

git merge feature/edit || true
cat > banner.txt <<'EOF'
Hello learner, welcome to the lab
EOF
git add banner.txt
git commit -m "resolve banner conflict"
```
Conflict markers flag the disputed region; you edit the file, git add, and commit to resolve.

Step 5 — diff, stash, tag

```
echo "draft note" >> README.md
git diff
git stash
git status
git stash pop

git tag v0.1
git tag --list
```
git stash shelves uncommitted work so you can switch context; tags mark releases.

Step 6 — Rewrite history: reset and rebase

```
echo "oops" > scratch.txt
git add scratch.txt
git commit -m "WIP scratch"

git reset --soft HEAD~1
git restore --staged scratch.txt
rm scratch.txt

git switch -c topic/rebase
echo "line A" >> README.md && git commit -am "topic: line A"
git switch main
echo "main change" >> README.md && git commit -am "main: change"
git switch topic/rebase
git rebase main || true
```
reset --soft HEAD~1 undoes a commit but keeps changes staged; rebase replays commits onto a new base.

Step 7 — Add a remote and push/pull

```
git init --bare /tmp/origin.git
cd /root/lab/lab
git remote add origin /tmp/origin.git
git push -u origin main
git push origin v0.1

git clone /tmp/origin.git /tmp/teammate
cd /tmp/teammate
echo "teammate edit" >> README.md
git commit -am "team edit"
git push

cd /root/lab/lab
git fetch
git pull --rebase
```
A bare repo has no working tree and is what services like GitHub host; pull --rebase keeps history linear.

Test it / key takeaway: Git tracks history through commits and branches; merge/rebase integrate work and pull --rebase keeps a linear remote history.

Lab 24 — Responsible AI Use for Linux Administrators

Exam objective: 4.5 — Summarize best practices and responsible uses of artificial intelligence (AI)

Goal: Adopt a verify-before-paste workflow for AI-generated Linux commands: redact secrets before sending prompts to hosted models, lint and sandbox suggestions, weigh local vs hosted models, and draft a corporate AI usage policy.

What you'll build: Build a verification workspace, redact secrets in prompts, shellcheck and sandbox an AI-suggested command, and author a verification checklist plus an AI usage policy.

Key concepts: verify-before-paste, secret redaction, local vs hosted LLMs, shellcheck, API key via env var, AI usage policy, data governance, attribution

Step-by-step

Step 1 — Set up a verification workspace

```
apt update && apt install -y shellcheck jq curl
mkdir -p /root/ai-lab && cd /root/ai-lab
shellcheck --version | head -1
```

Every AI-generated shell snippet you accept should pass shellcheck first; jq/curl talk to JSON APIs.

Step 2 — Try a local LLM (optional)

```
curl -fsSL https://ollama.com/install.sh | sh || echo "Ollama install skipped"
(ollama serve >/tmp/ollama.log 2>&1 &) ; sleep 2
ollama pull tinyllama 2>/dev/null && \
  ollama run tinyllama "Write a one-line bash command to list the 5 largest
files in /var/log" \
  || echo "Local model unavailable; continue with the placeholder API
workflow."
```

Local models keep prompts on your machine, which matters when prompts contain sensitive data.

Step 3 — Hosted-API placeholder workflow with redaction

```
export LLM_API_KEY="REDACTED-EXAMPLE"

cat > /root/ai-lab/ask.sh <<'EOF'
#!/usr/bin/env bash
set -euo pipefail
prompt="${1:-Write a bash one-liner to show top 5 largest files under
/var/log}"
safe_prompt=$(printf '%s' "$prompt" \
  | sed -E 's/(AKIA[0-9A-Z]{16})/REDACTED-AWS-KEY/g' \
  | sed -E 's/(ghp_[A-Za-z0-9]{20,})/REDACTED-GH-TOKEN/g' \
  | sed -E 's/(password=)[^ ]+/\1REDACTED/gi')
```

```
curl -s -X POST https://httpbin.org/post \
  -H "Authorization: Bearer ${LLM_API_KEY}" \
  -H "Content-Type: application/json" \
  -d "$(jq -n --arg p "$safe_prompt" '{prompt:$p}')" \
  | jq '.json.prompt'
EOF
chmod +x /root/ai-lab/ask.sh
/root/ai-lab/ask.sh "list AKIAABCDEFGHIJKLMNOP and password=hunter2"
```

The script masks AWS keys, GitHub tokens, and password= strings, and reads the API key from an env var, never hard-coded.

Step 4 — Verify an AI-suggested command

```
cat > /root/ai-lab/suggested.sh <<'EOF'
#!/usr/bin/env bash
# AI-suggested: top 5 largest files under /var/log
find /var/log -type f -printf '%s %p\n' 2>/dev/null \
  | sort -nr | head -5 | awk '{printf "%10d  %s\n",$1,$2}'
EOF
chmod +x /root/ai-lab/suggested.sh

shellcheck /root/ai-lab/suggested.sh
cat /root/ai-lab/suggested.sh
/root/ai-lab/suggested.sh
```

The rule: read every line, run shellcheck, then execute in a non-production environment first.

Step 5 — Build a verification checklist

```
cat > /root/ai-lab/checklist.md <<'EOF'
# Verify-before-paste checklist

- [ ] Did I read every line of the suggested command/script?
- [ ] Did I run shellcheck (or pylint/mypy for Python)?
- [ ] Does it touch /, /etc, /var, /boot, or production paths?
- [ ] Does it use rm -rf, dd, mkfs, chmod 777, or pipe to sh?
- [ ] Are there hard-coded credentials, tokens, or hostnames?
- [ ] Have I tested it in a sandbox (Killercoda, VM, container)?
- [ ] Have I attributed AI assistance per company policy?
EOF
cat /root/ai-lab/checklist.md
```

Treat the list as gating criteria; if any box is unchecked, do not run the snippet on a real system.

Step 6 — Draft a corporate AI usage policy

```
cat > /root/ai-lab/policy.md <<'EOF'
# AI Usage Policy (template)

## Permitted models
- Local models on company hardware: allowed for internal, non-secret data.
- Hosted models on approved enterprise tier: only after redaction.
- Personal/free hosted tiers: NOT allowed for internal data.
```

```
## Data handling
- No customer PII, credentials, or keys sent without DPO approval.
- Treat all model outputs as untrusted until reviewed.

## Verification
- All AI-generated code must pass shellcheck/flake8 and line-by-line review.

## Attribution
- Note AI assistance in commit messages when it shaped the change.
EOF
wc -l /root/ai-lab/policy.md
```

A real policy covers scope, permitted models, data handling, verification, governance, attribution, and incidents.

Step 7 — Compare local vs hosted at a glance

```
cat <<'EOF'
Local model (Ollama, llamafile)
  + Prompts and data never leave the host
  + Works offline; no per-token cost
  - Smaller models, lower quality

Hosted model (OpenAI/Anthropic/etc.)
  + Highest-quality models, fast
  - Prompts traverse the internet to a third party
  - Cost per token; training/retention concerns
EOF
```

Local models trade quality for privacy while hosted models do the opposite; corporate policy decides which fits.

Test it / key takeaway: Always verify before paste: redact secrets, shellcheck, sandbox, then run; choose local vs hosted models per policy.

Domain 5 — Troubleshooting (22% of the exam)

This domain covers exam objectives: 5.1 Summarize monitoring concepts and configurations; 5.2 Given a scenario, troubleshoot hardware, storage, and OS issues; 5.3 Given a scenario, troubleshoot networking issues; 5.4 Given a scenario, troubleshoot security issues; 5.5 Given a scenario, troubleshoot performance issues.

Lab 25 — Monitoring Concepts and Tools

Exam objective: 5.1 — Summarize monitoring concepts and configurations in a Linux system

Goal: Learn the monitoring vocabulary (SLA/SLO/SLI) and stand up a working monitoring stack to produce metrics, alerts, webhooks, health checks, and aggregated logs.

What you'll build: Install SNMP, node_exporter, Prometheus and Alertmanager, then write and validate alert rules, test webhooks, and inspect journald logs.

Key concepts: SLA vs SLO vs SLI, SNMP OID/MIB walk, Prometheus scrape config, node_exporter metrics endpoint, promtool check rules, webhook alerting, curl health checks, journalctl log aggregation

Step-by-step

Step 1 — Install SNMP and walk the system MIB

```
apt update
apt install -y snmpd snmp snmp-mibs-downloader
systemctl enable --now snmpd
snmpwalk -v 2c -c public localhost 1.3.6.1.2.1.1
```

SNMP exposes device data through an OID tree; the system group holds hostname, uptime, contact, location.

Step 2 — Install and probe node_exporter

```
apt install -y prometheus-node-exporter
systemctl enable --now prometheus-node-exporter
ss -tlnp | grep 9100
curl -s http://localhost:9100/metrics | head -n 20
```

node_exporter publishes host metrics on port 9100 in Prometheus text exposition format.

Step 3 — Install Prometheus and scrape node_exporter

```
apt install -y prometheus
cat >/etc/prometheus/prometheus.yml <<'EOF'
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
EOF
systemctl restart prometheus
curl -s 'http://localhost:9090/api/v1/targets' | head -c 400
```

Prometheus pulls metrics from each target on the scrape interval into a local time-series DB.

Step 4 — Write and validate an alert rule

```
cat >/etc/prometheus/rules.yml <<'EOF'
groups:
- name: node-alerts
  rules:
  - alert: NodeDown
    expr: up{job="node"} == 0
    for: 1m
    labels:
      severity: critical
    annotations:
```

```
      summary: "Node exporter is down"
EOF
apt install -y prometheus-alertmanager
promtool check rules /etc/prometheus/rules.yml
```
promtool validates YAML structure and PromQL before Prometheus reloads the rules.

Step 5 — Simulate a webhook receiver

```
(nc -l -p 8000 -q 1 >/tmp/webhook.log 2>&1 &) ; sleep 1
curl -s -X POST -H 'Content-Type: application/json' \
  -d '{"alert":"NodeDown","severity":"critical"}' \
  http://localhost:8000/
sleep 1
cat /tmp/webhook.log
```
Alertmanager delivers alerts by POSTing JSON to a webhook; nc verifies the payload shape.

Step 6 — Health check pattern

```
curl -fsS http://localhost:9100/metrics >/dev/null && echo "node_exporter OK"
curl -fsS http://localhost:9090/-/healthy && echo
curl -fsS http://localhost:9090/-/ready && echo
```
curl -f exits non-zero on HTTP errors while -sS keeps errors visible: the scriptable health-check idiom.

Step 7 — Log aggregation via journalctl

```
journalctl -u prometheus -n 20 --no-pager
journalctl -u prometheus-node-exporter -n 20 --no-pager
journalctl --since "10 min ago" -p err --no-pager
```
The systemd journal is the local log aggregation layer, forwardable to Loki/Elastic/Splunk.

Test it / key takeaway: node_exporter, Prometheus, and its health/ready endpoints all respond, and promtool confirms the alert rule is valid.

Lab 26 — Troubleshooting Hardware, Storage, and OS Issues

Exam objective: 5.2 — Analyze and troubleshoot hardware, storage, and Linux OS issues

Goal: Deliberately inject common OS and storage failures then diagnose and repair each, keeping every breakage contained to loopback devices, throwaway units, or subshells.

What you'll build: Reproduce a full filesystem, inode exhaustion, GRUB validation, a runaway CPU process, a failed systemd unit, a broken PATH, and a memory leak, then fix them.

Key concepts: ENOSPC / df -h, inode exhaustion / df -i, grub-mkconfig validation, runaway process / top -bn1, failed systemd unit, broken PATH recovery, memory leak / ps rss / free

Step-by-step

Step 1 — Break: fill a filesystem (ENOSPC)

```
mkdir -p /mnt/tinyfs
dd if=/dev/zero of=/tmp/tiny.img bs=1M count=64
mkfs.ext4 -F /tmp/tiny.img
mount -o loop /tmp/tiny.img /mnt/tinyfs
dd if=/dev/zero of=/mnt/tinyfs/fill bs=1M count=200 || echo "ENOSPC as
expected"
df -h /mnt/tinyfs
du -shx /mnt/tinyfs/* | sort -h
rm /mnt/tinyfs/fill
```

dd fails with ENOSPC once the 64MB loopback fills; du finds the biggest consumer and deletion restores space.

Step 2 — Break: inode exhaustion

```
df -i /mnt/tinyfs
mkdir /mnt/tinyfs/many
(cd /mnt/tinyfs/many && for i in $(seq 1 20000); do : > f$i 2>/dev/null ||
break; done)
df -i /mnt/tinyfs
rm -rf /mnt/tinyfs/many
df -i /mnt/tinyfs
```

A filesystem can be near-empty by bytes yet refuse files because all inodes are used; only df -i reveals it.

Step 3 — Validate a GRUB config without committing it

```
grub-mkconfig -o /tmp/grub.cfg.new 2>&1 | tail -n 5
diff -u /boot/grub/grub.cfg /tmp/grub.cfg.new | head -n 40 || true
```

Generating to a scratch path previews boot-loader changes before overwriting the real grub.cfg.

Step 4 — Break: runaway CPU process

```
bash -c 'while :; do :; done' &
RUNAWAY=$!
sleep 2
top -bn1 -p $RUNAWAY | tail -n 5
ps -o pid,pcpu,cmd -p $RUNAWAY
kill -TERM $RUNAWAY
wait $RUNAWAY 2>/dev/null
```

A busy-loop pegs one CPU at 100%; top -bn1 -p PID confirms the culprit and kill resolves it.

Step 5 — Break: a failing systemd unit

```
cat >/etc/systemd/system/lab-broken.service <<'EOF'
[Unit]
Description=Deliberately broken unit
[Service]
ExecStart=/bin/false
EOF
systemctl daemon-reload
systemctl start lab-broken.service || true
systemctl status lab-broken.service --no-pager | head -n 15
journalctl -u lab-broken.service --no-pager | tail -n 10
```

/bin/false exits 1 so systemd marks the unit failed; status and journal show which command failed.

Step 6 — Break and recover a broken PATH

```
( export PATH=/nonexistent
  ls 2>&1 || echo "ls vanished"
  /bin/ls / | head
  export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
  ls / | head
)
echo "Parent shell PATH unchanged: $PATH"
```

A subshell confines the broken PATH; absolute paths like /bin/ls still work as the recovery technique.

Step 7 — Simulate a memory leak

```
cat >/tmp/leak.py <<'EOF'
data = []
while True:
    data.append("x" * 1024 * 100)
EOF
python3 /tmp/leak.py &
LEAK=$!
sleep 3
ps -o pid,rss,vsz,cmd -p $LEAK
free -m
kill -KILL $LEAK
```

ps -o rss shows resident memory climbing while free -m shows shrinking available memory: the leak fingerprint.

Test it / key takeaway: Each fault is confirmed by the right counter (df, df -i, top, systemctl status, ps rss) and reversed cleanly without touching the host.

Lab 27 — Troubleshooting Network Connectivity

Exam objective: 5.3 — Analyze and troubleshoot networking issues on a Linux system

Goal: Break and repair the common network faults (DNS, routing, firewall, MTU, IP, link, dual-stack) while practicing the standard L3/L4 triage toolkit.

What you'll build: Baseline the host, then inject DNS, gateway, firewall, MTU, duplicate-IP, and link-down faults, and finish with IPv4/IPv6 and packet-capture triage.

Key concepts: ip addr / ip route baseline, resolv.conf / dig, default gateway, ufw port block, MTU / ping -M do, duplicate IP / arping, dummy interface link toggle, tcpdump / mtr / traceroute / nmap

Step-by-step

Step 1 — Baseline and install tools

```
apt update
apt install -y dnsutils iproute2 iputils-ping mtr-tiny traceroute nmap tcpdump
ufw netcat-openbsd
ip -br addr
ip route
cat /etc/resolv.conf
ss -tulpn | head
```
A baseline of interfaces, routes, resolver, and listening sockets is the most useful incident artifact.

Step 2 — Break: DNS, then fix

```
cp /etc/resolv.conf /tmp/resolv.conf.bak
sed -i 's/^nameserver/#nameserver/' /etc/resolv.conf
dig +time=2 +tries=1 example.com || echo "DNS broken as expected"
echo 'nameserver 8.8.8.8' > /etc/resolv.conf
dig +short example.com
cp /tmp/resolv.conf.bak /etc/resolv.conf
```
Commenting nameservers reproduces the classic 'internet is down' report; a public resolver confirms DNS is the fault.

Step 3 — Break: default gateway, then revert

```
ORIG=$(ip route show default)
ip route del default 2>/dev/null
ip route add default via 10.10.10.10 2>/dev/null || true
ping -c 2 -W 2 1.1.1.1 || echo "No route as expected"
ip route del default 2>/dev/null
ip route add $ORIG
ping -c 2 -W 2 1.1.1.1
```
A wrong gateway looks like an unplugged cable; saving the original route string is the safe recovery pattern.

Step 4 — Break: firewall blocks a port

```
ufw --force enable
ufw deny 8080/tcp
python3 -m http.server 8080 >/tmp/http.log 2>&1 &
sleep 2
IF=$(ip -br link | awk '$1!="lo" && $2=="UP" {print $1; exit}')
curl -s --max-time 3 --interface lo http://127.0.0.1:8080/ | head -c 60
curl -s --max-time 3 --interface "$IF" http://127.0.0.1:8080/ || echo "Blocked
on $IF as expected"
ufw delete deny 8080/tcp; ufw --force disable
```
Loopback bypasses ufw but the physical interface does not; comparing the two pinpoints firewall rules.

Step 5 — Break: MTU mismatch

```
IF=$(ip -br link | awk '$1!="lo" && $2=="UP" {print $1; exit}')
ORIG_MTU=$(cat /sys/class/net/$IF/mtu)
ip link set $IF mtu 1300
ping -M do -s 1400 -c 2 -W 2 1.1.1.1 || echo "Fragmentation needed as expected"
ip link set $IF mtu $ORIG_MTU
```

```
ping -M do -s 1400 -c 2 -W 2 1.1.1.1
```
-M do sets the don't-fragment bit; oversize payloads are refused, the fingerprint of a VPN/tunnel MTU mismatch.

Step 6 — Break: duplicate IP and link down

```
IF=$(ip -br link | awk '$1!="lo" && $2=="UP" {print $1; exit}')
ip addr add 10.99.99.99/24 dev $IF
arping -c 1 -I $IF 10.99.99.99 2>/dev/null || true
ip addr del 10.99.99.99/24 dev $IF
modprobe dummy
ip link add lab0 type dummy
ip link set lab0 up; ip link set lab0 down; ip link del lab0
```
A secondary address and a dummy interface let you safely observe duplicate-IP ARP flapping and link toggling.

Step 7 — IPv4 vs IPv6 and capture

```
curl -4 -s --max-time 3 https://ifconfig.co || true
curl -6 -s --max-time 3 https://ifconfig.co || true
mtr -rwc 3 1.1.1.1 | head
traceroute -n -m 5 1.1.1.1 | head
timeout 3 tcpdump -i any -nn -c 5 'icmp' &
ping -c 3 1.1.1.1 >/dev/null; wait
nmap -sT -p 22,80,443 127.0.0.1
```
curl -4/-6 forces address family; tcpdump, mtr, traceroute, and nmap complete the L3/L4 triage toolkit.

Test it / key takeaway: Each single-variable fault produces its expected symptom and reverts cleanly, and IPv4/IPv6 plus capture tools confirm reachability.

Lab 28 — Troubleshooting Security Issues

Exam objective: 5.4 — Analyze and troubleshoot security issues on a Linux system

Goal: Diagnose and repair security faults spanning SELinux contexts, POSIX ACLs, certificate trust, account lockout, broken repos, and weak TLS, using a Rocky container for the SELinux portion.

What you'll build: Pull a Rocky container for SELinux practice, then break and fix ACLs, CA trust, pam_faillock lockout, a bad apt repo, and weak TLS on the Ubuntu host.

Key concepts: SELinux context / chcon / restorecon, AVC denials / ausearch / audit2allow, POSIX ACL / getfacl / setfacl, CA trust / update-ca-certificates, pam_faillock lockout, apt repo / TLS s_client, unattended-upgrades

Step-by-step

Step 1 — Prep host and pull a Rocky container

```
apt update
```

```
apt install -y docker.io acl openssl libpam-modules ca-certificates
systemctl start docker
docker pull rockylinux:9
docker run -dit --name selab --privileged rockylinux:9 bash
docker exec selab dnf -y install policycoreutils policycoreutils-python-utils
selinux-policy-targeted setools-console audit httpd
```

The privileged container lets you load SELinux userspace tooling to practice the commands without an enforcing kernel.

Step 2 — Break: SELinux context, then restorecon

```
docker exec selab bash -c '
mkdir -p /var/www/html && echo hello > /tmp/index.html
cp /tmp/index.html /var/www/html/index.html
ls -Z /var/www/html/index.html
chcon -t user_home_t /var/www/html/index.html
ls -Z /var/www/html/index.html
restorecon -v /var/www/html/index.html
ls -Z /var/www/html/index.html
'
```

chcon shows a copy landing with the wrong type; restorecon resets it from the policy database.

Step 3 — Inspect AVC denials with audit2allow

```
docker exec selab bash -c '
ausearch -m avc 2>/dev/null | tail -n 20 || echo "no AVCs recorded"
echo "type=AVC msg=audit(0): avc: denied { read } for pid=1 comm=\"demo\"
name=\"x\" dev=\"sda1\" ino=1 scontext=u:r:httpd_t:s0
tcontext=u:object_r:user_home_t:s0 tclass=file" > /tmp/avc.log
audit2allow -i /tmp/avc.log
'
```

ausearch -m avc lists policy denials; audit2allow generates a candidate module, which you must read before applying.

Step 4 — Break: ACL on Ubuntu host

```
useradd -m alice 2>/dev/null || true
echo secret > /tmp/payroll.txt
setfacl -m u:alice:--- /tmp/payroll.txt
getfacl /tmp/payroll.txt
sudo -u alice cat /tmp/payroll.txt 2>&1 || echo "ACL denied as expected"
setfacl -x u:alice /tmp/payroll.txt
sudo -u alice cat /tmp/payroll.txt
```

POSIX ACLs override mode bits; getfacl is the only way to see them since ls -l shows only a +.

Step 5 — Break: certificate trust

```
mkdir -p /tmp/ca && cd /tmp/ca
openssl req -x509 -newkey rsa:2048 -nodes -days 1 \
  -keyout ca.key -out ca.crt -subj '/CN=Lab CA' 2>/dev/null
openssl s_client -connect self-signed.badssl.com:443 -servername self-
signed.badssl.com </dev/null 2>&1 | grep -E 'verify|Verification' | head
cp ca.crt /usr/local/share/ca-certificates/lab-ca.crt
update-ca-certificates 2>&1 | tail -n 3
```

Self-signed certs are untrusted; installing the CA and rerunning update-ca-certificates rebuilds the trust bundle.

Step 6 — Break: account lockout with pam_faillock

```
grep -q faillock /etc/pam.d/common-auth || cat >>/etc/pam.d/common-auth <<'EOF'
auth required pam_faillock.so preauth deny=3 unlock_time=120
auth required pam_faillock.so authfail deny=3 unlock_time=120
EOF
for i in 1 2 3 4; do echo "wrongpw" | su - alice -c "true" 2>/dev/null; done
faillock --user alice 2>/dev/null || echo "faillock state recorded"
faillock --user alice --reset 2>/dev/null || true
sed -i '/pam_faillock.so/d' /etc/pam.d/common-auth
```

pam_faillock records failed attempts; --reset clears the counter and re-enables login.

Step 7 — Break: misconfigured apt repo and weak TLS

```
echo 'deb http://no.such.host.invalid/ubuntu jammy main' >
/etc/apt/sources.list.d/broken.list
apt update 2>&1 | tail -n 5 || true
rm /etc/apt/sources.list.d/broken.list; apt update >/dev/null 2>&1
openssl s_client -connect www.google.com:443 -tls1 </dev/null 2>&1 | grep -iE
'protocol|cipher|alert' | head
openssl s_client -connect www.google.com:443 -tls1_2 </dev/null 2>&1 | grep -iE
'protocol|cipher' | head
apt install -y unattended-upgrades
```

A bad sources.list line simulates a repo outage; -tls1 is rejected while -tls1_2 succeeds, the cipher negotiation test.

Test it / key takeaway: restorecon fixes SELinux type, getfacl/setfacl explains hidden ACL denials, the CA trust bundle rebuilds, faillock resets, and modern TLS negotiates while TLS1.0 is refused.

Lab 29 — Troubleshooting Performance Issues

Exam objective: 5.5 — Analyze and troubleshoot performance issues

Goal: Stress CPU, memory, disk I/O, and network in turn, then walk the universal performance loop: baseline, spike, identify the bottleneck with the right counter, and resolve it.

What you'll build: Install stress-ng, fio, and sysstat, baseline the host, then generate controlled CPU, memory, disk, and network spikes and attribute each to the right counter and PID.

Key concepts: baseline: uptime/free/vmstat/mpstat, stress-ng CPU spike / load average, memory pressure / vmstat si-so, fio disk storm / iostat -x, D-state / ping jitter, pidstat context switches, ps --sort triage script

Step-by-step

Step 1 — Install the toolkit and baseline

```
apt update
apt install -y stress-ng fio sysstat htop iotop iftop dstat procps
uptime
free -m
vmstat 1 3
mpstat 1 2
df -h /
```

Record uptime/load, memory, vmstat, and mpstat first; without a baseline 'high CPU' has no meaning.

Step 2 — Generate and observe a CPU spike

```
stress-ng --cpu 4 --timeout 30 &
sleep 5
uptime
mpstat -P ALL 1 3
top -bn1 | head -n 15
wait
```

Load average crosses the CPU count when work outpaces cores; mpstat -P ALL shows if it is one thread or spread.

Step 3 — Memory pressure and swap behavior

```
free -m
stress-ng --vm 2 --vm-bytes 256M --timeout 20 &
sleep 5
free -m
vmstat 1 5
wait
free -m
```

vmstat si/so show pages swapping in/out, the unambiguous sign of memory pressure; zero means RAM is fine.

Step 4 — Disk I/O storm

```
fio --name=lab --rw=randwrite --bs=4k --size=128M --runtime=20 --time_based \
    --filename=/tmp/fio.dat --ioengine=sync >/tmp/fio.log 2>&1 &
sleep 3
iostat -x 1 3
pidstat -d 1 3
wait
rm -f /tmp/fio.dat
```

iostat -x shows per-device %util/await/queue depth; pidstat -d attributes IOPS to processes.

Step 5 — Network jitter and blocked processes

```
ping -i 0.2 -c 20 8.8.8.8 | tail -n 5
dd if=/dev/zero of=/tmp/big bs=1M count=200 oflag=direct &
DDPID=$!
sleep 1
ps -o pid,stat,wchan,cmd -p $DDPID
ps -eo stat,pid,cmd | awk '$1 ~ /^D/' | head
```

```
wait $DDPID; rm -f /tmp/big
```
ping mdev quantifies jitter; a process in state D is uninterruptible-sleep, almost always waiting on disk or NFS.

Step 6 — Context switches and interactive tools

```
pidstat -w 1 5
timeout 3 dstat -tcnd 1 2>/dev/null || timeout 3 vmstat 1
timeout 3 htop -d 10 || true
which perf >/dev/null && perf top -d 5 || echo "perf not installed; skip"
```
pidstat -w splits voluntary from involuntary context switches; high involuntary means the CPU is oversubscribed.

Step 7 — Walk the full loop on a synthetic incident

```
echo "BASELINE:"; uptime; free -m | head -n 2
stress-ng --cpu 2 --vm 1 --vm-bytes 128M --io 1 --timeout 25 &
sleep 6
echo "TOP CPU:"; ps -eo pid,pcpu,pmem,cmd --sort=-pcpu | head -n 5
echo "TOP MEM:"; ps -eo pid,pcpu,pmem,cmd --sort=-pmem | head -n 5
echo "IO WAIT:"; iostat -c 1 1 | tail -n 2
wait
echo "RECOVERED:"; uptime
```
The universal triage script: baseline, spike, top CPU/memory consumers, IO wait, then recovery.

Test it / key takeaway: Each spike maps to its diagnostic counter (load vs mpstat, si/so vs free, %util vs await) and the triage script shows baseline-to-recovery.

Lab 30 — Capstone: End-to-End Server Build and Triage

Exam objective: Capstone — Capstone: End-to-end server build & triage (all domains)

Goal: Integrate Objectives 1-5 into one coherent build: a hardened admin account, LVM storage, nginx with a systemd override, a locked firewall, a health-check timer, and git-versioned config, then inject a fault and triage it end-to-end.

What you'll build: Provision webops with hardened SSH, build LVM-backed /srv/web, deploy nginx, lock ufw to web ports, add a health-check timer, version /etc/nginx in git, then inject and remediate a disk-full fault.

Key concepts: hardened SSH / narrow sudo, LVM: pvcreate/vgcreate/lvcreate, systemd drop-in override, ufw default-deny allowlist, systemd timer health check, git-versioned /etc/nginx, fault injection and triage loop

Step-by-step

Step 1 — Provision a user and harden SSH

```
apt update
apt install -y openssh-server sudo nginx lvm2 ufw git curl
useradd -m -s /bin/bash -G sudo webops
```

```
echo 'webops:LabPass!23' | chpasswd
echo 'webops ALL=(ALL) NOPASSWD: /bin/systemctl, /usr/bin/journalctl'
>/etc/sudoers.d/webops
sed -i 's/^#\?PermitRootLogin.*/PermitRootLogin no/' /etc/ssh/sshd_config
sed -i 's/^#\?PasswordAuthentication.*/PasswordAuthentication no/'
/etc/ssh/sshd_config
sshd -t && systemctl restart ssh
```

A non-root admin with narrow sudo plus key-only SSH is the standard hardening baseline.

Step 2 — Build LVM-backed /srv/web over loopback

```
dd if=/dev/zero of=/var/lab-pv.img bs=1M count=200
LOOP=$(losetup --find --show /var/lab-pv.img)
pvcreate $LOOP
vgcreate labvg $LOOP
lvcreate -n weblv -L 150M labvg
mkfs.ext4 /dev/labvg/weblv
mkdir -p /srv/web && mount /dev/labvg/weblv /srv/web
echo "/dev/labvg/weblv /srv/web ext4 defaults 0 2" >>/etc/fstab
echo '<h1>Capstone OK</h1>' >/srv/web/index.html
```

Loopback PV to VG to LV to ext4 mirrors the real disk pipeline; fstab proves the boot-time mount path.

Step 3 — Install nginx with a systemd override binding /srv/web

```
rm -f /etc/nginx/sites-enabled/default
cat >/etc/nginx/sites-available/capstone <<'EOF'
server {
    listen 80 default_server;
    root /srv/web;
    index index.html;
    location / { try_files $uri $uri/ =404; }
}
EOF
ln -sf /etc/nginx/sites-available/capstone /etc/nginx/sites-enabled/capstone
mkdir -p /etc/systemd/system/nginx.service.d
printf '[Service]\nReadWritePaths=/srv/web\n'
>/etc/systemd/system/nginx.service.d/override.conf
systemctl daemon-reload
nginx -t && systemctl restart nginx
curl -sf http://localhost/ | head
```

A drop-in override extends a packaged unit without touching the vendor file.

Step 4 — Firewall: allow only 80/443 and SSH

```
ufw --force reset >/dev/null
ufw default deny incoming
ufw default allow outgoing
ufw allow 22/tcp
ufw allow 80/tcp
ufw allow 443/tcp
ufw --force enable
ufw status verbose
```

Default-deny inbound with an explicit allowlist is the only firewall pattern for a real server.

Step 5 — Health-check script with a systemd timer

```
cat >/usr/local/bin/web-health.sh <<'EOF'
#!/usr/bin/env bash
set -euo pipefail
code=$(curl -s -o /dev/null -w "%{http_code}" http://localhost/)
ts=$(date -Is)
if [[ "$code" != "200" ]]; then echo "$ts FAIL code=$code" | tee -a
/var/log/web-health.log; exit 1; fi
echo "$ts OK" >>/var/log/web-health.log
EOF
chmod +x /usr/local/bin/web-health.sh
printf '[Unit]\nDescription=Web health check\n[Service]
\nType=oneshot\nExecStart=/usr/local/bin/web-health.sh\n'
>/etc/systemd/system/web-health.service
printf '[Unit]\nDescription=Run web health check every minute\n[Timer]
\nOnBootSec=30s\nOnUnitActiveSec=1min\nUnit=web-health.service\n[Install]
\nWantedBy=timers.target\n' >/etc/systemd/system/web-health.timer
systemctl daemon-reload
systemctl enable --now web-health.timer
systemctl list-timers --no-pager | head
```

A systemd timer is the modern cron replacement that also emits journal entries you can correlate with metrics.

Step 6 — Commit /etc/nginx to a local git repo

```
cd /etc/nginx
git init -q
git config user.email ops@example.local
git config user.name "Lab Ops"
git add -A
git commit -q -m "Initial nginx configuration for capstone"
git log --oneline
```

Versioning /etc is the cheap, no-dependency form of configuration management: every change is recoverable.

Step 7 — Inject a fault and triage

```
systemctl stop nginx
dd if=/dev/zero of=/srv/web/filler bs=1M count=160 2>/dev/null || true
systemctl start nginx || true
systemctl status nginx --no-pager | head -n 12
journalctl -u nginx -n 20 --no-pager
ss -tlnp | grep -E ':80|nginx' || echo "nginx not listening"
df -h /srv/web
curl -v --max-time 3 http://localhost/ 2>&1 | head -n 20
rm -f /srv/web/filler
systemctl restart nginx
curl -sf http://localhost/ && echo OK
```

The consolidated triage flow: status, journal, sockets, storage, probe, then remediate and verify recovery.

Test it / key takeaway: After remediation curl returns the Capstone page with a 200, confirming the full build and the status/journal/sockets/storage/probe triage loop.

Quick Command Reference

The essential tools and terms per exam domain (see each lab for full usage):

Domain	Lab	Key commands & concepts
1	1. Boot Process & Filesystem Hierarchy	/etc/os-release, GRUB2, /proc/cmdline, initramfs, lsinitramfs, FHS, systemd targets, usrmerge
1	2. Kernel Modules & Device Management	lsmod, modinfo, modprobe, insmod/rmmod, depmod, dmesg, lspci/lsusb/lscpu/lsblk, dracut vs mkinitramfs
1	3. Storage with LVM, Partitions & Filesystems	losetup, parted/GPT, pvcreate/vgcreate/lvcreate, mkfs.ext4/xfs/btrfs, blkid/UUID, /etc/fstab, mount hardening (nodev/nosuid/noexec), lvextend/resize2fs/xfs_growfs
1	4. Network Configuration	ip -br link/address/route, /etc/nsswitch.conf, /etc/resolv.conf, getent, dig/nslookup, netplan generate, ss, tcpdump/nmap
1	5. Shell Operations & Text Processing	export/env/unset, login vs interactive shells, redirection > >> < << <<< | tee, grep/cut/sort/uniq/wc, sed/awk, tr/xargs, shell scripting
1	6. Backup & Restore	tar (czf/xzf/tzf), cpio, dd, ddrescue, rsync --delete, sha256sum -c, zcat/zgrep/gzip -l, gzip/bzip2/xz tradeoffs
1	7. Virtualization with QEMU/KVM & libvirt	QEMU vs KVM, /dev/kvm vs TCG, qemu-img (qcow2/raw), qemu-img convert/resize, qcow2 snapshots, virsh list/net-list, bridged/NAT/host-only networks
2	8. File & Directory Management	ls -la / stat / file, find predicates (-name, -mtime, -size), locate & updatedb, lsof, diff & sdiff, hard vs symbolic links, block/character device files
2	9. Local Account & Group Management	/etc/passwd, /etc/shadow, /etc/group, /etc/skel, useradd / adduser / usermod -aG, chpasswd, passwd -S, chage, chsh, getent, id, UID>=1000 user vs UID<1000 system vs service (-r), userdel -r / groupdel
2	10. Processes, Jobs & Scheduling	ps auxf / pstree / pidstat / mpstat, /proc/<PID> and lsof -p, process states (R,S,D,Z,T) & strace, jobs: &, Ctrl+Z, bg, fg, nohup, disown,

		nice / renice priority, kill / pkill / killall signals, at, cron, anacron
2	11. Software & Package Management	apt update/install/remove/purge/autoremove, apt-cache policy, dpkg -l / -L, signed-by GPG keyrings, dnf / rpm -q / rpm -V, pip, npm -g, cargo, update-alternatives, snap & flatpak sandboxes
2	12. Service Management with systemd	systemctl list-units / list-unit-files / --failed, start/stop/restart/reload, enable/disable, mask/unmask, custom .service (Type=oneshot) + daemon-reload, .timer units (OnUnitActiveSec), .mount units, systemd-analyze blame / critical-chain, hostnamectl / timedatectl / resolvectl, journalctl -u
2	13. Containers with Docker & Podman	docker.io vs podman (daemonless/rootless), docker pull/run -d -p --name, docker ps, logs / exec -it / inspect / stats, bind-mount volumes (-v host:container:ro), Dockerfile: FROM/RUN/COPY/USER/ENTRYPOINT/CMD, docker network create (built-in DNS), rootless Podman & --privileged risks
3	14. AAA: sudo, PAM, and Polkit	sudo / visudo / sudoers.d, PAM stacks (auth, account, session), auditctl / ausearch, journalctl / rsyslog / logrotate, pkexec (Polkit), SSSD / LDAP / Kerberos
3	15. Firewalls: ufw, firewalld, iptables, nftables, ipset	ufw allow/deny/enable, firewalld zones and rich rules, iptables / nftables chains, ipset hash:ip, SNAT / DNAT / MASQUERADE, conntrack states (NEW/ESTABLISHED/RELATED)
3	16. OS Hardening: Permissions, SELinux, SSH, Fail2ban	chmod setuid/setgid/sticky bit, umask / chattr immutability, POSIX ACLs (setfacl/getfacl), SELinux chcon/restorecon/audit2allow, OpenSSH hardening (PermitRootLogin), fail2ban, SUID hunting (find -perm -4000)
3	17. Account Hardening	/etc/login.defs (PASS_MAX_DAYS), pam_pwquality / pwscore, chage aging, passwd -l/-u, pam_faillock lockout, nologin / rbash restricted shells, Google Authenticator TOTP MFA, HIBP k-anonymity API
3	18. Cryptography: GPG, LUKS, OpenSSL, WireGuard	GPG batch key generation, encrypt/sign, LUKS2 (cryptsetup, loopback), OpenSSL genrsa / req -x509, hashing vs HMAC, WireGuard Curve25519 keys, TLS version/cipher inspection

		(s_client)
3	19. Compliance, Auditing, and File Integrity	AIDE file-integrity baseline, rkhunter rootkit scan, debsums / rpm -V package verification, lynis system audit, nmap -sV service detection, shred secure deletion, OpenSCAP / CIS benchmarks / banners
4	20. Infrastructure as Code with Ansible	Ansible, inventory, playbook, idempotency, handlers/templates, ansible-galaxy, GitOps, cloud-init
4	21. Bash Scripting	shebang, parameter expansion, arrays, test operators [[ ]], getopts, IFS, set -euo pipefail, shellcheck
4	22. Python for System Administration	venv/pip, core types (bool/int/float/str/list/dict), type hints, os.walk, json module, requests, black/flake8/PEP 8
4	23. Git Version Control	git init/config, branch/merge --no-ff, conflict resolution, stash/tag, reset --soft/rebase, bare remote, push/pull --rebase, .gitignore
4	24. Responsible AI Use for Linux Administrators	verify-before-paste, secret redaction, local vs hosted LLMs, shellcheck, API key via env var, AI usage policy, data governance, attribution
5	25. Monitoring Concepts and Tools	SLA vs SLO vs SLI, SNMP OID/MIB walk, Prometheus scrape config, node_exporter metrics endpoint, promtool check rules, webhook alerting, curl health checks, journalctl log aggregation
5	26. Troubleshooting Hardware, Storage, and OS Issues	ENOSPC / df -h, inode exhaustion / df -i, grub-mkconfig validation, runaway process / top -bn1, failed systemd unit, broken PATH recovery, memory leak / ps rss / free
5	27. Troubleshooting Network Connectivity	ip addr / ip route baseline, resolv.conf / dig, default gateway, ufw port block, MTU / ping -M do, duplicate IP / arping, dummy interface link toggle, tcpdump / mtr / traceroute / nmap
5	28. Troubleshooting Security Issues	SELinux context / chcon / restorecon, AVC denials / ausearch / audit2allow, POSIX ACL / getfacl / setfacl, CA trust / update-ca-certificates, pam_faillock lockout, apt repo / TLS s_client, unattended-upgrades
5	29. Troubleshooting Performance Issues	baseline: uptime/free/vmstat/mpstat, stress-ng CPU spike / load average, memory pressure / vmstat si-so, fio disk storm /

		iostat -x, D-state / ping jitter, pidstat context switches, ps --sort triage script
5	30. Capstone: End-to-End Server Build and Triage	hardened SSH / narrow sudo, LVM: pvcreate/vgcreate/lvcreate, systemd drop-in override, ufw default-deny allowlist, systemd timer health check, git-versioned /etc/nginx, fault injection and triage loop

Support & Assessment

At the assessment step, follow this order:

1. TRAQOM — scan the TRAQOM QR code on the LMS and complete the survey.

2. Assessment Digital Attendance.

3. Assessment — Written Assessment (SAQ) + Practical Performance (PP), open book.

4. Submit the assessment answers on the LMS.

5. Sign the Assessment Summary Record.

Courseware and the assessment are on the LMS — https://lms-tms.tertiaryinfotech.com/. For support: enquiry@tertiaryinfotech.com · +65 6100 0613 · www.tertiarycourses.com.sg