

Lab Guide

Five exercises across five infrastructure artifacts: nmap results, Kubernetes pod specs, Docker Compose files, nginx configs, and AWS IAM policies — find the misconfigurations that break the castle walls.

Authorized Use Only

All configurations are **synthetic**. Do not run scans against the IPs or domains referenced. The misconfiguration patterns you learn here apply to real infrastructure only under authorized assessment.

Prerequisites

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all five files into a folder named `infra-lab/`
- Familiarity with: TCP/IP, Docker, Kubernetes pods, AWS IAM, nginx directives
- Optional: `kubesecc`, `trivy config`, `checkov`, `cloudsploit` for cross-checking

Lab Scenario

AcmeCorp runs a hybrid environment: some legacy VMs, a Kubernetes cluster, Docker containers, an nginx reverse proxy, and AWS workloads. You have pulled config artifacts from all five. Your mission: map the attack surface, identify every misconfiguration, and build the shortest chain from "unauthenticated internet" to "cluster admin."

Download Lab Files

Save all five files into a folder named `infra-lab/` before starting the exercises.

↓ `nmap-scan.txt` ~7 KB

↓ `k8s-pod-spec.txt` ~6 KB

↓ `docker-compose.txt` ~6 KB

↓ `nginx-config.txt` ~5 KB

↓ `aws-iam-policy.txt` ~6 KB

Exercise 1 — Nmap Scan Analysis

Open `nmap-scan.txt`. Nmap output is dense and contains many clues about running services and OS fingerprints.

Task 1.1 — Build the host inventory

Below is nmap scan output. For each host, produce:

1. IP address and reverse DNS (if present).
2. Operating system (best guess from fingerprint).
3. Every open port + service + version.
4. Any service with a known CVE (name the CVE).
5. Any default or weak service (telnet, ftp anonymous, SNMP with public community, SMB v1, RDP without NLA).
6. The single most attackable host and why.
7. The shortest path from "unauthenticated network access" to "shell on one host".

Also: flag any host that has an unusual combination (e.g., a Windows box exposing SSH, a Linux box exposing RDP, a database directly exposed to the internet).

[paste nmap-scan.txt]

✓ **Self-check:** You should have one host ranked as "primary target" with specific version info and a concrete next-step command.

Exercise 2 — Kubernetes Pod Spec Audit

Open `k8s-pod-spec.txt`. Kubernetes misconfigurations are a common root cause of cluster breakouts.

Task 2.1 — Find the breakout path

Below is a Kubernetes Pod/Deployment spec. Audit for:

1. `privileged: true` or `allowPrivilegeEscalation: true`
2. `hostNetwork`, `hostPID`, `hostIPC`, `hostPath` volumes
3. Missing `securityContext` (`runAsUser`, `runAsNonRoot`, `fsGroup`)
4. Missing resource limits (DoS risk) or requests
5. `capabilities`: any `CAP_SYS_ADMIN`, `NET_ADMIN`, `CAP_SYS_PTRACE`?
6. Service accounts with `cluster-admin` or overly broad RBAC
7. Secrets mounted as env vars (leaked in logs) vs. volumes
8. `imagePullPolicy`: Always vs. pinned digests (supply chain)
9. Image tags using `:latest`
10. Any obvious container escape primitives

For each finding:

- CIS Kubernetes Benchmark control ID (if applicable)
- Exploit scenario
- Fix (show the corrected YAML block)

Finally: given this pod's access level, what could an attacker do if they achieved RCE inside the container? Rank impact.

[paste `k8s-pod-spec.txt`]

✓ **Self-check:** You should be able to name at least one escape primitive and one fix that the client could apply without breaking the app.

Exercise 3 — Docker Compose Review

Open `docker-compose.txt`. Compose files often encode secrets, trust assumptions, and network boundaries.

Task 3.1 — Identify the weak service

Below is a `docker-compose.yml` file. Review as a red team operator:

1. Every service: image, exposed ports, mounted volumes, env vars, network, capabilities.
2. Any secret in env (passwords, tokens, API keys, DB URIs).
3. Any service with volumes mounting host-sensitive paths (`/`, `/var/run/docker.sock`, `/etc`, `/home`).
4. Any network that places a database or admin UI on the same bridge as an internet-facing service.
5. Any service using `:latest` or an unverified image source.
6. Any command/entrypoint that runs as root unnecessarily.
7. Any healthcheck or restart policy that could be abused.

Then:

- Which service is the weakest (easiest to compromise)?
- Which service is the most valuable (contains secrets/data)?
- Build the shortest pivot from weak to valuable.

[paste `docker-compose.txt`]

✓ **Self-check: You should find at least one secret and one dangerous volume mount.**

Exercise 4 — Nginx Configuration Audit

Open `nginx-config.txt`. Nginx sits in front of everything — misconfigurations often lead to path traversal, SSRF, or auth bypass.

Task 4.1 — Find the routing flaws

Below is an nginx configuration. Audit for:

1. Alias/location off-by-one bugs that enable path traversal (e.g., ``location /static` + `alias /data/static`` without trailing slashes).
2. Missing `auth_basic` or `auth_request` on sensitive locations.
3. `proxy_pass` to internal services without SSRF protection (user-controlled input reaching upstream URL).
4. Open redirect via `rewrite`/`return` with user input.
5. Missing security headers (HSTS, CSP, X-Frame-Options).
6. Large `client_max_body_size` enabling DoS or upload abuse.
7. SSL/TLS: weak protocols (TLSv1.0/1.1), weak ciphers, missing OCSP stapling.
8. Any server block that exposes `/server-status`, `/nginx_status`, or `.git/.env` without restriction.
9. `proxy_cache` configurations that could leak authenticated data.

Output every finding with severity and the corrected directive.

[paste nginx-config.txt]

✓ Self-check: Nginx configs almost always have at least one missing auth check or path traversal primitive. You should have at least one exploit URL you could send to test it.

Exercise 5 — AWS IAM Policy Audit

Open `aws-iam-policy.txt`. AWS IAM is the control plane for cloud access — a bad policy is the cloud equivalent of "admin:admin".

Task 5.1 — Find the privilege escalation path

Below is an AWS IAM policy. Audit for:

1. Any Action with "*" (star) that is not explicitly narrowed by Resource and Condition.
2. Resource "*" combined with write/create/delete actions.
3. `iam:PassRole` to a role the principal should not control.
4. `iam:CreateAccessKey` / `UpdateAssumeRolePolicy` /

- AttachRolePolicy / PutRolePolicy – classic privesc primitives.
5. lambda:CreateFunction + iam:PassRole = priv-esc via Lambda.
 6. ec2:RunInstances + iam:PassRole = priv-esc via instance profile.
 7. Missing Condition blocks (aws:SourceIp, aws:MultiFactorAuthPresent).
 8. sts:AssumeRole to broadly-trusted principals.
 9. Any Deny that can be bypassed (e.g., Deny on a specific resource while Allow is broader).

For each finding:

- Map to a Rhino Security Labs privesc technique or MITRE ATT&CK.
- Show the exact AWS CLI commands an attacker would run.
- Provide the least-privilege corrected policy.

[paste aws-iam-policy.txt]

✓ **Self-check: AWS policies with wildcards almost always allow privilege escalation. You should be able to name a specific escalation path and the exact CLI commands to execute it.**

Final Exercise — Build the Infrastructure Kill Chain

Task 6.1 — Cross-layer attack chain

Here are five infrastructure audits from the same environment.
Build a single attack chain:

1. Initial foothold (from nmap, which port/service is entry?).
2. Web-to-internal pivot (via nginx misconfig?).
3. Container escape (via docker-compose or k8s spec?).
4. Cluster/host lateral movement.
5. Cloud privilege escalation (via IAM policy?).
6. Domain/root/cluster-admin endgame.

Then output:

- An ordered kill chain (step 1 -> step N) with specific commands.
- A prioritized fix list: which 3 changes would break the chain in the most places?
- A hardening checklist the ops team could implement in one week.

Format as markdown, max 1500 words.

✓ Self-check: Your kill chain should cross at least 3 of the 5 artifacts. Infrastructure attacks almost always pivot between network, container, and cloud layers — if yours stays in one layer, you are not thinking like a cloud red teamer.

Reflection

- Which artifact revealed the most misconfigurations per kilobyte?
- How did the AI handle the AWS IAM policy? IAM is notoriously hard for humans — where did the AI shine or fail?
- What is the biggest risk of using AI to audit production IaC before deployment?
- If you could only audit ONE artifact type before a pen test, which would you pick and why?

What's Next

Lesson 08 moves from finding vulnerabilities to **weaponizing them** — using AI to turn crash dumps and CVE details into working exploit code.