Chapter 11 — Supply-Chain Controls

11.1 Current State and Scope

As-built supply-chain posture is minimal but deliberately clean so far: everything installed to date (Docker, Compose, Prometheus, nginx, certbot) came from **Ubuntu official repos or the official Grafana apt repo** — no curl-pipe-bash installs, no third-party PPAs, no unpinned container images yet, because no application containers exist yet. This chapter sets the controls **before** the application build starts pulling npm/crates/PyPI dependencies and publishing images — retrofitting provenance onto an already-polluted dependency tree is an order of magnitude harder than starting gated. Scope: container images, OS packages, language dependencies (Node/pnpm, Rust/Cargo, Python), the artifact registry, and the build pipeline that connects them. Controlling SEC-001 requirement: INF-03 (repository controls) plus the supply-chain family imported into KO-001; AI-02 intersects where MCP tool containers are concerned (§6.4).

11.2 Registry Policy

GitHub Container Registry (GHCR) under the SBS GitHub organization is the sole artifact registry:

- One registry, private visibility, org-owned — same ownership/revocability principle as §2.1 and §10.3. No Docker Hub publishing; no images pulled from Docker Hub **except** the official library base images enumerated in the allowlist (§11.3).
- Pull access from droplets via a registry-scoped read-only token stored in Vault (`secret/platform/ghcr/`), rotated on the 90-day static schedule (§8.4). Push access belongs to CI only — no human pushes images; an image that wasn't built by the pipeline doesn't exist as far as deployment is concerned.
- The §3.4 egress allowlist admits exactly: GHCR endpoints, Ubuntu archive/security mirrors, the Grafana apt repo, and the language registries (registry.npmjs.org, crates.io/static.crates.io, pypi.org/files.pythonhosted.org) **from build contexts only** — production droplets pull finished images, never raw dependencies; the language registries are unreachable from prod runtime (a compromised container cannot `pnpm add` its way to a payload).

11.3 Image Provenance

- **Base image allowlist**, pinned by digest, reviewed quarterly: official `node:<LTS>-slim`, `rust:<pinned>-slim` (build stage only), `python:3.x-slim`, `nginx:stable`, `postgres` / `mongo` /

`redis` official images for the §6.5 interim, `hashicorp/vault`. Nothing outside the list without a PR touching the allowlist file (CODEOWNERS: security reviewer per §10.4).
- **Digest pinning everywhere:** `FROM node:22-slim@sha256:…` in Dockerfiles; compose files reference deployed images by digest, not floating tags (`:latest` is prohibited and CI-linted). Tag-based human readability is preserved by *also* tagging, but the digest is what deploys — a registry-side tag repoint cannot swap what prod runs.
- **Multi-stage builds** — build stages (compilers, package managers) never ship; runtime stages carry the application and its production dependencies only. Runtime images run as non-root UID, no shell where the base permits (`distroless`-style trimming is a later optimization, not a Day-1 gate).
- **Rebuild cadence:** weekly CI rebuild of all images against updated base digests + `apt upgrade` layer, so CVE patching is a pipeline event, not an emergency (§INF-05 patching applies to images as much as hosts). Trivy scan (§11.6) gates the result.

11.4 Dependency Pinning

Per-ecosystem, all enforced in CI (a lockfile that isn't verified is documentation, not control):

Ecosystem	Pinning mechanism	CI enforcement
Node / pnpm	`pnpm-lock.yaml` committed; `packageManager` field + corepack pins pnpm itself	`pnpm install --frozen-lockfile` — any drift fails the build; `pnpm audit` report attached
Rust / Cargo	`Cargo.lock` committed (binaries); `rust-toolchain.toml` pins the compiler	`cargo build --locked`; `cargo audit` (RustSec)
Python	`requirements.txt` with `--hash=sha256:…` per package (pip-compile generated)	`pip install --require-hashes`; `pip-audit`
OS packages	Versions asserted in the Ansible base role for security-relevant packages	Drift surfaces in the nightly `--check` run (§10.5)

Dependency **updates** are PRs like any other change (Renovate bot, grouped weekly, auto-PR but never auto-merge) — the update path goes through the same review + CI gauntlet, so a poisoned upstream release sits in a diff a human looks at, not in a silent nightly pull. New top-level dependencies require a one-line justification in the PR body; transitive bloat is reviewed via the lockfile diff.

11.5 Artifact Signing and Verification

- **Commits:** signed commits required on both repos (already a §10.3 branch-protection condition) — SSH-key signing acceptable; the key inventory rides the per-person identity work of item 17.
- **Images:** CI signs every pushed image with **cosign keyless** (Sigstore, GitHub OIDC identity) — the signature attests "built by this workflow, from this commit, in this repo," which is the provenance claim that matters. No long-lived signing key to protect or rotate (the §8 lifecycle problem cosign keyless exists to delete).

- **Verification at deploy:** the deploy step (§6.7 / §10.4) runs `cosign verify` against the expected workflow identity **before** `compose up`; an unsigned or wrongly-attested image fails the deploy. This closes the loop: registry compromise alone cannot inject a runnable image, because the attacker must also produce a valid Sigstore attestation from the protected CI workflow.
- **SBOM:** CI generates SPDX SBOMs (syft) per image, attached as a cosign attestation and archived to `sbs-dash-backups` — when the next log4shell-class event lands, "are we exposed" is a query against stored SBOMs, not an archaeology project.

11.6 Pipeline Integrity and Scanning

The build pipeline is itself supply chain:

- GitHub Actions pinned by **action digest** (not `@v4` floating tags) — the same rule applied to the tooling as to the images; third-party actions from a reviewed allowlist only.
- Workflow permissions default `contents: read`; the push job alone gets `packages: write` + `id-token: write` (for cosign OIDC). No self-hosted runners at this fleet size — GitHub-hosted runners keep runner compromise off the threat sheet at the cost of trusting GitHub, which the design already does (repo, registry).
- **Trivy** scans every built image (CI gate: fail on CRITICAL with fix available; HIGH warns and files an issue) and re-scans the deployed digest set nightly against fresh CVE data — the nightly catches vulnerabilities published *after* the image shipped, feeding MON-06 (vulnerability scanning) with the §11.3 weekly rebuild as the remediation vehicle.
- gitleaks (already items 47/§10.4) covers the secrets half of pipeline hygiene.

11.7 Verification

Gates (LH-SBS-INST-001): base-image allowlist file exists, all Dockerfiles resolve `FROM` digests within it (lint); `:latest` absent from every compose/Dockerfile (grep gate); frozen-lockfile builds pass and a deliberately drifted lockfile fails (negative test); cosign verify passes on a pipeline image and **fails** on a manually-pushed unsigned test image (negative test); deploy refuses the unsigned image end-to-end; SBOM attestation present for every deployed digest; Trivy CI gate demonstrated on a seeded-CVE test image; language registries unreachable from prod runtime (egress probe, §3.4); Actions digests pinned (workflow lint); prod running-set digests ⊆ registry signed-set (drift cross-check with §10.5).

Revision #1
Created 2026-07-18 11:49:21 UTC by SBS Admin
Updated 2026-07-18 11:51:35 UTC by SBS Admin