# **A Comprehensive Guide to Containerizing Linux-Native Security Assessment Tools**

## **1. Introduction**

Integrating security scanning into modern software development lifecycles, particularly within CI/CD pipelines, necessitates consistent, reproducible, and isolated execution environments. Containerization using Docker offers an effective solution for packaging Linux-native security assessment tools, ensuring their dependencies and configurations are managed reliably across different systems. This guide provides a comprehensive approach to containerizing a suite of essential security tools: Syft, Grype, Semgrep, FindSecBugs, Bandit, ESLint, SonarQube Scanner CLI, ScanCode Toolkit, LicenseFinder, Sigstore (Cosign), in-toto, YARA, and Capa.

For each tool, this report details recommended base Docker images, installation procedures, configuration strategies, command-line usage patterns within containers, approaches for processing output, volume mapping requirements for data persistence and access, considerations for resource constraints, common containerization challenges with solutions, health check implementations, and example Dockerfile and docker-compose definitions. The aim is to provide actionable guidance for DevOps and Security teams looking to leverage containerization for enhanced security posture automation.

## **2. Core Concepts in Security Tool Containerization**

Before diving into specific tools, several core concepts underpin effective containerization for security scanning:

* **Base Image Selection**: Choosing a minimal, secure base image (e.g., scratch, Alpine Linux, distroless, Chainguard Wolfi) is crucial. Security tools, especially, benefit from a reduced attack surface within their own execution environment. Static binaries (like those often produced by Go, used in Syft, Grype, Cosign) are particularly well-suited for scratch or minimal distroless images, often only requiring ca-certificates for network operations.1 Tools requiring interpreters (Python, Node.js, JVM) need appropriate runtime base images.
* **Dependency Management**: Containerization isolates tool dependencies. Installation should occur within the Dockerfile using package managers (apk, apt, pip, npm, gem) or by downloading pre-compiled binaries. Multi-stage builds are recommended to keep final images lean, separating build-time dependencies from runtime requirements.
* **Configuration Management**: Configuration can be passed via environment variables, command-line arguments, or mounted configuration files (e.g., .grype.yaml, .semgrepignore, sonar-project.properties). Mounting config files via volumes offers flexibility without rebuilding the image.
* **Volume Mapping**: Essential for interacting with the host system or persisting data. Common use cases include mounting source code directories for scanning, Docker sockets for image analysis, configuration files, output directories for reports, and cache directories (e.g., for vulnerability databases) to improve performance.3
* **Resource Constraints**: Security analysis can be resource-intensive (CPU, memory). Defining appropriate limits and reservations in container orchestration (Docker Compose, Kubernetes) is vital for stability and preventing resource starvation on the host.5
* **Security Considerations**: Run containers as non-root users whenever possible. Minimize privileges. Ensure the scanner image itself is regularly updated and scanned for vulnerabilities. Consider signed, verifiable images (e.g., using Sigstore, offered by Chainguard 2).

## **3. Tool-Specific Containerization Guides**

This section provides detailed containerization guidance for each specified tool.

### **3.1 Syft**

Syft is a CLI tool and Go library for generating Software Bills of Materials (SBOMs) from container images and filesystems, often used as input for vulnerability scanners like Grype.8

* **Base Docker Image Recommendation**:
  + Due to its static Go binary nature, Syft is ideal for minimal images.
  + scratch: The official anchore/syft image often uses scratch as a base, adding only necessary components like ca-certificates.1 This minimizes the scanner's own attack surface.
  + cgr.dev/chainguard/syft:latest: Based on the minimal Wolfi Linux "undistro", offering features like daily builds, build-time SBOMs, and Sigstore signatures for enhanced provenance.2 Note minor differences in ENTRYPOINT, CMD, and WORKDIR compared to the official image.2
  + Multi-stage build: If building from source, use a Go build image in the first stage and copy the static binary to a scratch or minimal runtime image in the final stage.
* **Installation Dependencies and Commands**:
  + Primary dependency is the Syft binary itself.
  + ca-certificates: Required for HTTPS communication (e.g., pulling images from registries, version checks) if using scratch base.1
  + *Dockerfile (using pre-built binary)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG SYFT\_VERSION=latest # Or specify a version like v1.2.3  
    FROM alpine:3.19 as certs  
    FROM scratch  
    COPY --from=certs /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-certificates.crt  
    ARG SYFT\_VERSION  
    # Adjust architecture (amd64, arm64) as needed  
    ADD https://github.com/anchore/syft/releases/download/v${SYFT\_VERSION}/syft\_${SYFT\_VERSION}\_linux\_amd64.tar.gz /syft.tar.gz  
    RUN tar -xzf /syft.tar.gz -C /usr/local/bin/ syft && rm /syft.tar.gz  
    # Official image uses /syft, Chainguard uses /usr/bin/syft  
    ENTRYPOINT ["/usr/local/bin/syft"]  
    # Official image WORKDIR is /tmp, Chainguard has none. Setting one for clarity.  
    WORKDIR /scan-target  
    CMD ["--help"]
  + The official anchore/syft Dockerfile content can be inspected on Docker Hub layers 10 or potentially GitHub 1 (though direct access to raw file failed 11). It confirms the scratch base, copying ca-certificates.crt, setting WORKDIR /tmp, and ENTRYPOINT ["/syft"].
* **Configuration File Requirements**:
  + Syft is primarily configured via CLI flags.
  + Supports configuration file (.syft.yaml, .syft/config.yaml, ~/.syft.yaml, <XDG\_CONFIG\_HOME>/syft/config.yaml).12
  + Configuration precedence: CLI flags > Environment Variables (SYFT\_\*) > Config File > Defaults.12
  + Key options include output, quiet, file-metadata.selection, catalogers selection/exclusion, registry credentials, etc..8
  + Mount the config file using a volume if needed: -v./my-syft.yaml:/.syft.yaml.
* **Command-Line Usage Patterns**:
  + Scan container image: docker run --rm -v /var/run/docker.sock:/var/run/docker.sock anchore/syft <image\_name>:<tag>.2
  + Scan directory: docker run --rm -v $(pwd):/scan-target anchore/syft dir:/scan-target.2
  + Specify output format: -o <format> (e.g., syft-json, spdx-json, cyclonedx-xml).13
  + Specify output file: -o <format>=/scan-target/sbom.json.13
  + Select catalogers: --select-catalogers +java or -go-module-binary-cataloger.8
  + Exclude paths: --exclude./node\_modules/\*\*.8
* **Output Processing Approach**:
  + Default output is a human-readable table (syft-table).13
  + Common machine-readable formats: syft-json (native, most detail), spdx-json, spdx-tag-value, cyclonedx-json, cyclonedx-xml, github-json.9
  + Output includes package name, version, type, licenses, locations, etc..9
  + JSON output is often piped to Grype for vulnerability scanning.14
* **Volume Mapping Strategy**:
  + /var/run/docker.sock (or Podman socket): To allow Syft to interact with the container daemon for image scanning.8 Mount read-only if possible.
  + /scan-target (or /src, /app): Mount the host directory containing source code or files to be scanned.2
  + /output: Mount a host directory to write report files to, if using -o <format>=<file>.
  + /config: If mounting custom registry credentials (e.g., config.json).8
  + /tmp: Official image sets WORKDIR /tmp 10; temporary files might be written here. Chainguard image does not set a WORKDIR.2
* **Resource Constraints**:
  + Generally lightweight for typical container images.
  + Memory usage can increase significantly when scanning large filesystems or images with many files/layers, especially if file content analysis or extensive cataloging is enabled.15
  + Indexing large disks can be memory and time-intensive.15
  + Recommend setting baseline memory limits (e.g., 512MB - 2GB) and CPU limits (e.g., 1 CPU), adjusting based on typical scan targets. Monitor usage.
* **Common Containerization Issues and Solutions**:
  + *Issue*: Cannot connect to Docker daemon. *Solution*: Ensure the Docker socket (/var/run/docker.sock) is correctly mounted into the container and permissions allow access.
  + *Issue*: Cannot pull private images. *Solution*: Mount the Docker config.json containing credentials (-v ~/.docker/config.json:/config/config.json -e DOCKER\_CONFIG=/config) 8 or use Kubernetes secrets.
  + *Issue*: High memory usage on large filesystems. *Solution*: Increase container memory limits. Use --exclude to skip irrelevant large directories. Investigate if specific catalogers are causing high usage. Ongoing work aims to improve whole-filesystem scanning performance.15
  + *Issue*: Permission denied accessing mounted volumes. *Solution*: Ensure the container user (often root in official image, check specific image) has read access to mounted source directories and write access to output/cache volumes if used. Run container with host user ID (--user $(id -u):$(id -g)) cautiously, or adjust host directory permissions.
* **Health Check Implementation**:
  + Simple check: CMD ["syft", "version"].
  + Basic functionality check: CMD ["syft", "--help"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=3s CMD syft version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Minimal, based on official image principles)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG SYFT\_VERSION=latest # Or specify a version like v1.2.3  
    FROM alpine:3.19 as certs  
    FROM scratch  
    COPY --from=certs /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-certificates.crt  
    ARG SYFT\_VERSION  
    # Fetch Syft binary for linux/amd64  
    ADD https://github.com/anchore/syft/releases/download/v${SYFT\_VERSION}/syft\_${SYFT\_VERSION}\_linux\_amd64.tar.gz /syft.tar.gz  
    # Extract only the syft binary to /syft path (matching official image)  
    RUN tar -xzf /syft.tar.gz -C / syft && rm /syft.tar.gz  
    # Create and set working directory (matching official image)  
    WORKDIR /tmp  
    ENTRYPOINT ["/syft"]  
    CMD ["--help"]  
      
    # Add OCI labels (optional but good practice)  
    ARG BUILD\_DATE  
    ARG BUILD\_VERSION  
    ARG VCS\_REF  
    ARG VCS\_URL  
    LABEL org.opencontainers.image.created=$BUILD\_DATE  
    LABEL org.opencontainers.image.title="syft"  
    LABEL org.opencontainers.image.description="CLI tool and library for generating a Software Bill of Materials from container images and filesystems"  
    LABEL org.opencontainers.image.source=$VCS\_URL  
    LABEL org.opencontainers.image.revision=$VCS\_REF  
    LABEL org.opencontainers.image.vendor="Anchore, Inc."  
    LABEL org.opencontainers.image.version=$BUILD\_VERSION  
    LABEL org.opencontainers.image.licenses="Apache-2.0"
  + *docker-compose.yaml (scanning local directory)*:  
    YAML  
    services:  
     syft-scan:  
     # Use official image, Chainguard, or build the Dockerfile above  
     image: anchore/syft:latest  
     # Mount current directory into the container's WORKDIR or a dedicated scan path  
     volumes:  
     -.:/scan-target # Mount current directory into /scan-target  
     # Command to scan the mounted directory and output SPDX JSON to stdout  
     command: dir:/scan-target -o spdx-json  
     # Optional: Define resource limits  
     deploy:  
     resources:  
     limits:  
     cpus: '1.0'  
     memory: 1G  
     reservations:  
     cpus: '0.5'  
     memory: 256M

The nature of Syft as a static Go binary lends itself extremely well to containerization using minimal base images like scratch or Wolfi.1 This directly minimizes the attack surface of the scanner itself, a critical consideration for a tool involved in security assessments. The choice between the official anchore/syft image and the Chainguard cgr.dev/chainguard/syft variant presents a trade-off: the official image provides direct alignment with the source project, while the Chainguard image offers enhanced, verifiable provenance through build-time SBOMs and Sigstore signatures, built on their security-focused Wolfi base.2 Organizations prioritizing stricter supply chain security standards (like SLSA) might lean towards the Chainguard option, despite minor differences in default entrypoint or working directory configurations.2

### **3.2 Grype**

Grype is an open-source vulnerability scanner for container images, filesystems, and SBOMs, often used in conjunction with Syft.3

* **Base Docker Image Recommendation**:
  + Similar to Syft, Grype is a Go application, suitable for minimal images.
  + scratch: The official anchore/grype image uses scratch with ca-certificates copied in.17
  + cgr.dev/chainguard/grype:latest: Recommended for a secure, minimal, ready-to-use image based on Wolfi.7 Chainguard also offers FIPS-compliant variants (grype-fips).19 Differences in ENTRYPOINT, CMD, WORKDIR exist compared to the official image.7
  + Multi-stage build: If building from source, use a Go build image and copy the static binary to scratch or a minimal base.
* **Installation Dependencies and Commands**:
  + Requires the Grype binary.
  + Requires ca-certificates for DB updates and potentially registry interactions.17
  + Requires access to the Grype vulnerability database. This can be downloaded on first run or updated explicitly (grype db update).
  + *Dockerfile (using pre-built binary, similar to Syft)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG GRYPE\_VERSION=latest # Or specify a version like v1.2.3  
    FROM alpine:3.19 as certs  
    FROM scratch  
    COPY --from=certs /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-certificates.crt  
    ARG GRYPE\_VERSION  
    # Adjust architecture (amd64, arm64) as needed  
    ADD https://github.com/anchore/grype/releases/download/v${GRYPE\_VERSION}/grype\_${GRYPE\_VERSION}\_linux\_amd64.tar.gz /grype.tar.gz  
    RUN tar -xzf /grype.tar.gz -C / grype && rm /grype.tar.gz  
    # Official image uses /grype, Chainguard uses /usr/bin/grype  
    ENTRYPOINT ["/grype"]  
    # Official image WORKDIR is /tmp, Chainguard has none.  
    WORKDIR /tmp  
    CMD ["--help"]  
      
    # Add OCI labels (optional but good practice)  
    ARG BUILD\_DATE  
    ARG BUILD\_VERSION  
    ARG VCS\_REF  
    ARG VCS\_URL  
    LABEL org.opencontainers.image.created=$BUILD\_DATE  
    LABEL org.opencontainers.image.title="grype"  
    LABEL org.opencontainers.image.description="A vulnerability scanner for container images and filesystems"  
    LABEL org.opencontainers.image.source=$VCS\_URL  
    LABEL org.opencontainers.image.revision=$VCS\_REF  
    LABEL org.opencontainers.image.vendor="Anchore, Inc."  
    LABEL org.opencontainers.image.version=$BUILD\_VERSION  
    LABEL org.opencontainers.image.licenses="Apache-2.0"
  + The official anchore/grype Dockerfile content can be inferred from layer details 17 or potentially viewed on GitHub 18 (raw access failed 20). It confirms scratch base, ca-certificates, WORKDIR /tmp, and ENTRYPOINT ["/grype"].
* **Configuration File Requirements**:
  + Primarily configured via CLI flags.
  + Supports configuration file (.grype.yaml, .grype/config.yaml, ~/.grype.yaml, <XDG\_CONFIG\_HOME>/grype/config.yaml).3 Mount via volume (-v./.grype.yaml:/.grype.yaml).
  + Key options: fail-on-severity, ignore rules (for vulnerability ID, fix state, package details), output, file, db settings (cache dir, update URL, auto-update), exclude paths, vex-documents.3
  + Environment variables (GRYPE\_\*) override config file values.3
* **Command-Line Usage Patterns**:
  + Scan container image (daemon access): docker run --rm -v /var/run/docker.sock:/var/run/docker.sock anchore/grype <image\_name>:<tag>.3
  + Scan container image (direct registry pull): docker run --rm anchore/grype registry:<image\_name>:<tag> 21 or simply docker run --rm anchore/grype <image\_name>:<tag>.14
  + Scan directory: docker run --rm -v $(pwd):/scan-target anchore/grype dir:/scan-target.14
  + Scan SBOM: docker run --rm -v $(pwd)/sbom.json:/scan/sbom.json anchore/grype sbom:/scan/sbom.json.14
  + Pipe from Syft: docker run --rm anchore/syft <image> -o json | docker run --rm -i anchore/grype.14
  + Fail on severity: --fail-on <severity> (e.g., high, critical).3
  + Specify output format: -o <format> (e.g., table, json, cyclonedx-json, sarif).3
  + Update DB explicitly: docker run --rm -v grype\_db\_cache:/root/.cache/grype anchore/grype db update.
* **Output Processing Approach**:
  + Default format is table for human readability.23
  + json format provides the most comprehensive data.23
  + cyclonedx-json / cyclonedx-xml formats for SBOM-based vulnerability exchange.14
  + sarif format for integration with security dashboards and tools.3
  + Custom templates via -o template -t <template\_file>.21
  + External tools can process Grype's output, e.g., grype2html for reports 24, DefectDojo for vulnerability management 25, or custom parsers like grype-parser.26
  + The internal database schema evolved (v5 to v6), primarily affecting storage and indexing efficiency, while aiming to maintain output compatibility.27
* **Volume Mapping Strategy**:
  + /var/run/docker.sock: For scanning images via the Docker daemon.3 Mount read-only if possible.
  + /scan-target (or /src, /app): Mount host directory containing source code, files, or SBOMs to be scanned.14
  + /root/.cache/grype (default path, configurable via db.cache-dir or GRYPE\_DB\_CACHE\_DIR): **Crucial** for persisting the vulnerability database cache between runs. Use a named volume.3
  + /config: If mounting a .grype.yaml configuration file.3
  + /output: If writing report to a file using -o <format> --file <path> or -o <format>=<path>.
  + /tmp: Default WORKDIR in the official image.17 Temporary files might be used here.
* **Resource Constraints**:
  + Network access required for initial DB download and updates, unless DB is pre-loaded or manually managed. DB download size is ~65MB with v6 schema.27
  + Memory usage depends on the size of the scan target (image layers, filesystem files, SBOM size). Scanning large targets can consume significant memory.15
  + CPU usage occurs during analysis and matching.
  + Recommend setting memory limits (e.g., 1GB - 4GB, monitor and adjust) and CPU limits (e.g., 1-2 CPUs). Insufficient resources can lead to slow scans or crashes.
* **Common Containerization Issues and Solutions**:
  + *Issue*: Slow scans due to repeated DB downloads. *Solution*: Mount a persistent named volume to the cache directory (/root/.cache/grype or configured path).3 Ensure the volume is writable by the container user. Pre-warm the cache if needed.
  + *Issue*: False positives or irrelevant vulnerabilities reported. *Solution*: Utilize ignore rules in .grype.yaml to filter by CVE ID, fix state, package details, etc..3 Use --fail-on <severity> to focus on critical issues.23 Consider --only-fixed or --only-notfixed.3 Explore OpenVEX integration for external justification data.3 Check for tool comparison differences (Trivy/Clair might have different FP rates).29
  + *Issue*: Cannot access private container registries. *Solution*: Mount Docker config.json or use Kubernetes secrets, similar to Syft.3
  + *Issue*: Permission errors writing to cache volume. *Solution*: Ensure the volume mount has correct permissions for the container user (often root in anchore/grype, check image specifics 3). Consider chown on the host volume or running the container with specific user/group IDs (use cautiously).
  + *Issue*: Outdated vulnerability data. *Solution*: Ensure db.auto-update is enabled (default) 3 or run grype db update periodically. Ensure the container has network access to the DB update URL.
* **Health Check Implementation**:
  + Simple check: CMD ["grype", "version"].
  + Check DB status: CMD ["grype", "db", "status"]. This command exits non-zero if the DB is missing or too old (based on db.validate-age and db.max-allowed-built-age config 3), making it suitable for a health check.
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=1h --timeout=15s --start-period=2m CMD grype db status | | exit 1 (Allows initial DB download time).
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (using Chainguard, pre-warming cache)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    FROM cgr.dev/chainguard/grype:latest as downloader  
    # Download DB in a separate stage  
    RUN grype db update  
      
    FROM cgr.dev/chainguard/grype:latest  
    ENV GRYPE\_DB\_CACHE\_DIR=/cache/grype/db  
    # Copy the pre-downloaded cache  
    COPY --from=downloader /home/nonroot/.cache/grype/db /cache/grype/db  
    # Ensure correct ownership if image runs non-root (Chainguard often does)  
    # RUN chown -R nonroot:nonroot /cache/grype  
    # USER nonroot # If needed  
    WORKDIR /scan-target  
    ENTRYPOINT ["grype"]  
    CMD ["--help"]
  + *docker-compose.yaml (Syft -> Grype pipeline)*:  
    YAML  
    services:  
     generate-sbom:  
     image: anchore/syft:latest  
     volumes:  
     - /var/run/docker.sock:/var/run/docker.sock # If scanning image  
     -./app-code:/scan-target # Or mount code  
     - sbom\_data:/output # Shared volume for SBOM  
     # Scan image 'my-app:latest', output syft-json to shared volume  
     command: my-app:latest -o syft-json=/output/sbom.json  
     # Or scan directory: command: dir:/scan-target -o syft-json=/output/sbom.json  
      
     scan-sbom:  
     image: anchore/grype:latest  
     depends\_on:  
     - generate-sbom  
     volumes:  
     - sbom\_data:/input # Mount shared volume  
     - grype\_db\_cache:/root/.cache/grype # Persistent DB cache  
     # Scan the SBOM from the shared volume, fail on high severity  
     command: sbom:/input/sbom.json --fail-on high -o table  
     deploy:  
     resources:  
     limits:  
     cpus: '1.0'  
     memory: 2G  
     reservations:  
     cpus: '0.5'  
     memory: 512M  
      
    volumes:  
     sbom\_data: # Volume to share SBOM between Syft and Grype  
     grype\_db\_cache: # Persistent volume for Grype DB cache

Grype's utility is significantly amplified by its vulnerability database. Consequently, the primary operational challenge in containerizing Grype revolves around managing this database efficiently. Downloading the database (~65MB with the v6 schema 27) on every execution is impractical for frequent scans, especially in CI/CD environments. Therefore, implementing a persistent volume strategy for the cache directory (/root/.cache/grype by default 3) is paramount for performance.4 Furthermore, Grype is frequently paired with Syft in a pipeline: Syft generates the SBOM, and Grype consumes it.14 This decoupling allows for faster vulnerability rescans as only the relatively small SBOM needs to be processed by Grype, rather than the entire container image or filesystem, whenever the vulnerability database is updated.14 Configuration via .grype.yaml provides essential control for tailoring scans, particularly through ignore rules and severity thresholds, which are vital for managing findings effectively in complex projects.3

### **3.3 Semgrep**

Semgrep is a fast, static analysis tool for finding bugs, detecting vulnerabilities, and enforcing code standards across various languages.30

* **Base Docker Image Recommendation**:
  + semgrep/semgrep:latest: The official image is recommended.32 It's based on alpine:3.19 and includes Python 3.11, bash, jq, and curl.33 This provides a ready-to-use environment with common utilities.
  + semgrep/semgrep:latest-nonroot: A non-root variant is available and recommended for enhanced security.34 Layer details show it adds a semgrep user (UID 1000).35
  + If custom rules require additional Python packages or other dependencies not in the official image, use semgrep/semgrep:latest as a base and add installation steps, or use a standard Python image (e.g., python:3.11-alpine) and install Semgrep via pip.
* **Installation Dependencies and Commands**:
  + The official Docker image comes with Semgrep CLI pre-installed.33
  + Requires Python >= 3.8 if installing manually.36
  + Installation via pip: python3 -m pip install semgrep.37
  + Dependencies like bash, jq, curl are included in the official image but might be removed in the future; install explicitly if relied upon heavily in custom scripts.33
  + *Dockerfile (extending official image with custom dependencies)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    # Use the non-root image for better security  
    FROM semgrep/semgrep:latest-nonroot  
    # Switch to root temporarily to install packages  
    USER root  
    # Example: Install git if needed for fetching rules or code  
    RUN apk add --no-cache git  
    # Switch back to the non-root user  
    USER semgrep  
    WORKDIR /src  
    # Entrypoint is likely inherited or set to 'semgrep'  
    # CMD might default to 'semgrep --help' or similar
* **Configuration File Requirements**:
  + Rule configuration is primarily done via the --config CLI flag, pointing to:
    - Semgrep Registry rulesets (e.g., p/ci, p/owasp-top-ten, r/my-org.ruleset).37
    - Local YAML rule files (my-rule.yaml).31
    - Directories containing YAML rule files (/rules).37
    - URLs to rule files.
    - auto to automatically select rules based on project content (requires login/metrics).37
  + .semgrepignore: File in the project root (or specified path) to exclude files/directories from scanning, using gitignore syntax.41 Mount via volume if needed.
  + Semgrep AppSec Platform Token: Set via SEMGREP\_APP\_TOKEN environment variable for features like private rules, managed scans, and semgrep ci integration.30
* **Command-Line Usage Patterns**:
  + Scan local directory with registry rules: docker run --rm -v $(pwd):/src semgrep/semgrep:latest-nonroot scan --config p/ci /src.37
  + Scan with local custom rules: docker run --rm -v $(pwd):/src -v./my-rules:/rules semgrep/semgrep:latest-nonroot scan --config /rules /src.46
  + Combine registry and local rules: ... scan --config p/python --config /rules /src.37
  + Use semgrep ci for CI integration (diff-aware scanning, platform reporting): docker run --rm -v $(pwd):/src -e SEMGREP\_APP\_TOKEN=$TOKEN semgrep/semgrep:latest-nonroot ci.40 Requires Git history (.git dir mounted or available).
  + Specify output format: --json, --sarif, --text, --gitlab-sast.37
  + Fail on findings (for CI gating): Default semgrep ci exits 1 on blocking findings.43 semgrep scan exits 0 by default, use --error to exit 1 on findings.37
  + Login to AppSec Platform (if not using token env var): docker run --rm -it -v ~/.semgrep:/home/semgrep/.semgrep semgrep/semgrep:latest-nonroot login. Mount .semgrep directory to persist credentials. Passing SEMGREP\_APP\_TOKEN is generally preferred for CI.44
* **Output Processing Approach**:
  + Default output is human-readable text to stdout.
  + --json: Machine-readable JSON format.37 Schema available in semgrep/semgrep-interfaces.
  + --sarif: Standard Static Analysis Results Interchange Format, widely supported by security dashboards (GitHub Security, etc.) and IDEs.37
  + Specific CI formats: --gitlab-sast, --gitlab-secrets.37
  + Other formats: --junit-xml, --emacs, --vim.37
  + JSON output can be processed with tools like jq.33
* **Volume Mapping Strategy**:
  + /src (or /app, /code): Mount the host directory containing the source code to be scanned.42 Mount read-only if possible.
  + /rules: Mount a host directory containing custom Semgrep rule files (YAML).46
  + /.semgrepignore: Mount a custom ignore file if not at the root of /src.
  + /home/semgrep/.semgrep (for non-root image) or /root/.semgrep (for root image): Mount to persist login credentials across container runs if using semgrep login.
  + /output: Mount a host directory if writing report files using options like --sarif-output /output/report.sarif.
* **Resource Constraints**:
  + Memory: Can be significant, especially with deep analysis (interfile) or large codebases.5 OOM errors (exit code -9, -11) are possible.5 Allocate sufficient memory (e.g., 2GB - 8GB+, monitor usage).6
  + CPU: Analysis can be CPU-intensive. Scales with the number of jobs (--jobs, defaults to core count).37 Limit container CPUs as needed.
  + Network: Required to download registry rulesets (--config p/...) or if using Semgrep AppSec Platform integration.
* **Common Containerization Issues and Solutions**:
  + *Issue*: High Memory Usage / OOM Errors. *Solution*: Increase container memory limit. Run single-threaded (--jobs 1). Use --max-memory LIMIT to restrict memory per file/rule. Exclude large/problematic files via .semgrepignore. Optimize or disable slow/memory-intensive rules.5
  + *Issue*: Slow Scans. *Solution*: Profile with --time 51; identify and potentially exclude slow rules/files.5 Use .semgrepignore effectively. Adjust timeouts (--timeout, --timeout-threshold, --interfile-timeout).5 Ensure sufficient CPU/memory allocation.
  + *Issue*: Authentication failure with Semgrep AppSec Platform. *Solution*: Ensure SEMGREP\_APP\_TOKEN is correctly set as an environment variable in the container.44 Verify the token is valid and has the necessary permissions. Check network connectivity to semgrep.dev.
  + *Issue*: Cannot find custom rules. *Solution*: Ensure the volume mount for rules is correct (-v./local-rules:/rules) and the --config /rules path matches the mount point inside the container. Check file permissions.
  + *Issue*: Permission denied accessing source code or writing output. *Solution*: Run the container using the host user's UID/GID (--user $(id -u):$(id -g)) especially when using the non-root image.35 Alternatively, adjust host directory permissions (less ideal). Ensure output volumes are writable.
  + *Issue*: semgrep ci fails due to missing git history. *Solution*: Ensure the .git directory is included in the source code volume mount.
* **Health Check Implementation**:
  + Simple check: CMD ["semgrep", "--version"].
  + Basic functionality check: CMD ["semgrep", "scan", "--help"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=5s CMD semgrep --version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (based on non-root, adding git)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    FROM semgrep/semgrep:latest-nonroot  
      
    USER root  
    # Install git, needed for 'semgrep ci' or rules fetched from git repos  
    RUN apk add --no-cache git  
    USER semgrep  
      
    WORKDIR /src  
    # Inherit ENTRYPOINT/CMD from base or override if needed  
    # ENTRYPOINT ["semgrep"]  
    # CMD ["scan", "--config", "p/ci", "/src"]
  + *docker-compose.yaml (scanning local code with custom rules and platform token)*:  
    YAML  
    services:  
     semgrep-scan:  
     image: semgrep/semgrep:latest-nonroot  
     user: "${UID:-1000}:${GID:-1000}" # Run as host user for permissions  
     volumes:  
     -.:/src # Mount current directory (including.git if using 'ci')  
     -./semgrep-rules:/etc/semgrep-rules:ro # Mount custom rules read-only  
     environment:  
     # Pass token for Semgrep AppSec Platform features/reporting  
     - SEMGREP\_APP\_TOKEN=${SEMGREP\_APP\_TOKEN}  
     # Scan with custom rules and a registry ruleset, output SARIF  
     command: scan --config /etc/semgrep-rules --config p/owasp-top-ten --sarif -o /src/semgrep-report.sarif /src  
     # Or use 'ci' command for CI integration: command: ci  
     deploy:  
     resources:  
     limits:  
     cpus: '2.0'  
     memory: 4G  
     reservations:  
     cpus: '1.0'  
     memory: 1G

Containerizing Semgrep offers significant advantages for consistent static analysis. The choice between standalone scanning (semgrep scan) and integrated scanning (semgrep ci) heavily influences the container setup. Using semgrep ci or features of the Semgrep AppSec Platform necessitates handling authentication, typically via the SEMGREP\_APP\_TOKEN environment variable 44, and often requires access to Git metadata. Rule management is another key aspect; rules can be fetched dynamically from the Semgrep Registry, mounted as local files/directories, or baked into the container image for a self-contained artifact.37 Given that Semgrep performs deeper code analysis compared to dependency or SBOM scanners, resource management (memory, CPU, timeouts) is critical 5, and configuring appropriate container limits is essential for reliable operation, especially in CI environments. Using the official non-root image variant enhances security.35

### **3.4 FindSecBugs (via SpotBugs)**

FindSecBugs is a plugin for SpotBugs, a static analysis tool for finding bugs in Java code. It specifically focuses on security vulnerabilities.52 Containerizing FindSecBugs involves containerizing SpotBugs and ensuring the plugin is correctly loaded.

* **Base Docker Image Recommendation**:
  + No official Docker image exists specifically for FindSecBugs or SpotBugs.52
  + Requires a Java Runtime Environment (JRE) or JDK. SpotBugs 4.x requires JRE 11 or later.52
  + Recommended base images:
    - eclipse-temurin:11-jre-alpine or eclipse-temurin:17-jre-alpine: Minimal JRE based on Alpine.
    - eclipse-temurin:11-jdk-focal or eclipse-temurin:17-jdk-focal: Full JDK based on Ubuntu, if compilation or other JDK tools are needed within the container.
  + If integrating with build tools, consider official maven or gradle images which include a JDK.57 Use multi-stage builds to keep the final image smaller if the build tool isn't needed at runtime.
* **Installation Dependencies and Commands**:
  + **Java Runtime**: Must be present in the base image (JRE 11+).
  + **SpotBugs**: Download the distribution .tgz or .zip from GitHub releases.56 Extract it within the image (e.g., to /opt/spotbugs).
  + **FindSecBugs Plugin**: Download the findsecbugs-plugin-\*.jar file from Maven Central 53 or GitHub releases. Place this JAR inside the plugin directory of the extracted SpotBugs installation (e.g., /opt/spotbugs/plugin/).
  + *Dockerfile Steps*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG SPOTBUGS\_VERSION=4.8.6 # Check latest stable SpotBugs version  
    ARG FSB\_PLUGIN\_VERSION=1.12.0 # Check latest FindSecBugs plugin version  
    ARG JAVA\_VERSION=17 # Or 11  
      
    # Use a builder stage to download and extract  
    FROM eclipse-temurin:${JAVA\_VERSION}-jdk-alpine as builder  
    ARG SPOTBUGS\_VERSION  
    ARG FSB\_PLUGIN\_VERSION  
    WORKDIR /tmp  
    # Install wget and tar  
    RUN apk add --no-cache wget tar  
    # Download and extract SpotBugs  
    RUN wget https://github.com/spotbugs/spotbugs/releases/download/${SPOTBUGS\_VERSION}/spotbugs-${SPOTBUGS\_VERSION}.tgz && \  
     tar -xzf spotbugs-${SPOTBUGS\_VERSION}.tgz && \  
     rm spotbugs-${SPOTBUGS\_VERSION}.tgz  
    # Download FindSecBugs plugin JAR into SpotBugs plugin directory  
    RUN wget https://repo1.maven.org/maven2/com/h3xstream/findsecbugs/findsecbugs-plugin/${FSB\_PLUGIN\_VERSION}/findsecbugs-plugin-${FSB\_PLUGIN\_VERSION}.jar \  
     -O /tmp/spotbugs-${SPOTBUGS\_VERSION}/plugin/findsecbugs-plugin-${FSB\_PLUGIN\_VERSION}.jar  
      
    # Final stage with JRE only  
    FROM eclipse-temurin:${JAVA\_VERSION}-jre-alpine  
    ARG SPOTBUGS\_VERSION  
    ENV SPOTBUGS\_HOME=/opt/spotbugs  
    # Copy extracted SpotBugs with plugin from builder stage  
    COPY --from=builder /tmp/spotbugs-${SPOTBUGS\_VERSION} ${SPOTBUGS\_HOME}  
    # Add SpotBugs bin to PATH  
    ENV PATH="${SPOTBUGS\_HOME}/bin:${PATH}"  
    # Set workdir for scans  
    WORKDIR /scan-target  
    # Default entrypoint to run SpotBugs CLI  
    ENTRYPOINT ["spotbugs", "-textui"]  
    # Default command shows help  
    CMD ["-help"]
* **Configuration File Requirements**:
  + SpotBugs/FindSecBugs configuration is often managed via build tool plugins (Maven pom.xml 59, Gradle build.gradle 60). The plugin configuration specifies effort level, thresholds, includes/excludes, and plugin activation.
  + SpotBugs Filter Files: XML files used to suppress specific warnings or select specific bug types/classes/methods for analysis.61 Specified via -include <file> or -exclude <file> CLI options or build tool config.59 Mount these files via volumes.
  + SpotBugs Analysis Properties: Can be set via -property name=value CLI option.63
* **Command-Line Usage Patterns** (using SpotBugs CLI):
  + Basic scan with FindSecBugs enabled (assuming plugin JAR is in plugin dir): docker run --rm -v $(pwd):/scan-target my-findsecbugs-image -effort:max -output /scan-target/report.xml /scan-target/my-app.jar
  + Using exclude filter: ... -exclude /scan-target/config/spotbugs-exclude.xml....61
  + Specifying project file (less common for CI): ... -project /scan-target/my-project.fbp....63
  + Explicitly listing plugin (usually not needed if placed in plugin dir): ... -pluginList ${SPOTBUGS\_HOME}/plugin/findsecbugs-plugin-\*.jar....63
  + Target is typically compiled Java code (.jar files or directories containing .class files).
* **Output Processing Approach**:
  + SpotBugs generates reports in XML (default for -textui), HTML, or SARIF formats.
  + XML output is machine-readable and can be parsed by CI/CD platforms (e.g., Jenkins with SpotBugs plugin) or custom scripts.
  + HTML output provides a human-readable report.
  + SARIF output allows integration with modern security dashboards.
* **Volume Mapping Strategy**:
  + /scan-target: Mount the host directory containing the compiled Java code (.jar files, .class directories) to be scanned.
  + /config: Mount a host directory containing SpotBugs filter files (e.g., spotbugs-exclude.xml).61
  + /output: Mount a host directory to write the generated report files (XML, HTML, SARIF).
  + If building within the container:
    - Mount source code: /src.
    - Mount Maven cache: /root/.m2.
    - Mount Gradle cache: /root/.gradle.
* **Resource Constraints**:
  + Java Application: Requires sufficient JVM heap memory (-Xmx). Static analysis of large codebases can be memory-intensive.64
  + CPU: Analysis can be CPU-bound.
  + Configure JVM heap size via JAVA\_OPTS environment variable passed to the container or directly in the spotbugs script/command if supported.
  + Set container memory limits (e.g., 2GB - 8GB+, monitor usage) and CPU limits.
* **Common Containerization Issues and Solutions**:
  + *Issue*: java.lang.OutOfMemoryError during scan.64 *Solution*: Increase the container's memory limit. Increase the JVM maximum heap size using -Xmx (e.g., docker run -e JAVA\_OPTS="-Xmx4g"...). Adjust SpotBugs effort level (-effort:min uses less memory).63
  + *Issue*: FindSecBugs plugin not detected/loaded. *Solution*: Verify the findsecbugs-plugin-\*.jar is correctly placed in the ${SPOTBUGS\_HOME}/plugin directory within the container image. Ensure SpotBugs and plugin versions are compatible. Use -pluginList option for explicit loading if needed.63 Check SpotBugs startup logs for plugin loading messages (use -debug flag).
  + *Issue*: Incorrect paths in reports or filter files. *Solution*: Ensure paths used in commands and filter files correspond to the paths *inside* the container (e.g., /scan-target/com/mycompany/..., not host paths).
  + *Issue*: ClassNotFoundException or similar analysis errors. *Solution*: Ensure all required dependency JARs for the application being scanned are provided to SpotBugs (usually via -auxclasspath option if not running via build tool which handles classpath).
* **Health Check Implementation**:
  + Simple check: CMD ["spotbugs", "-version"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=5s CMD spotbugs -version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Using SpotBugs CLI, based on previous example)*: (See Dockerfile example under Installation section above)
  + *docker-compose.yaml (Scanning pre-built JAR with exclude filter)*:  
    YAML  
    services:  
     findsecbugs-scan:  
     build:  
     context:. # Directory containing the Dockerfile from Installation section  
     args:  
     # Optional: Specify versions if not using defaults in Dockerfile  
     # SPOTBUGS\_VERSION: 4.8.6  
     # FSB\_PLUGIN\_VERSION: 1.12.0  
     # JAVA\_VERSION: 17  
     volumes:  
     # Mount the compiled application JAR into the container's workdir  
     -./my-app/target/my-app.jar:/scan-target/my-app.jar:ro  
     # Mount a configuration directory containing the exclude filter  
     -./config:/scan-target/config:ro  
     # Mount an output directory for the report  
     -./reports:/scan-target/reports  
     # Run SpotBugs CLI: exclude issues from config file, output XML report  
     command: >  
     -exclude /scan-target/config/spotbugs-exclude.xml  
     -xml:withMessages  
     -output /scan-target/reports/spotbugs-report.xml  
     /scan-target/my-app.jar  
     deploy:  
     resources:  
     limits:  
     memory: 4G # Adjust based on project size  
     reservations:  
     memory: 1G

Containerizing FindSecBugs fundamentally means containerizing SpotBugs and managing the FindSecBugs plugin dependency.52 Since FindSecBugs operates as a plugin, the container must provide a compatible SpotBugs environment (JRE 11+, SpotBugs installation) and ensure the plugin JAR is correctly located within the SpotBugs plugin directory.56 The analysis targets compiled Java bytecode (.class files or .jar archives), meaning the container needs access to these build artifacts. This often leads to two main containerization approaches: 1) Include build tools (Maven/Gradle) in the image to compile the code and then run the analysis, resulting in a larger, more complex image. 2) Use a simpler JRE-based image that scans pre-compiled artifacts mounted via volumes, requiring a separate build step outside the scanner container. The latter approach generally leads to a leaner, more focused scanner image. Managing Java's memory requirements (-Xmx) is a common challenge, necessitating proper JVM configuration within the container environment.64

### **3.5 Bandit**

Bandit is a tool designed to find common security issues in Python code by processing files, building Abstract Syntax Trees (ASTs), and running plugins against the AST nodes.65

* **Base Docker Image Recommendation**:
  + ghcr.io/pycqa/bandit/bandit:latest: This is the official image provided by the PyCQA organization, available on GitHub Container Registry.66 It's multi-architecture and signed with Sigstore.66 Inspecting its layers or Dockerfile (if available) would reveal the base image (likely a standard Python image like python:3.x-slim or python:3.x-alpine).
  + python:3.x-slim or python:3.x-alpine: Standard Python images can be used as a base if building manually or extending functionality. Choose a Python version compatible with Bandit (>=3.9 required).66 Alpine is smaller, Slim (Debian-based) might have better compatibility for some dependencies.
  + Other community images exist (e.g., opensorcery/bandit 65, secfigo/bandit 68), but the official ghcr.io image is generally preferred for up-to-date versions and official support.
* **Installation Dependencies and Commands**:
  + Python (>= 3.9).66
  + Bandit package itself.
  + Optional extras for specific features: toml (for pyproject.toml config), baseline (for baseline feature), sarif (for SARIF output).69
  + *Dockerfile (using official Python base)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG PYTHON\_VERSION=3.11 # Use a supported version >= 3.9  
    FROM python:${PYTHON\_VERSION}-slim  
      
    # Install Bandit and desired extras  
    RUN pip install --no-cache-dir bandit[toml,sarif]  
      
    # Create a non-root user and group  
    RUN groupadd --gid 1001 bandit && \  
     useradd --uid 1001 --gid 1001 --create-home bandit  
      
    # Set working directory  
    WORKDIR /scan-target  
      
    # Switch to non-root user  
    USER bandit  
      
    # Set entrypoint to bandit command  
    ENTRYPOINT ["bandit"]  
    # Default command shows help  
    CMD ["--help"]
* **Configuration File Requirements**:
  + Bandit can be configured using a YAML file (default name .bandit.yml or specified via -c/--config) or via pyproject.toml under the [tool.bandit] section (requires bandit[toml] extra and -c pyproject.toml).70
  + Configuration files can specify tests to run (tests), tests to skip (skips), paths to exclude (exclude\_dirs), plugin configurations, severity/confidence levels, etc..71
  + Can also use .bandit INI file (legacy, requires -r flag).71
  + Mount config files via volumes: -v./my-bandit-config.yaml:/scan-target/.bandit.yml.
* **Command-Line Usage Patterns**:
  + Scan a directory recursively: docker run --rm -v $(pwd):/scan-target ghcr.io/pycqa/bandit/bandit -r /scan-target.65
  + Specify config file: ... bandit -c /scan-target/custom-config.yaml -r /scan-target.
  + Filter by severity/confidence: -l / -ll / -lll or --severity-level {low,medium,high} and -i / -ii / -iii or --confidence-level {low,medium,high}.69
  + Specify output format: -f {csv,json,html,sarif,xml,yaml,txt}.69
  + Specify output file: -o /scan-target/report.json.69
  + Exclude paths: --exclude./tests,./venv.70
  + Run specific tests/profiles: -t B101,B102 or -p ShellInjection.69
  + Skip specific tests: -s B601.71
* **Output Processing Approach**:
  + Default output is human-readable text (screen) to stdout.
  + Common machine-readable formats: json, csv, xml, yaml, sarif.72 SARIF is useful for security dashboard integration.
  + HTML format provides a browseable report.72
  + Custom formats possible using --msg-template.72
* **Volume Mapping Strategy**:
  + /scan-target (or /src, /code): Mount the host directory containing the Python source code to be scanned.65 Mount read-only if possible.
  + /config: Mount a host directory containing configuration files (.bandit.yml, pyproject.toml).
  + /output: Mount a host directory to write report files to if using -o.
* **Resource Constraints**:
  + AST generation and analysis can consume memory and CPU, particularly on large codebases or complex files.
  + Generally less resource-intensive than full compilation or deep dataflow analysis tools.
  + Recommend setting moderate resource limits (e.g., 1-2GB memory, 1-2 CPUs) and monitoring. Specific benchmarks for Bandit itself were not found, but general principles apply.73
* **Common Containerization Issues and Solutions**:
  + *Issue*: Bandit cannot find source files or config files. *Solution*: Verify volume mounts are correct (-v host/path:/container/path). Ensure the paths used in the bandit command correspond to the paths *inside* the container. Check file permissions within the container, especially if running as non-root.
  + *Issue*: Incorrect Python version or missing dependencies for the scanned code (if Bandit plugins rely on project specifics, though less common). *Solution*: Ensure the base image's Python version matches project requirements if necessary. Install project dependencies within the container if plugins require them (less ideal, increases image size).
  + *Issue*: Permission denied accessing mounted volumes or writing reports. *Solution*: Run the container with the host user's UID/GID (--user $(id -u):$(id -g)). Ensure the non-root user inside the container (if used) has appropriate permissions on mounted volumes.
* **Health Check Implementation**:
  + Simple check: CMD ["bandit", "--version"].
  + Basic functionality check: CMD ["bandit", "-h"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=3s CMD bandit --version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (using official image)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    # Use the official multi-arch image from GHCR  
    FROM ghcr.io/pycqa/bandit/bandit:latest  
      
    # Image likely runs as root by default, or has a specific user.  
    # If it runs root, consider adding a non-root user:  
    # RUN groupadd --gid 1001 bandit && \  
    # useradd --uid 1001 --gid 1001 --create-home bandit  
    # USER bandit  
      
    WORKDIR /scan-target  
    # Entrypoint/CMD are likely already set to run bandit  
    # ENTRYPOINT ["bandit"]  
    # CMD ["--help"]
  + *docker-compose.yaml (scanning local code with config file)*:  
    YAML  
    services:  
     bandit-scan:  
     image: ghcr.io/pycqa/bandit/bandit:latest  
     # Run as host user to avoid permission issues on volume mounts  
     user: "${UID:-1000}:${GID:-1000}"  
     volumes:  
     # Mount current directory containing Python code  
     -.:/scan-target:ro  
     # Mount config file (optional)  
     -./config/bandit.yaml:/scan-target/.bandit.yml:ro  
     # Mount output directory (optional)  
     -./reports:/scan-target/reports  
     # Scan recursively, use config file, output SARIF report  
     command: >  
     -r  
     -c /scan-target/.bandit.yml  
     -f sarif  
     -o /scan-target/reports/bandit-report.sarif  
     /scan-target  
     deploy:  
     resources:  
     limits:  
     cpus: '1.0'  
     memory: 2G  
     reservations:  
     cpus: '0.5'  
     memory: 512M

Bandit's focus on Python AST analysis makes it relatively straightforward to containerize.65 The official ghcr.io/pycqa/bandit/bandit image is the recommended starting point, providing a pre-configured environment.66 Configuration is flexible, supporting both dedicated YAML files and integration into pyproject.toml 71, allowing teams to choose the method that best fits their project structure. Volume mapping is primarily needed for accessing the target source code and potentially configuration files or report outputs. Running as a non-root user, either by using a non-root base image or configuring one, is a key security consideration.

### **3.6 ESLint**

ESLint is a highly pluggable linter tool for identifying and reporting on patterns in JavaScript code, enforcing code style, and finding potential errors.75

* **Base Docker Image Recommendation**:
  + No single official ESLint Docker image is maintained by the ESLint team itself. Several community/vendor images exist (e.g., pipelinecomponents/eslint 76, onaci/eslint 77), but their maintenance and base images vary.
  + Recommended approach: Use an official Node.js base image corresponding to the project's Node version requirement (ESLint requires Node.js ^18.18.0, ^20.9.0, or >=21.1.0 75).
    - node:20-alpine or node:22-alpine: Minimal Node.js image based on Alpine.
    - node:20-slim or node:22-slim: Debian-based minimal Node.js image.
  + Use multi-stage builds if project dependencies are needed only for specific ESLint plugins during linting but not for the final application runtime.
* **Installation Dependencies and Commands**:
  + Node.js (compatible version).75
  + npm, yarn, pnpm, or bun package manager (usually included with Node.js image).
  + ESLint package (eslint).
  + Any ESLint plugins (eslint-plugin-\*), shareable configurations (eslint-config-\*), and custom parsers (@typescript-eslint/parser, etc.) used by the project. These should ideally be listed in the project's package.json.
  + *Dockerfile (using Node base, installing project deps)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG NODE\_VERSION=20 # Use project's Node version  
    FROM node:${NODE\_VERSION}-alpine  
      
    # Set working directory  
    WORKDIR /app  
      
    # Copy package files  
    COPY package.json package-lock.json\* yarn.lock\* pnpm-lock.yaml\*./  
      
    # Install project dependencies (including eslint, plugins, configs)  
    # Choose the appropriate command for your package manager  
    # RUN npm ci  
    # RUN yarn install --frozen-lockfile  
    RUN pnpm install --frozen-lockfile  
    # RUN bun install --frozen-lockfile  
      
    # Copy the rest of the source code  
    COPY..  
      
    # Create a non-root user (node user often exists in official images)  
    # RUN addgroup -g 1001 eslint && adduser -u 1001 -G eslint -s /bin/sh -D eslint  
    USER node # Use the existing node user  
      
    # Set entrypoint to run eslint via npx/yarn dlx/pnpm dlx/bunx  
    # This uses the locally installed ESLint version  
    ENTRYPOINT ["pnpm", "dlx", "eslint"]  
    # Default command lints the current directory  
    CMD ["."]
* **Configuration File Requirements**:
  + ESLint configuration file: eslint.config.js (flat config, recommended), eslint.config.mjs, eslint.config.cjs.78 Older formats (.eslintrc.\*, package.json#eslintConfig) are still supported but deprecated.
  + The config file defines rules, plugins, parsers, environments, globals, and file overrides.78
  + Requires plugins and configs specified in the file to be installed (usually via package.json).
  + The container needs access to this configuration file, typically by copying it into the image (COPY eslint.config.js.) or mounting the project directory.
* **Command-Line Usage Patterns**:
  + Lint specific files/directories: docker run --rm -v $(pwd):/app my-eslint-image src/ tests/
  + Use --fix to automatically fix fixable problems: ... eslint. --fix.80
  + Use --fix-dry-run to see potential fixes without applying them.81
  + Specify config file (if not auto-detected): -c /app/custom.eslint.config.js.81
  + Specify output format: -f <format> (e.g., stylish (default), json, junit, html, gitlab).76
  + Specify output file: -o /app/reports/eslint-report.json.81
  + Lint specific extensions: --ext.js,.jsx,.ts.81
* **Output Processing Approach**:
  + Default output (stylish) is human-readable console output.
  + json format provides structured data for machine processing.
  + junit format is common for CI test reporting.
  + gitlab format integrates with GitLab Code Quality reports.76
  + Other formatters can be installed as plugins.
* **Volume Mapping Strategy**:
  + /app (or /usr/src/app, /code): Mount the host directory containing the JavaScript/TypeScript source code and package.json/eslint.config.js. This is the primary volume needed.
  + /output or /app/reports: Mount a host directory if writing report files using -o.
* **Resource Constraints**:
  + Node.js application. Memory and CPU usage depend on the number of files, complexity of code, and the rules/plugins enabled.
  + Parsing large files or complex rules (especially those involving type checking via @typescript-eslint) can be resource-intensive.
  + Recommend setting reasonable limits (e.g., 1-4GB memory, 1-2 CPUs) and monitoring.
* **Common Containerization Issues and Solutions**:
  + *Issue*: ESLint cannot find plugins/configs specified in eslint.config.js. *Solution*: Ensure npm install (or equivalent) was run successfully within the Dockerfile *after* copying package.json and lock files. Verify plugins are listed as dependencies in package.json. If using pnpm, ensure workspace setup is correct within the container if applicable.75
  + *Issue*: Incorrect Node.js version. *Solution*: Use a Node.js base image matching the project's required version specified in package.json (engines field).
  + *Issue*: Slow linting times. *Solution*: Ensure sufficient memory/CPU. Utilize ESLint caching (--cache flag, map .eslintcache file via volume). Profile ESLint run to identify slow rules/plugins. Exclude irrelevant files/directories using ignores in eslint.config.js or a .eslintignore file.
  + *Issue*: Permission errors accessing mounted code or writing cache/reports. *Solution*: Run the container as the host user (--user $(id -u):$(id -g)) or ensure the node user (or other non-root user) in the container has appropriate permissions on the mounted volumes.
* **Health Check Implementation**:
  + Simple check: CMD ["node", "-e", "require('eslint'); console.log('OK');"] (checks if eslint module is importable).
  + Check CLI version: CMD ["pnpm", "dlx", "eslint", "--version"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=10s CMD pnpm dlx eslint --version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (using Node.js base, pnpm)*: (See Dockerfile example under Installation section above)
  + *docker-compose.yaml (running lint with auto-fix)*:  
    YAML  
    services:  
     eslint-lint:  
     build:. # Assumes Dockerfile is in the current directory  
     user: "${UID:-1000}:${GID:-1000}" # Run as host user  
     volumes:  
     # Mount project code into the container's WORKDIR  
     -.:/app  
     # Optional: Mount node\_modules from host to avoid reinstalling in container  
     # -./node\_modules:/app/node\_modules  
     # Optional: Mount cache directory  
     -./.eslintcache:/app/.eslintcache  
     # Lint current directory, attempt fixes, use cache  
     command:. --fix --cache  
     deploy:  
     resources:  
     limits:  
     cpus: '1.5'  
     memory: 2G  
     reservations:  
     cpus: '0.5'  
     memory: 512M

Containerizing ESLint involves setting up a Node.js environment with the correct ESLint version, plugins, and configurations as defined by the project being scanned.75 The most robust approach is typically to use an official Node.js base image and run the project's package manager (npm install, yarn install, pnpm install) inside the container build process. This ensures all necessary dependencies specified in package.json (including ESLint itself, plugins, configs, parsers) are available.82 Volume mapping is primarily needed to provide the container access to the source code and potentially persist the ESLint cache file (.eslintcache) for improved performance on subsequent runs. Running the container as a non-root user (like the node user often available in official Node images) is recommended practice.83

### **3.7 SonarQube Scanner CLI**

The SonarScanner CLI is the standard command-line scanner for sending analysis reports to a SonarQube server or SonarCloud.4

* **Base Docker Image Recommendation**:
  + sonarsource/sonar-scanner-cli:latest: This is the official image provided by SonarSource.84
  + The image base varies; recent versions (e.g., 5.x scanner, image tag 10.x) use amazoncorretto:17-al2023 85, indicating Amazon Linux 2023 with Corretto JDK 17. Older versions might use different bases.
  + The image includes the necessary Java runtime and the SonarScanner CLI tool itself.
  + It also includes git, tar, and nodejs.85
* **Installation Dependencies and Commands**:
  + Java Runtime (JRE): Included in the official image. Required by the scanner.
  + SonarScanner CLI: Included in the official image.
  + Project-specific build tools (Maven, Gradle,.NET SDK, etc.) are *not* typically included in the scanner image itself. The scanner usually analyzes code *after* it has been built.
  + *Dockerfile (if extending official image, e.g., adding custom certs)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    # Use the official SonarScanner CLI image  
    FROM sonarsource/sonar-scanner-cli:latest  
      
    # The base image uses 'scanner-cli' user (UID 1000) [85]  
    # Switch to root to modify system certs if needed  
    USER root  
      
    # Example: Add custom CA certificate for SonarQube server  
    COPY --chown=root:root./my-custom-ca.crt /usr/local/share/ca-certificates/my-custom-ca.crt  
    RUN update-ca-certificates  
      
    # Switch back to the default scanner user  
    USER scanner-cli  
      
    # WORKDIR and ENTRYPOINT are inherited from base image  
    # Base WORKDIR is /usr/src [85]  
    # Base ENTRYPOINT likely points to a script that runs sonar-scanner
* **Configuration File Requirements**:
  + sonar-project.properties: The primary configuration file, typically placed at the root of the project being scanned.4 Defines project key, sources, exclusions, test paths, coverage report paths, language-specific properties, etc.
  + Alternatively, most properties can be passed as command-line arguments using -D<property>=<value> (e.g., -Dsonar.projectKey=my-project).4
  + Required properties:
    - sonar.projectKey: Unique key for the project in SonarQube.
    - sonar.sources: Path(s) to source code directories (relative to project root).
    - sonar.host.url: URL of the SonarQube server or SonarCloud (https://sonarcloud.io). Passed via -e SONAR\_HOST\_URL in Docker.84
    - sonar.token (or sonar.login): Authentication token for the SonarQube server. Passed via -e SONAR\_TOKEN or SONAR\_LOGIN/SONAR\_PASSWORD in Docker.4
* **Command-Line Usage Patterns**:
  + Basic scan (using sonar-project.properties in current dir):  
    Bash  
    docker run --rm \  
     -e SONAR\_HOST\_URL="https://sonarqube.mycompany.com" \  
     -e SONAR\_TOKEN="sqp\_abcdef..." \  
     -v "$(pwd):/usr/src" \  
     sonarsource/sonar-scanner-cli  
    84
  + Passing properties via -D:  
    Bash  
    docker run --rm \  
     -e SONAR\_HOST\_URL="https://sonarqube.mycompany.com" \  
     -e SONAR\_TOKEN="sqp\_abcdef..." \  
     -v "$(pwd):/usr/src" \  
     sonarsource/sonar-scanner-cli \  
     -Dsonar.projectKey=my-app \  
     -Dsonar.sources=. \  
     -Dsonar.projectName="My Application"
  + Enable debug logging: Add -X or -Dsonar.verbose=true to the command.4
* **Output Processing Approach**:
  + The scanner primarily sends its analysis report to the configured SonarQube server (sonar.host.url).
  + Console output shows the progress of the scan, configuration details, and a summary including a link to the analysis results on the SonarQube server.
  + Errors during the scan are reported to the console.
  + The main "output" to process is the analysis result viewed within the SonarQube UI.
* **Volume Mapping Strategy**:
  + /usr/src: Mount the host directory containing the project's source code and the sonar-project.properties file. This is the default WORKDIR in the official image.85
  + /opt/sonar-scanner/.sonar/cache (or ${SONAR\_USER\_HOME}/cache): Mount a persistent named volume here to cache downloaded analyzer plugins and sensor data between runs, significantly speeding up subsequent scans.4
  + Mount paths specified in sonar-project.properties for external reports (e.g., coverage reports via sonar.coverage.jacoco.xmlReportPaths, linting reports) must be accessible within the container. Example: If sonar.coverage.jacoco.xmlReportPaths=build/reports/jacoco/test/jacocoTestReport.xml, then the build/reports directory needs to be part of the /usr/src volume mount or mounted separately.
* **Resource Constraints**:
  + Java application, requires adequate JVM heap memory. Analysis can be memory-intensive, especially for large projects or complex languages.
  + Default JVM options might be set within the scanner's launch script. Can often be overridden via the SONAR\_SCANNER\_OPTS environment variable passed to the container (e.g., -e SONAR\_SCANNER\_OPTS="-Xmx2g").
  + CPU usage can be high during analysis phases.
  + Requires network access to the SonarQube server/SonarCloud and potentially artifact repositories (Maven Central, npm registry) to download analyzers.
  + Set container memory limits (e.g., 2GB - 8GB+) and CPU limits.
* **Common Containerization Issues and Solutions**:
  + *Issue*: Cannot connect to SonarQube server. *Solution*: Verify SONAR\_HOST\_URL is correct and reachable from the container. Check firewall rules. If using HTTPS with self-signed certificates, add the CA cert to the container's trust store (see Dockerfile example).4
  + *Issue*: Authentication failure. *Solution*: Ensure SONAR\_TOKEN (or SONAR\_LOGIN/PASSWORD) is correct and the token has sufficient permissions ('Execute Analysis' permission) on the SonarQube project.
  + *Issue*: Scanner runs out of memory. *Solution*: Increase container memory limit. Increase JVM heap size via SONAR\_SCANNER\_OPTS environment variable (e.g., -e SONAR\_SCANNER\_OPTS="-Xmx4g").
  + *Issue*: Scanner cannot find coverage/lint reports. *Solution*: Ensure the paths specified in sonar-project.properties (e.g., sonar.coverage.jacoco.xmlReportPaths) are correct *relative to the container's filesystem* (usually within the /usr/src mount). Ensure the reports are generated *before* the scanner runs and are accessible via volume mounts.
  + *Issue*: Permission denied accessing source code or cache. *Solution*: The official image runs as user scanner-cli (UID 1000).85 Ensure the mounted source code directory (/usr/src) is readable and the cache directory (/opt/sonar-scanner/.sonar/cache) is writable by this user ID. Use docker run --user $(id -u):$(id -g) if necessary (less secure) or adjust host directory permissions.
* **Health Check Implementation**:
  + The scanner is typically short-lived (runs analysis and exits). A traditional health check monitoring a long-running process isn't applicable.
  + For the SonarQube *server* container (not the scanner), health checks target the server's API endpoint (e.g., /api/system/health). This is outside the scope of containerizing the scanner CLI.
  + A basic check for the scanner image could verify Java or the scanner binary: CMD ["sonar-scanner", "-v"].
  + Dockerfile HEALTHCHECK (less critical for scanner): HEALTHCHECK --interval=5m --timeout=3s CMD sonar-scanner -v | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Adding custom CA)*: (See Dockerfile example under Installation section above)
  + *docker-compose.yaml (Running scan with persistent cache)*:  
    YAML  
    services:  
     sonarqube-scan:  
     image: sonarsource/sonar-scanner-cli:latest  
     # Run as host user to simplify volume permissions  
     # Note: The base image user is scanner-cli (1000). If host UID is different,  
     # permissions on the cache volume might need adjustment.  
     user: "${UID:-1000}:${GID:-1000}"  
     volumes:  
     # Mount project source code (read-only recommended)  
     -.:/usr/src:ro  
     # Mount persistent cache volume (must be writable)  
     - sonar\_cache:/opt/sonar-scanner/.sonar/cache  
     # Mount build reports if needed (example for JaCoCo)  
     -./target/site/jacoco:/usr/src/target/site/jacoco:ro  
     environment:  
     - SONAR\_HOST\_URL=${SONAR\_HOST\_URL} # Pass from host env  
     - SONAR\_TOKEN=${SONAR\_TOKEN} # Pass from host env  
     # Optional: Increase JVM heap size  
     - SONAR\_SCANNER\_OPTS=-Xmx2g  
     # Command implicitly uses sonar-project.properties from /usr/src  
     # Add -D flags here if not using properties file  
     # command: -Dsonar.projectKey=... -Dsonar.sources=...  
     deploy:  
     resources:  
     limits:  
     memory: 3G # Should be > SONAR\_SCANNER\_OPTS heap size  
     reservations:  
     memory: 1G  
      
    volumes:  
     sonar\_cache: # Define the named volume for persistence

The SonarScanner CLI container acts as a client to a SonarQube server or SonarCloud. Its primary role is to analyze the codebase (mounted via /usr/src) based on configuration (usually sonar-project.properties within the mounted code, or -D flags) and submit the results.4 Key containerization aspects include providing necessary credentials (SONAR\_TOKEN, SONAR\_HOST\_URL) via environment variables and managing the analysis cache (/opt/sonar-scanner/.sonar/cache) using a persistent volume for performance.4 Since the official image runs as a dedicated non-root user (scanner-cli, UID 1000) 85, careful attention must be paid to volume permissions, especially for the writable cache directory. The scanner often consumes external reports (like code coverage); ensuring these reports are generated beforehand and their paths correctly specified relative to the container's filesystem is crucial.

### **3.8 ScanCode Toolkit**

ScanCode Toolkit is a tool to scan code for licenses, copyrights, package manifests, dependencies, and other metadata, often used for license compliance and SBOM generation.86

* **Base Docker Image Recommendation**:
  + No single official image is promoted heavily, but the project provides a Dockerfile in its repository.88 Building this Dockerfile is a documented installation method.
  + The base image used in the provided Dockerfile should be inspected (not available in snippets). It likely requires Python (3.9-3.12 supported) and potentially C libraries for some dependencies.87 A standard python:3.x-slim or python:3.x-alpine image is a reasonable starting point if building manually.
  + Community images exist, like registry.gitlab.com/osp-silver/oss/scancode-toolkit-image 91, but using the project's own Dockerfile or building from a standard Python base ensures alignment with project dependencies.
* **Installation Dependencies and Commands**:
  + Python (3.9, 3.10, 3.11, or 3.12).87
  + System dependencies (vary by OS, needed for some Python packages): bzip2, xz-utils, zlib, libxml2-dev, libxslt1-dev are mentioned for Debian/Ubuntu.88 Install these using the base image's package manager (apk add, apt-get install).
  + Installation via pip: pip install scancode-toolkit[full].87 The [full] extra installs dependencies needed for most features.
  + Installation from source: Clone repo, run ./configure, then use the activated virtual environment.88 This is more complex within a Dockerfile but possible using multi-stage builds.
  + *Dockerfile (Building from Python base)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG PYTHON\_VERSION=3.11 # Use a supported version  
    FROM python:${PYTHON\_VERSION}-slim  
      
    # Install system dependencies (Debian/Ubuntu example)  
    RUN apt-get update && apt-get install -y --no-install-recommends \  
     bzip2 xz-utils zlib1g libxml2-dev libxslt1-dev \  
     && rm -rf /var/lib/apt/lists/\*  
      
    # Install ScanCode Toolkit with full extras  
    RUN pip install --no-cache-dir "scancode-toolkit[full]"  
      
    # Create a non-root user  
    RUN groupadd --gid 1001 scancode && \  
     useradd --uid 1001 --gid 1001 --create-home scancode  
      
    WORKDIR /scancode-work  
    USER scancode  
      
    ENTRYPOINT ["scancode"]  
    CMD ["--help"]
  + *Dockerfile (Building from source using project's Dockerfile)*: Check the official repository 92 for their Dockerfile and build it: docker build -t scancode-toolkit..88
* **Configuration File Requirements**:
  + ScanCode is primarily configured via command-line options.
  + No specific central configuration file format like .scancode.yaml is mentioned in the primary docs. Configuration relates to CLI flags passed during invocation.
* **Command-Line Usage Patterns**:
  + Basic scan: docker run --rm -v $(pwd):/project scancode-toolkit -clipeu --json-pp /project/scan-results.json /project.88 This scans the mounted /project directory for copyrights, licenses, packages, emails, and URLs, outputting pretty-printed JSON.
  + Specify output format: --json <file>, --json-pp <file>, --spdx-tv <file>, --spdx-rdf <file>, --cyclonedx <file>, --html <file>, etc..86 - outputs to stdout.
  + Select specific scans: -c (copyright), -l (license), -p (package), -e (email), -u (url), -i (info). Default is -clipeu.
  + Number of processes: -n <number> for parallel scanning.91
  + Ignore paths: --ignore "\*.o" --ignore "\*~".91
  + Timeout: --timeout <seconds>.
  + Use extractcode first for archives: The extractcode utility is bundled. Run it before scancode if archives need unpacking: docker run... scancode-toolkit extractcode /project/archive.zip then docker run... scancode-toolkit... /project/archive.zip-extract/.86 Some community images might integrate this.91
* **Output Processing Approach**:
  + Multiple output formats available: JSON (raw or pretty-printed), SPDX (Tag/Value or RDF), CycloneDX (XML or JSON), HTML, CSV, custom templates.86
  + JSON is the most comprehensive format for machine processing.
  + SPDX and CycloneDX formats are standard SBOM formats suitable for interoperability.
  + HTML provides a human-readable report.
  + ScanCode Workbench can be used to visualize JSON results.93
* **Volume Mapping Strategy**:
  + /project (or /scan, /app): Mount the host directory containing the codebase to be scanned.88 Mount read-only if possible.
  + /output: Mount a host directory if writing report files directly to the host filesystem. Alternatively, write reports within the /project mount.
* **Resource Constraints**:
  + License/copyright scanning involves file I/O and pattern matching. Can be CPU-intensive, especially with parallel processing (-n).
  + Package manifest parsing is generally less intensive.
  + Memory usage depends on the number and size of files being scanned and the number of parallel processes. Large codebases require more memory.
  + Recommend setting limits (e.g., 2GB-8GB+ memory, 2+ CPUs depending on -n value) and monitoring.
* **Common Containerization Issues and Solutions**:
  + *Issue*: Missing Python or system dependencies. *Solution*: Ensure the base image has the correct Python version (3.9+) and that necessary system libraries (like libxml2) are installed using the image's package manager.88 Use the scancode-toolkit[full] pip install extra.87
  + *Issue*: Slow scans on large codebases. *Solution*: Increase CPU/memory resources. Adjust the number of parallel processes (-n) based on available cores and memory. Use --timeout if specific files hang. Exclude irrelevant directories (--ignore). Pre-extract archives using extractcode.
  + *Issue*: Permission errors accessing mounted code or writing reports. *Solution*: Run the container with the host user's UID/GID (--user $(id -u):$(id -g)). Ensure the non-root user inside the container (if used) has read access to the code mount and write access to the output mount.
  + *Issue*: Inaccurate results for archives. *Solution*: Ensure archives are extracted *before* scanning using extractcode.86 ScanCode doesn't automatically scan inside archives by default.
* **Health Check Implementation**:
  + Simple check: CMD ["scancode", "--version"].
  + Basic functionality check: CMD ["scancode", "--help"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=10s CMD scancode --version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Building from Python base)*: (See Dockerfile example under Installation section above)
  + *docker-compose.yaml (Scanning local directory)*:  
    YAML  
    services:  
     scancode-scan:  
     # Assumes an image named 'scancode-toolkit' built from the Dockerfile above  
     # Or use a pre-built image if available and trusted  
     image: scancode-toolkit:latest  
     user: "${UID:-1000}:${GID:-1000}" # Run as host user  
     volumes:  
     # Mount project code read-only  
     -.:/scancode-work:ro  
     # Mount output directory  
     -./scancode-reports:/scancode-work/output  
     # Scan mounted code, use 4 processes, output SPDX TV and JSON  
     command: >  
     -clp  
     --spdx-tv /scancode-work/output/sbom.spdx  
     --json-pp /scancode-work/output/scancode.json  
     --processes 4  
     --strip-root  
     /scancode-work  
     deploy:  
     resources:  
     limits:  
     # Adjust based on codebase size and number of processes  
     cpus: '4.0'  
     memory: 8G  
     reservations:  
     cpus: '1.0'  
     memory: 1G

ScanCode Toolkit requires a Python environment and potentially some underlying C libraries, making base image selection important.88 While community images exist 91, building from the project's own Dockerfile 92 or a standard Python image provides more control and ensures compatibility.89 The tool is configured primarily via extensive command-line options controlling which scanners run (-c, -l, -p, etc.) and the output format (--json, --spdx-tv, etc.).86 A key operational aspect is handling archives; ScanCode typically requires archives to be extracted beforehand using the bundled extractcode utility for accurate analysis.86 Volume mapping is needed to provide access to the target codebase and retrieve the generated reports. Performance can be tuned using parallel processing (-n), but this increases resource requirements.91

### **3.9 LicenseFinder**

LicenseFinder works with package managers to find project dependencies, detect their licenses, and compare them against a permitted list.94

* **Base Docker Image Recommendation**:
  + licensefinder/license\_finder:latest: This appears to be the official Docker image listed on Docker Hub.95 It aims to bundle support for multiple package managers (Ruby, Python, Node, Java, Go, etc.).94 Inspecting its layers or Dockerfile (if available) is recommended to understand its base and included dependencies. Older community images (tabll/license-finder) exist but may be outdated.96
  + If building manually, the base image choice depends heavily on the *specific* package managers needed for the projects being scanned. A multi-stage build combining bases (e.g., Ruby for the tool itself, plus Node, Python, JDK as needed) might be necessary but complex. Using the official image is generally much simpler.
* **Installation Dependencies and Commands**:
  + Ruby (>= 2.6.0) is required to run LicenseFinder itself.94
  + Requires the specific package managers for the projects being scanned (e.g., bundler, pip, npm, maven, gradle, go, yarn, cargo, etc.).94
  + The official Docker image licensefinder/license\_finder intends to pre-install many of these.94
  + Installation of LicenseFinder tool: gem install license\_finder.94
  + *Dockerfile (Illustrative - Complex due to multi-language support;* ***using official image is highly recommended****)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    # Example: Starting with Ruby base, adding Node and Python  
    FROM ruby:3.1-slim as base  
      
    # Install Node.js (example using nodesource)  
    RUN apt-get update && apt-get install -y --no-install-recommends curl gnupg && \  
     curl -fsSL https://deb.nodesource.com/setup\_20.x | bash - && \  
     apt-get install -y nodejs  
      
    # Install Python & pip  
    RUN apt-get install -y python3 python3-pip  
      
    # Install LicenseFinder gem  
    RUN gem install license\_finder  
      
    #... Add other package managers (Java, Go, Rust etc.)...  
    # This becomes very complex quickly.  
      
    WORKDIR /scan-target  
    ENTRYPOINT ["license\_finder"]  
    CMD ["--help"]  
      
    # Consider non-root user setup
* **Configuration File Requirements**:
  + doc/dependency\_decisions.yml (or configured path): This file is used to record decisions about licenses (permitting, restricting) and specific dependencies (approving, marking as transitive). LicenseFinder interacts with this file (license\_finder approvals add...).
  + Configuration options can influence which package managers are active or how reports are generated, potentially via environment variables or future config files (documentation primarily focuses on CLI interaction and the decisions file).
* **Command-Line Usage Patterns**:
  + Run scan (reports unapproved dependencies): docker run --rm -v $(pwd):/scan-target licensefinder/license\_finder.94
  + Run via provided dlf script (wraps docker run): dlf (equivalent to license\_finder) or dlf report --format=html > report.html.94
  + Show help: docker run --rm licensefinder/license\_finder --help.
  + Generate report: docker run --rm -v $(pwd):/scan-target licensefinder/license\_finder report --format=csv --save=report.csv. Formats include text, html, csv, markdown, json.
  + Manage approvals:
    - Permit a license: ... license\_finder licenses permit MIT
    - Approve a dependency: ... license\_finder approvals add MyDependency --who="Dev Team" --why="Approved for use"
  + Use --quiet to suppress progress dots, --debug for verbose output.94
  + Ensure project dependencies are installed (bundle install, npm install, etc.) *before* running license\_finder, usually outside the scanner run unless dependencies are vendored.
* **Output Processing Approach**:
  + Default output lists unapproved dependencies or confirms all are approved.
  + license\_finder report generates a detailed report in various formats (HTML, CSV, JSON, Markdown, text).94
  + HTML report provides a browseable overview.
  + CSV/JSON formats are suitable for machine processing or integration with other compliance tools.
  + The primary "actionable" output is the list of dependencies requiring approval decisions.
* **Volume Mapping Strategy**:
  + /scan-target (or /code, /src): Mount the host directory containing the project source code, including package manager files (Gemfile, package.json, pom.xml, etc.) and the doc/dependency\_decisions.yml file.94 This volume needs to be writable if using commands that modify dependency\_decisions.yml.
  + Mount cache directories for specific package managers if needed (e.g., /root/.m2, /root/.gradle, /root/.npm) to potentially speed up dependency analysis if run within the container, though typically dependencies are installed beforehand on the host or in a CI build step.
* **Resource Constraints**:
  + Depends heavily on the package managers being invoked. Java-based managers (Maven, Gradle) can be memory-intensive.
  + Requires network access if package managers need to fetch metadata.
  + Ruby runtime itself is relatively lightweight.
  + Set resource limits based on the most demanding package manager expected (e.g., 2GB+ memory for Java projects).
* **Common Containerization Issues and Solutions**:
  + *Issue*: Required package manager (e.g., Gradle, Maven, Go) not found or wrong version. *Solution*: Use the official licensefinder/license\_finder image which aims to include necessary tools.94 If a specific version is needed that's not in the image, building a custom image is required, which is complex. Ensure the project uses versions compatible with the tools in the container.
  + *Issue*: LicenseFinder fails because project dependencies are not installed. *Solution*: Run the appropriate package manager install command (bundle install, npm install, mvn install -DskipTests, etc.) on the host or in a preceding CI stage *before* running the LicenseFinder container. Dependencies generally need to be present for analysis.
  + *Issue*: Cannot write to dependency\_decisions.yml. *Solution*: Ensure the volume mount for the project directory is writable, or run approval commands on the host rather than inside the container.
  + *Issue*: Network issues fetching package metadata. *Solution*: Ensure the container has network access. Configure proxy settings if necessary (e.g., via environment variables like HTTP\_PROXY, HTTPS\_PROXY).
* **Health Check Implementation**:
  + Simple check: CMD ["license\_finder", "--version"].
  + Check Ruby environment: CMD ["ruby", "-v"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=3s CMD license\_finder --version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Illustrative - Using official image as base is preferred)*: (See complex example in Installation section; prefer official image)
  + *docker-compose.yaml (Using official image to scan local project)*:  
    YAML  
    services:  
     license-scan:  
     image: licensefinder/license\_finder:latest  
     # Run as host user to handle permissions on dependency\_decisions.yml  
     user: "${UID:-1000}:${GID:-1000}"  
     volumes:  
     # Mount project directory (needs to be writable for approvals)  
     -.:/scan-target  
     # Optional: Mount package manager caches if analysis needs them  
     # - ~/.m2:/root/.m2  
     # - ~/.gradle:/root/.gradle  
     # Default command runs license\_finder check  
     # Or specify 'report' command:  
     # command: report --format html --save /scan-target/license-report.html  
     working\_dir: /scan-target # Ensure commands run in project context  
     deploy:  
     resources:  
     limits:  
     # Adjust based on expected package managers (Java needs more)  
     memory: 2G  
     reservations:  
     memory: 512M

LicenseFinder's multi-language support presents its main containerization challenge: the container needs not only Ruby (for LicenseFinder itself) but also the various package managers (Bundler, npm, pip, Maven, Gradle, Go tools, Cargo, etc.) relevant to the projects being scanned.94 The official licensefinder/license\_finder image attempts to solve this by bundling many common package managers 97, making it the most practical choice over building a complex multi-language image manually. Users typically mount their project directory into the container and run license\_finder, which then invokes the necessary package managers to analyze dependencies.94 Managing the doc/dependency\_decisions.yml file often requires a writable volume mount or running approval commands outside the container.

### **3.10 Sigstore (Cosign)**

Cosign is the primary command-line utility for signing and verifying software artifacts (like container images, blobs, WebAssembly modules) using the Sigstore infrastructure (Fulcio, Rekor, OIDC).99

* **Base Docker Image Recommendation**:
  + gcr.io/projectsigstore/cosign:latest or specific version tag (e.g., gcr.io/projectsigstore/cosign:v2.2.4): This is the official image provided by the Sigstore project.
  + As Cosign is a Go application, it's a static binary. The official image base is likely minimal (e.g., distroless or scratch + ca-certs). Inspecting layers is recommended.
  + Chainguard offers cgr.dev/chainguard/cosign, built on Wolfi with Sigstore signatures, providing a verifiable, minimal alternative.
* **Installation Dependencies and Commands**:
  + Cosign binary.
  + ca-certificates: For interacting with Sigstore services (Fulcio CA, Rekor transparency log) and OCI registries over HTTPS.
  + Git (optional, needed for some keyless verification scenarios involving GitHub Actions workflow identity).
  + *Dockerfile (using pre-built binary)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG COSIGN\_VERSION=v2.2.4 # Use latest stable release  
    FROM alpine:3.19 as certs  
    # Base image with only ca-certificates  
    FROM scratch  
    COPY --from=certs /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-certificates.crt  
      
    ARG COSIGN\_VERSION  
    # Download cosign binary for linux amd64  
    ADD https://github.com/sigstore/cosign/releases/download/${COSIGN\_VERSION}/cosign-linux-amd64 /usr/local/bin/cosign  
    RUN chmod +x /usr/local/bin/cosign  
      
    # Optional: Install git if needed for specific verification types  
    # FROM alpine:3.19 as git\_installer  
    # RUN apk add --no-cache git  
    # COPY --from=git\_installer /usr/bin/git /usr/bin/git  
    # COPY --from=git\_installer /usr/lib/lib\* /usr/lib/  
    # COPY --from=git\_installer /lib/ld-musl\* /lib/  
      
    WORKDIR /work  
    ENTRYPOINT ["/usr/local/bin/cosign"]  
    CMD ["--help"]
* **Configuration File Requirements**:
  + Cosign primarily uses command-line flags and environment variables.
  + Key management: Keys can be provided via files (--key <file>), environment variables (COSIGN\_PASSWORD), cloud KMS (e.g., GCP, AWS, Azure, HashiCorp Vault), or hardware tokens (PKCS#11).
  + Environment Variables:
    - COSIGN\_PASSWORD: Password for private keys.
    - COSIGN\_EXPERIMENTAL=1: Enables experimental features (like keyless signing in earlier versions, now default).
    - TUF\_ROOT: Path to TUF root metadata (for Sigstore client trust).
    - SIGSTORE\_\*: Variables for configuring endpoints (e.g., SIGSTORE\_FULCIO\_URL, SIGSTORE\_REKOR\_URL).
* **Command-Line Usage Patterns**:
  + Keyless signing (OIDC): docker run --rm -it -v $(pwd):/work gcr.io/projectsigstore/cosign sign <image\_uri> (Requires interactive login flow via browser or device flow in CI).
  + Signing with key file: docker run --rm -v $(pwd):/work -e COSIGN\_PASSWORD=$KEY\_PASS gcr.io/projectsigstore/cosign sign --key /work/cosign.key <image\_uri>
  + Verifying image: docker run --rm gcr.io/projectsigstore/cosign verify --certificate-identity user@example.com --certificate-oidc-issuer https://accounts.example.com <image\_uri>.99
  + Verifying with key: docker run --rm -v $(pwd):/work gcr.io/projectsigstore/cosign verify --key /work/cosign.pub <image\_uri>
  + Signing blobs: docker run --rm -v $(pwd):/work gcr.io/projectsigstore/cosign sign-blob --key /work/cosign.key --bundle /work/blob.bundle /work/myblob.txt.99
  + Verifying blobs: docker run --rm -v $(pwd):/work gcr.io/projectsigstore/cosign verify-blob --key /work/cosign.pub --bundle /work/blob.bundle /work/myblob.txt.99
  + Attaching attestations (e.g., SPDX SBOM): docker run --rm -v $(pwd):/work gcr.io/projectsigstore/cosign attest --key /work/cosign.key --type spdxjson --predicate /work/sbom.spdx.json <image\_uri>
  + Verifying attestations: docker run --rm -v $(pwd):/work gcr.io/projectsigstore/cosign verify-attestation --key /work/cosign.pub --type spdxjson <image\_uri>
* **Output Processing Approach**:
  + Commands typically output status messages (e.g., "Pushing signature...", "Verification for... succeeded") to stderr/stdout.
  + verify commands exit 0 on success, non-zero on failure. Suitable for scripting/CI checks.
  + Signatures and attestations are pushed to the OCI registry alongside the image, not usually output as files unless signing blobs (--bundle).
  + cosign triangulate <image\_uri> can be used to find associated signatures/attestations in the registry.
* **Volume Mapping Strategy**:
  + /work (or /config): Mount host directory containing key files (cosign.key, cosign.pub), blobs to sign/verify, predicate files (SBOMs), or bundles.
  + /root/.docker/config.json (or path specified by DOCKER\_CONFIG): Mount Docker config file if needed for authenticating to the OCI registry where images/signatures are stored. Cosign uses go-containerregistry which respects standard Docker credential helpers.
* **Resource Constraints**:
  + Cosign itself is a lightweight Go application.
  + Primary resource usage is network I/O for communicating with Sigstore services (Fulcio, Rekor) and the OCI registry.
  + CPU/Memory usage is generally low. Set minimal resource limits (e.g., 128-512MB memory, <1 CPU).
* **Common Containerization Issues and Solutions**:
  + *Issue*: Keyless signing fails in non-interactive environment (CI). *Solution*: Use OIDC identity tokens specific to the CI provider (e.g., GitHub Actions OIDC tokens, gh CLI identity). Configure Cosign to use these tokens. Alternatively, use key-based signing with the key password passed via COSIGN\_PASSWORD environment variable, sourcing the password securely (e.g., from CI secrets).
  + *Issue*: Cannot connect to Sigstore services or OCI registry. *Solution*: Ensure container has outbound network access. Check firewall rules. Ensure ca-certificates are present and up-to-date in the image. Configure proxies if needed.
  + *Issue*: Authentication errors with OCI registry. *Solution*: Mount config.json or ensure Docker credential helpers are configured and accessible within the container environment. For specific cloud registries (ECR, GCR, ACR), ensure appropriate authentication mechanisms are set up (e.g., IAM roles, service accounts).
  + *Issue*: Permission denied accessing key files or writing bundles. *Solution*: Ensure volume mounts are correct and the container user has read access to key files and write access to output directories. Run as host user (--user $(id -u):$(id -g)) if necessary, or adjust host file permissions.
* **Health Check Implementation**:
  + Simple check: CMD ["cosign", "version"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=3s CMD cosign version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Minimal, using official image)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    # Use official cosign image  
    FROM gcr.io/projectsigstore/cosign:latest  
      
    # Image likely runs as non-root or root; check documentation/layers.  
    # Assume WORKDIR / work is set or default is acceptable.  
    # ENTRYPOINT ["cosign"]  
    # CMD ["--help"]
  + *docker-compose.yaml (Signing an image using a key)*:  
    YAML  
    services:  
     cosign-sign:  
     image: gcr.io/projectsigstore/cosign:latest  
     user: "${UID:-1000}:${GID:-1000}" # Run as host user for key access  
     volumes:  
     # Mount directory containing keys  
     -./sigstore-keys:/work/keys:ro  
     # Mount Docker config for registry authentication  
     - ~/.docker/config.json:/root/.docker/config.json:ro  
     environment:  
     # Pass key password securely from host environment  
     - COSIGN\_PASSWORD=${COSIGN\_PASSWORD}  
     # Enable experimental features if needed (usually not required now)  
     # - COSIGN\_EXPERIMENTAL=1  
     # Command to sign the image  
     command: sign --key /work/keys/cosign.key my-registry/my-image:latest  
     deploy:  
     resources:  
     limits:  
     memory: 512M  
     reservations:  
     memory: 128M

Cosign, as a static Go binary, is easily containerized using minimal base images like the official gcr.io/projectsigstore/cosign or Chainguard's offering. The main complexities arise from managing authentication and keys. Keyless signing, while convenient interactively, requires specific OIDC token handling in automated CI environments.99 Key-based signing requires securely managing the key file (via volumes) and its password (via COSIGN\_PASSWORD environment variable, sourced from secrets). Interaction with OCI registries for pushing/pulling signatures also necessitates proper registry authentication, often handled by mounting the Docker config.json. Network access to Sigstore services (Fulcio, Rekor) is fundamental for most operations.99

### **3.11 in-toto**

in-toto is a framework to secure the integrity of software supply chains by verifying that software is built according to predefined steps and policies. It involves creating metadata (links) about software build steps and verifying them against a layout.100 Containerization typically focuses on the *verification* aspect or running specific *functionary* steps.

* **Base Docker Image Recommendation**:
  + The official Python reference implementation (in-toto) is the most common.100
  + Requires Python. Use an official Python base image: python:3.x-slim or python:3.x-alpine.
  + The reference implementation itself has dependencies like securesystemslib, attrs, potentially cryptography (which might need C build tools like gcc, musl-dev on Alpine, or build-essential, libssl-dev, python3-dev on Debian).
  + A community example shows building a verification image based on python:latest.101
* **Installation Dependencies and Commands**:
  + Python (check in-toto package for supported versions).
  + pip.
  + Potentially build dependencies for cryptography (see above).
  + git (often needed to fetch layouts or materials).
  + Installation: pip install in-toto.101
  + *Dockerfile (for verification)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG PYTHON\_VERSION=3.11  
    FROM python:${PYTHON\_VERSION}-slim  
      
    # Install build dependencies for cryptography, and git  
    RUN apt-get update && apt-get install -y --no-install-recommends \  
     build-essential libssl-dev python3-dev git \  
     && rm -rf /var/lib/apt/lists/\*  
      
    # Install in-toto library  
    RUN pip install --no-cache-dir in-toto  
      
    # Create a directory for layouts, keys, artifacts as per convention [101]  
    RUN mkdir /in-toto  
    WORKDIR /in-toto  
      
    # Create a non-root user  
    RUN groupadd --gid 1001 intoto && \  
     useradd --uid 1001 --gid 1001 --create-home intoto  
    USER intoto  
      
    # Entrypoint for verification command  
    ENTRYPOINT ["in-toto-verify"]  
    CMD ["--help"]
* **Configuration File Requirements**:
  + Layout file (\*.layout): Defines the expected supply chain steps, functionaries, materials, and products. This is the primary configuration for verification.
  + Public key files (\*.pub): Keys corresponding to functionaries who signed link metadata.
  + Link metadata files (\*.link): Attestations about executed steps.
  + These files are typically mounted into the container.
* **Command-Line Usage Patterns**:
  + Verify supply chain: docker run --rm -v $(pwd)/layout:/in-toto/layout -v $(pwd)/keys:/in-toto/keys -v $(pwd)/links:/in-toto/links -v $(pwd)/product:/in-toto/product my-intoto-verifier --layout /in-toto/layout/root.layout --layout-keys /in-toto/keys/root\_key.pub --link-dir /in-toto/links --product /in-toto/product/final\_artifact
  + Run a step and generate link metadata: docker run --rm -v $(pwd):/in-toto my-intoto-runner in-toto-run --step-name <step> --materials <in\_files> --products <out\_files> --key /in-toto/keys/step\_key -- <command\_to\_run>
  + Generate layout template: in-toto-layout-gen...
* **Output Processing Approach**:
  + in-toto-verify: Exits 0 if verification passes, non-zero otherwise. Outputs status messages to stdout/stderr indicating success or specific verification failures.
  + in-toto-run: Generates a .link metadata file containing attestations about the executed command, signed with the provided key.
* **Volume Mapping Strategy**:
  + /in-toto (or subdirectories like /in-toto/layout, /in-toto/keys, /in-toto/links, /in-toto/product): Mount host directories containing the layout file(s), public keys, link metadata files, and the final product(s) to be verified.101
  + If using in-toto-run, mount the directory containing materials, product destination, and the private key.
* **Resource Constraints**:
  + Verification is typically lightweight, involving metadata parsing and cryptographic signature checks.
  + Running steps (in-toto-run) depends entirely on the resource needs of the <command\_to\_run>.
  + Set minimal limits for verification containers (e.g., 256-512MB memory, <1 CPU). Adjust significantly if running build steps via in-toto-run.
* **Common Containerization Issues and Solutions**:
  + *Issue*: Missing layout, keys, links, or product files. *Solution*: Ensure all necessary files are correctly mounted via volumes to the expected paths inside the container (e.g., /in-toto/...). Check file permissions.
  + *Issue*: Python dependency conflicts or build errors (e.g., for cryptography). *Solution*: Ensure necessary build tools (gcc, -dev packages) are installed in the Dockerfile *before* pip install in-toto. Use a compatible Python version.
  + *Issue*: Verification fails due to command execution errors (if layout specifies inspections). *Solution*: Ensure any tools required by inspection commands defined in the layout are installed in the verification container image.101
  + *Issue*: Key errors (wrong key, password needed). *Solution*: Ensure correct public keys are mounted for verification. If running steps, ensure the correct private key is mounted and handle password prompts if necessary (e.g., via environment variables or expect scripts, though less secure).
* **Health Check Implementation**:
  + Simple check: CMD ["in-toto-verify", "--version"].
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=3s CMD in-toto-verify --version | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (for verification)*: (See Dockerfile example under Installation section above)
  + *docker-compose.yaml (Running verification)*:  
    YAML  
    services:  
     intoto-verify:  
     build:. # Assumes Dockerfile above is in current dir  
     user: "${UID:-1000}:${GID:-1000}" # Run as host user  
     volumes:  
     # Mount necessary in-toto metadata and product files  
     -./demo-project/root.layout:/in-toto/root.layout:ro  
     -./demo-project/keys:/in-toto/keys:ro  
     -./demo-project/metadata:/in-toto/links:ro  
     -./demo-project/final\_product:/in-toto/product:ro  
     # Command to perform verification  
     command: >  
     --layout /in-toto/root.layout  
     --layout-keys /in-toto/keys/root\_layout\_key.pub  
     --link-dir /in-toto/links  
     --product /in-toto/product/package.tar.gz  
     deploy:  
     resources:  
     limits:  
     memory: 512M  
     reservations:  
     memory: 128M

Containerizing in-toto primarily focuses on creating a consistent environment for the in-toto-verify command, which checks supply chain integrity based on provided metadata.101 This requires a Python environment and the in-toto library, potentially including build tools for cryptographic dependencies. The main interaction with the host system is through volume mounts to bring in the layout, keys, link metadata, and final product artifacts for verification.101 Running actual supply chain steps via in-toto-run within a container is also possible but requires a more complex image containing all tools needed for that specific step, in addition to the in-toto tooling.

### **3.12 YARA**

YARA is a tool aimed at helping malware researchers identify and classify malware samples by creating rules (based on textual or binary patterns) to describe families of malware.102

* **Base Docker Image Recommendation**:
  + No single official YARA image is maintained by the core YARA project (VirusTotal/Google).
  + Community/Vendor images exist, e.g., cincan/yara 103, which bundles YARA and common rule sets. Evaluate their maintenance and base image.
  + Building from source: YARA is written in C. Requires build tools (gcc, make, automake, libtool, pkg-config) and dependencies (openssl-dev, jansson-dev, magic-dev, protobuf-c-compiler, protobuf-dev). A minimal base like debian:slim or alpine with build tools installed in a builder stage is suitable.
  + Recommended: Build a custom image using a minimal base (alpine or debian-slim) and compile YARA from source for control over version and dependencies.
* **Installation Dependencies and Commands**:
  + Build tools (if compiling from source): build-essential, automake, libtool, pkg-config (Debian); build-base, automake, libtool, pkgconf (Alpine).
  + Runtime dependencies: libssl, libjansson, libmagic, libprotobuf-c. Install dev packages first if building (openssl-dev, jansson-dev, file-dev or magic-dev, protobuf-c-dev).
  + YARA source code (download release tarball or clone git repo).
  + Compilation steps: ./bootstrap.sh, ./configure, make, make install (typical autotools flow).
  + *Dockerfile (Compiling YARA from source)*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG YARA\_VERSION=v4.5.0 # Check latest stable release  
    FROM alpine:3.19 as builder  
      
    ARG YARA\_VERSION  
    # Install build tools and YARA build dependencies  
    RUN apk add --no-cache \  
     build-base automake autoconf libtool pkgconf \  
     openssl-dev jansson-dev file-dev protobuf-c-dev  
      
    # Download and extract YARA source  
    RUN wget https://github.com/VirusTotal/yara/archive/refs/tags/${YARA\_VERSION}.tar.gz -O yara.tar.gz && \  
     tar -xzf yara.tar.gz && \  
     rm yara.tar.gz  
      
    # Build YARA  
    WORKDIR /yara-${YARA\_VERSION#v}  
    RUN./bootstrap.sh && \  
     ./configure --enable-magic --enable-dotnet --enable-macho --enable-dex && \  
     make && \  
     make install DESTDIR=/yara-install  
      
    # Final minimal image  
    FROM alpine:3.19  
    # Install runtime dependencies  
    RUN apk add --no-cache libssl3 libjansson libmagic libprotobuf-c  
      
    # Copy compiled YARA from builder stage  
    COPY --from=builder /yara-install/usr/local/ /usr/local/  
      
    # Create non-root user  
    RUN addgroup -S yara && adduser -S -G yara yara  
    USER yara  
      
    WORKDIR /scan-target  
    ENTRYPOINT ["yara"]  
    CMD ["--help"]
* **Configuration File Requirements**:
  + YARA itself doesn't typically use a central configuration file. Configuration is done via CLI options.
  + Rule files (.yar or .yara): These contain the patterns YARA searches for. They are the primary "configuration".
  + Compiled rules files: Rules can be pre-compiled using yarac for faster loading.
* **Command-Line Usage Patterns**:
  + Scan a file/directory with rules: docker run --rm -v $(pwd)/rules:/rules -v $(pwd)/target:/scan-target my-yara-image /rules/my\_rules.yar /scan-target/suspicious\_file.103
  + Scan with compiled rules: ... my\_compiled\_rules.yc /scan-target/directory.
  + Specify external variables: -d myvar=value.
  + Set stack size: -k <slots>.
  + Set timeout: -a <seconds>.
  + Compile rules: docker run --rm -v $(pwd)/rules:/rules my-yara-image yarac -o /rules/compiled.yc /rules/index.yar
* **Output Processing Approach**:
  + Default output lists matching rules and the files they matched in.
  + -s: Show matching strings.
  + -m: Show metadata associated with matching rules.
  + -n: Show non-matching files (negated results).
  + Output is primarily text-based to stdout. Parsing requires custom scripting or integration with tools that understand YARA output.
* **Volume Mapping Strategy**:
  + /rules: Mount host directory containing YARA rule files (.yar) or compiled rules (.yc).
  + /scan-target: Mount host directory containing the files or directories to be scanned.
  + /output: Mount host directory if YARA is configured (e.g., via scripting wrappers) to write output files.
* **Resource Constraints**:
  + Memory usage depends heavily on the complexity and number of rules loaded, and the size of the files being scanned. Compiling large rule sets can require significant memory.
  + Scanning large files can also consume substantial memory.
  + CPU usage depends on rule complexity and file size/count.
  + Set appropriate memory limits (e.g., 1GB+, monitor closely based on rule sets and targets) and CPU limits. YARA's -k (stack size) option might need tuning for complex rules.
* **Common Containerization Issues and Solutions**:
  + *Issue*: Compilation errors when building YARA from source. *Solution*: Ensure all build dependencies (automake, -dev libraries like openssl-dev, jansson-dev, file-dev, protobuf-c-dev) are installed correctly in the builder stage. Check compatibility between YARA version and dependency versions.
  + *Issue*: Runtime errors due to missing shared libraries (.so files). *Solution*: Ensure all runtime dependencies (libssl, libjansson, libmagic, libprotobuf-c) are installed in the final container image stage.
  + *Issue*: High memory usage or crashes during scans. *Solution*: Increase container memory limit. Optimize YARA rules. Scan smaller sets of files at a time. Use compiled rules (.yc) for faster loading. Adjust stack size (-k) if needed.
  + *Issue*: Cannot access rules or target files. *Solution*: Verify volume mounts are correct and paths used in the command match the container's filesystem. Check file permissions, especially if running as non-root.
* **Health Check Implementation**:
  + Simple check: CMD ["yara", "-v"] (version).
  + Dockerfile HEALTHCHECK: HEALTHCHECK --interval=5m --timeout=3s CMD yara -v | | exit 1
* **Example Dockerfile and docker-compose Definition**:
  + *Dockerfile (Compiling YARA)*: (See Dockerfile example under Installation section above)
  + *docker-compose.yaml (Scanning local directory with local rules)*:  
    YAML  
    services:  
     yara-scan:  
     build:. # Assumes Dockerfile above is in current dir  
     user: "${UID:-1000}:${GID:-1000}" # Run as host user  
     volumes:  
     # Mount directory with YARA rules read-only  
     -./yara-rules:/rules:ro  
     # Mount directory to be scanned read-only  
     -./scan-data:/scan-target:ro  
     # Scan target directory with all rules in /rules/index.yar  
     # Adjust rule path as needed  
     command: /rules/index.yar /scan-target  
     deploy:  
     resources:  
     limits:  
     memory: 2G # Adjust based on ruleset/target size  
     reservations:  
     memory: 512M

Containerizing YARA involves either using a pre-built community image 103 or compiling it from source within a Dockerfile, which requires managing C build tools and dependencies. Since YARA is configured via CLI and rule files, volume mounts are essential for providing access to both the .yar/.yc rule files and the target files/directories to be scanned. Performance, particularly memory usage, can be a concern with large or complex rule sets and when scanning large files, necessitating careful resource allocation and potentially rule optimization or the use of pre-compiled rules.

### **3.13 Capa**

Capa identifies capabilities in executable files (PE, ELF, shellcode) using static and dynamic analysis techniques, leveraging a library of rules.104

* **Base Docker Image Recommendation**:
  + No official Docker image is provided by Mandiant/FLARE team.104
  + Capa is a Python application. Requires Python >= 3.10.104
  + Recommended base: python:3.10-slim or python:3.11-slim (or newer supported versions). Alpine might work but could have compatibility issues with some dependencies.
  + FLARE-VM includes Capa, but using the full VM image as a base is excessive for just running Capa.105
* **Installation Dependencies and Commands**:
  + Python (>= 3.10).104
  + pip.
  + Capa package: pip install capa-rules. The capa-rules package includes the core tool and the default rule set.
  + Optional: If analyzing PE files without relying on bundled smda, install pefile. If analyzing ELF files, install pyelftools. These are usually installed as dependencies of capa-rules.
  + *Dockerfile*:  
    Dockerfile  
    # syntax=docker/dockerfile:1  
    ARG PYTHON\_VERSION=3.11 # Use supported version >= 3.10  
    FROM python:${PYTHON\_VERSION}-slim  
      
    # Install capa and its rules  
    # capa-rules includes capa core  
    RUN pip install --no-cache-dir capa-rules  
      
    # Create a non-root user  
    RUN groupadd --gid 1001 capa && \  
     useradd --uid 1001 --gid 1001 --create-home capa  
      
    WORKDIR /scan-target  
    USER capa  
      
    ENTRYPOINT ["capa"]  
    CMD ["--help"]
* \*\*Configuration File Requirements

#### Works cited

1. syft/Dockerfile at main · anchore/syft - GitHub, accessed May 8, 2025, <https://github.com/anchore/syft/blob/main/Dockerfile>
2. syft - overview - Chainguard Containers, accessed May 8, 2025, <https://images.chainguard.dev/directory/image/syft/overview>
3. anchore/grype: A vulnerability scanner for container images ... - GitHub, accessed May 8, 2025, <https://github.com/anchore/grype>
4. SonarScanner CLI | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-server/10.8/analyzing-source-code/scanners/sonarscanner/>
5. A Semgrep scan is having a problem - what next?, accessed May 8, 2025, <https://semgrep.dev/docs/kb/semgrep-code/semgrep-scan-troubleshooting>
6. Semgrep Network Broker, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-ci/network-broker>
7. grype - overview - Chainguard Containers, accessed May 8, 2025, <https://images.chainguard.dev/directory/image/grype/overview>
8. BenjiTrapp/syft-fork: CLI tool and library for generating a Software Bill of Materials from container images and filesystems - GitHub, accessed May 8, 2025, <https://github.com/BenjiTrapp/syft-fork>
9. How Syft Scans Software to Generate SBOMs - Anchore, accessed May 8, 2025, <https://anchore.com/blog/how-syft-scans-software-to-generate-sboms/>
10. Image Layer Details - anchore/syft:v0.91.0 | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/anchore/syft/v0.91.0/images/sha256-5bf41ee55321901e89dc2314328130046cb09ef937bcdb6fee810b0f5dc7cbe8>
11. accessed December 31, 1969, <https://raw.githubusercontent.com/anchore/syft/main/Dockerfile>
12. Configuration · anchore/syft Wiki - GitHub, accessed May 8, 2025, <https://github.com/anchore/syft/wiki/configuration>
13. Creating SBOMs From Images Using Syft - GitHub, accessed May 8, 2025, <https://github.com/cybeats/sbomgen/blob/main/tutorials/containers/creating-syft-sbom/creating-syft-sbom.md>
14. Open Source Vulnerability Scanning: Tools & How-To Guide - Anchore, accessed May 8, 2025, <https://anchore.com/software-supply-chain-security/open-source-container-vulnerability-scanning-tools/>
15. Improvements to scanning whole machine - Syft - Anchore Community - Discourse, accessed May 8, 2025, <https://anchorecommunity.discourse.group/t/improvements-to-scanning-whole-machine/301>
16. syft memory usage is high · Issue #1446 · anchore/syft - GitHub, accessed May 8, 2025, <https://github.com/anchore/syft/issues/1446>
17. Image Layer Details - anchore/grype:v0.86.1 - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/anchore/grype/v0.86.1/images/sha256-a4d05e4fa1b423996ad44fb3f29fb018672092ed9beeb63eac1b0f37c40a443d>
18. Dockerfile - anchore/grype - GitHub, accessed May 8, 2025, <https://github.com/anchore/grype/blob/main/Dockerfile>
19. Chainguard Containers - grype-fips - overview, accessed May 8, 2025, <https://images.chainguard.dev/directory/image/grype-fips/overview>
20. accessed December 31, 1969, <https://raw.githubusercontent.com/anchore/grype/main/Dockerfile>
21. Grype Scanner: Complete Guide to Container Vulnerability Scans - Techiescamp blog, accessed May 8, 2025, <https://blog.techiescamp.com/grype-vulnerability-scanner-guide/>
22. grype configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/beta/descriptors/repository_grype/>
23. View github: anchore/grype | OpenText Core SCA - Debricked, accessed May 8, 2025, <https://debricked.com/select/package/github-anchore/grype>
24. popey/grype2html: Convert Grype vulnerability scan results into interactive HTML reports., accessed May 8, 2025, <https://github.com/popey/grype2html>
25. Anchore Grype - DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/connecting_your_tools/parsers/file/anchore_grype/>
26. Iron Bank / grype-parser · GitLab - Repo One, accessed May 8, 2025, <https://repo1.dso.mil/ironbank-tools/grype-parser>
27. Grype DB Schema Evolution: From v5 to v6 - Smaller, Faster, Better - Anchore, accessed May 8, 2025, <https://anchore.com/blog/grype-db-schema-evolution-from-v5-to-v6-smaller-faster-better/>
28. anchore/grype-db - GitHub, accessed May 8, 2025, <https://github.com/anchore/grype-db>
29. Open-Source Container Security: A Deep Dive into Trivy, Clair, and Grype - Stakater, accessed May 8, 2025, <https://www.stakater.com/post/open-source-container-security-a-deep-dive-into-trivy-clair-and-grype>
30. A MCP server for using Semgrep to scan code for security vulnerabilities. - GitHub, accessed May 8, 2025, <https://github.com/semgrep/mcp>
31. How to Setup Semgrep Rules for Optimal SAST Scanning - Jit.io, accessed May 8, 2025, <https://www.jit.io/resources/appsec-tools/semgrep-rules-for-sast-scanning>
32. semgrep/semgrep - Docker Image | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/r/semgrep/semgrep>
33. Packages in the Semgrep Docker image, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-ci/packages-in-semgrep-docker>
34. Image Layer Details - semgrep/semgrep:latest-nonroot - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/semgrep/semgrep/latest-nonroot/images/sha256-4ff9d4521e76021aa207c5f0b4963f5699bee5c780ac290e786ce2905cad09a0>
35. Image Layer Details - semgrep/semgrep:1.79-nonroot | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/semgrep/semgrep/1.79-nonroot/images/sha256-4632b99be4815dcda9465aed37877fb11bd9c2dfe60e7c7431260dfe847ab75d>
36. semgrep-cli contributing, accessed May 8, 2025, <https://semgrep.dev/docs/contributing/semgrep-contributing>
37. Local CLI scans - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/getting-started/cli-oss>
38. semgrep configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/7/descriptors/repository_semgrep/>
39. Custom rules - Semgrep Secrets, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-secrets/rules>
40. CLI reference - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/cli-reference>
41. Semgrep CE in CI, accessed May 8, 2025, <https://semgrep.dev/docs/deployment/oss-deployment>
42. Installation and first steps | Testing Handbook, accessed May 8, 2025, <https://appsec.guide/docs/static-analysis/semgrep/installation/>
43. Configure blocking findings - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-ci/configuring-blocking-and-errors-in-ci>
44. README.md - Semgrep MCP Server - GitHub, accessed May 8, 2025, <https://github.com/semgrep/mcp/blob/main/README.md>
45. Servers semgrep/mcp - Magic Slides, accessed May 8, 2025, <https://www.magicslides.app/mcps/semgrep>
46. semgrep configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/8/descriptors/repository_semgrep/>
47. Use GitHub repository rulesets to implement Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/kb/semgrep-ci/github-repository-rulesets-semgrep>
48. [ANN] sarif 0.1.0 - Static Analysis Results Interchange Format (SARIF) For OCaml, accessed May 8, 2025, <https://discuss.ocaml.org/t/ann-sarif-0-1-0-static-analysis-results-interchange-format-sarif-for-ocaml/13821>
49. Semgrep - secureCodeBox, accessed May 8, 2025, <https://www.securecodebox.io/docs/scanners/semgrep/>
50. Upgrading Semgrep from OCaml 4 to OCaml 5, accessed May 8, 2025, <https://semgrep.dev/blog/2025/upgrading-semgrep-from-ocaml-4-to-ocaml-5/>
51. semgrep-core contributing, accessed May 8, 2025, <https://semgrep.dev/docs/contributing/semgrep-core-contributing>
52. SpotBugs, accessed May 8, 2025, <https://spotbugs.github.io/>
53. Find Security Bugs, accessed May 8, 2025, <https://find-sec-bugs.github.io/>
54. shibme/findsecbugs-docker - GitHub, accessed May 8, 2025, <https://github.com/shibme/findsecbugs-docker>
55. docker - Official Image - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/_/docker>
56. spotbugs Documentation, accessed May 8, 2025, <https://spotbugs.readthedocs.io/_/downloads/en/issue-1075/pdf/>
57. gradle - Official Image - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/_/gradle>
58. Download - Find Security Bugs, accessed May 8, 2025, <https://find-sec-bugs.github.io/download.htm>
59. Using the SpotBugs Maven Plugin - Read the Docs, accessed May 8, 2025, <https://spotbugs.readthedocs.io/en/latest/maven.html>
60. Integrating findsecbugs plugin with Gradle's findbugs plugin - Stack Overflow, accessed May 8, 2025, <https://stackoverflow.com/questions/34085928/integrating-findsecbugs-plugin-with-gradles-findbugs-plugin>
61. Chapter 8. Filter Files - FindBugs, accessed May 8, 2025, <https://findbugs.sourceforge.net/manual/filter.html>
62. Findbugs Maven plugin - findbugs-exclude with multiple projects - Stack Overflow, accessed May 8, 2025, <https://stackoverflow.com/questions/4036229/findbugs-maven-plugin-findbugs-exclude-with-multiple-projects>
63. Running SpotBugs — spotbugs 4.9.3 documentation - Read the Docs, accessed May 8, 2025, <https://spotbugs.readthedocs.io/en/stable/running.html>
64. How to integrate find-sec-bugs in spotbugs? - java - Stack Overflow, accessed May 8, 2025, <https://stackoverflow.com/questions/67336716/how-to-integrate-find-sec-bugs-in-spotbugs>
65. opensorcery/bandit - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/opensorcery/bandit>
66. Bandit - PyPI, accessed May 8, 2025, <https://pypi.org/project/bandit/>
67. PyCQA/bandit: Bandit is a tool designed to find common security issues in Python code., accessed May 8, 2025, <https://github.com/PyCQA/bandit>
68. secfigo/bandit - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/secfigo/bandit>
69. Getting Started — Bandit documentation, accessed May 8, 2025, <https://bandit.readthedocs.io/en/latest/start.html>
70. bandit configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/v8/descriptors/python_bandit/>
71. Configuration — Bandit documentation - Read the Docs, accessed May 8, 2025, <https://bandit.readthedocs.io/en/latest/config.html>
72. bandit configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/v7/descriptors/python_bandit/>
73. Bandwidth bandit: Understanding memory contention - IEEE Computer Society, accessed May 8, 2025, <https://www.computer.org/csdl/proceedings-article/ispass/2012/06189214/12OmNx5piTn>
74. Increased memory usage after switching from Cowboy to Bandit · Issue #313 - GitHub, accessed May 8, 2025, <https://github.com/mtrudel/bandit/issues/313>
75. Getting Started with ESLint - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/getting-started>
76. pipelinecomponents/eslint - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/pipelinecomponents/eslint>
77. Image Layer Details - onaci/eslint:latest - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/onaci/eslint/latest/images/sha256-82bea8e8d1f12bf1df6e79bcb63764efc3ab849c1b3c865cf4da63c704c2a6b0>
78. Configuration Files - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/configure/configuration-files>
79. Configure Plugins - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/configure/plugins>
80. Resolving eslint Issues in Projects - Layer5 Community Forum, accessed May 8, 2025, <https://discuss.layer5.io/t/resolving-eslint-issues-in-projects/2365>
81. Command Line Interface Reference - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/command-line-interface>
82. ESLint | WebStorm Documentation - JetBrains, accessed May 8, 2025, <https://www.jetbrains.com/help/webstorm/eslint.html>
83. 10 best practices to containerize Node.js web applications with Docker | Snyk Blog, accessed May 8, 2025, <https://snyk.io/blog/10-best-practices-to-containerize-nodejs-web-applications-with-docker/>
84. sonarsource/sonar-scanner-cli - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/sonarsource/sonar-scanner-cli>
85. sonar-scanner-cli-docker/Dockerfile at master - GitHub, accessed May 8, 2025, <https://github.com/SonarSource/sonar-scanner-cli-docker/blob/master/Dockerfile>
86. openComplianceCode/scancode-toolkit - Gitee, accessed May 8, 2025, <https://gitee.com/openComplianceCode/scancode-toolkit>
87. scancode-toolkit - PyPI, accessed May 8, 2025, <https://pypi.org/project/scancode-toolkit/>
88. Comprehensive Installation — ScanCode-Toolkit documentation, accessed May 8, 2025, <https://scancode-toolkit.readthedocs.io/en/stable/getting-started/install.html>
89. Comprehensive Installation - ScanCode Toolkit Documentation, accessed May 8, 2025, <https://scancode-test.readthedocs.io/en/update-rtd-config/getting-started/install.html>
90. scancode-toolkit/INSTALL.rst at develop - GitHub, accessed May 8, 2025, <https://github.com/nexB/scancode-toolkit/blob/develop/INSTALL.rst>
91. scancode-toolkit-image - GitLab, accessed May 8, 2025, <https://gitlab.com/osp-silver/oss/scancode-toolkit-image>
92. aboutcode-org/scancode-toolkit: :mag: ScanCode detects ... - GitHub, accessed May 8, 2025, <https://github.com/nexB/scancode-toolkit>
93. Are you new to Scancode-Toolkit?, accessed May 8, 2025, <https://scancode-toolkit.readthedocs.io/en/latest/getting-started/newcomer.html>
94. pivotal/LicenseFinder: Find licenses for your project's dependencies. - GitHub, accessed May 8, 2025, <https://github.com/pivotal/LicenseFinder>
95. licensefinder/license\_finder - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/licensefinder/license_finder/>
96. Image Layer Details - tabll/license-finder:latest - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/tabll/license-finder/latest/images/sha256-6e5496ca16a689a4a4a019f640e59291c5200b7d3349d60dc45f6caf9cf4d15b?context=explore>
97. Files · add-pnpm-support - license-finder Community Fork - GitLab, accessed May 8, 2025, <https://gitlab.com/gitlab-community/security-products/analyzers/license-finder/-/tree/add-pnpm-support>
98. license\_finder - Versions diffs - 3.0.2 → 3.0.4 - RubyGems - Mend, accessed May 8, 2025, <https://my.diffend.io/gems/license_finder/prev/3.0.4>
99. Sigstore Quickstart with Cosign, accessed May 8, 2025, <https://docs.sigstore.dev/quickstart/quickstart-cosign/>
100. Documentation | in-toto, accessed May 8, 2025, <https://in-toto.io/docs/>
101. engineerd/in-toto-container: Programmatically running in-toto verifications in a container - GitHub, accessed May 8, 2025, <https://github.com/engineerd/in-toto-container>
102. How To Use YARA Rules To Detect Emerging Cloud-Based Threats - Cado Security, accessed May 8, 2025, <https://www.cadosecurity.com/blog/how-to-use-yara-rules-to-detect-emerging-cloud-based-threats>
103. cincan/yara - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/cincan/yara>
104. capa - extract capabilities from executable files - GitHub Pages, accessed May 8, 2025, <https://mandiant.github.io/capa/>
105. capa: Automatically Identify Malware Capabilities | Mandiant | Google Cloud Blog, accessed May 8, 2025, <https://cloud.google.com/blog/topics/threat-intelligence/capa-automatically-identify-malware-capabilities/>