CleanStart™

Shell-less and Read-Only Container Architecture

Enforcing Minimal Runtime with Build-Time Profiling and Hardened Init

Built Clean. Run Secure.

Executive Summary

Most container security approaches focus on scanning images after they are built. In typical workflows, a container image is assembled from multiple base layers, scanned for vulnerabilities, and then patched repeatedly to reduce alerts. This process treats security as a cleanup step rather than a property of the image itself.

A more reliable approach is to reduce the runtime surface before the image is deployed. This requires controlling the contents of the container, limiting the available utilities, and preventing modification of the filesystem at runtime.

CleanStart uses a shell-less and read-only container architecture to minimize runtime attack surface while maintaining application compatibility. Production images do not contain a shell or package manager, and the root filesystem is mounted read-only. Temporary write access is limited to explicitly defined memory-backed paths. These controls are enforced during the build process and validated before the image is deployed.

Achieving both shell-less execution and read-only filesystem enforcement in the same container is technically difficult. Many applications rely on shell behavior, writable paths, or runtime initialization scripts. Removing these components often breaks application startup or prevents graceful shutdown.

CleanStart resolves this limitation with a hardened init process named clnimg-init. This statically compiled binary replaces the traditional shell entrypoint and provides the minimal runtime functionality required for container lifecycle management. Build-time tracing and automated image customization allow applications to run without modification while the runtime surface remains minimal.

This document describes the design of the shell-less and read-only container model, the build-time profiling process, and the runtime enforcement mechanisms used to produce hardened production images.

1. The Challenge of Minimizing Container Runtime Surface

Container images often contain more components than are required to run the application. Base images include shells, package managers, archive tools, and system utilities that are useful during development but unnecessary in production.

These components increase the attack surface and make it harder to verify the contents of the runtime environment. Even when vulnerability scanners are used, the presence of unused utilities generates alerts that must be reviewed and patched.

Two common hardening strategies are used to reduce risk.

The first approach removes the shell and debugging tools from the image. This limits the ability to execute arbitrary commands inside the container, but the filesystem usually remains writable. If an attacker can write a file to a temporary directory, it may still be possible to execute code using the application runtime.

The second approach mounts the root filesystem as read-only. This prevents modification of binaries and configuration files, but the image may still contain a shell or network utilities that can be used for lateral movement or data exfiltration.

Each of these methods improves security, but neither alone provides a minimal runtime environment.

Combining both restrictions in the same container is more difficult because many applications assume that a shell exists or that certain paths are writable during startup

2. The Dependency Paradox

In practice, most container images cannot be both shell-less and read-only without additional engineering.

Applications often rely on shell behavior during startup, even when the shell is not used directly by the developer. Common examples include environment variable expansion, signal handling, PID file creation, and temporary file storage.

If the shell is removed, these behaviors may fail. If the filesystem is locked, the application may crash when it attempts to write to a default path.

For this reason, many container images keep at least one of these capabilities enabled.

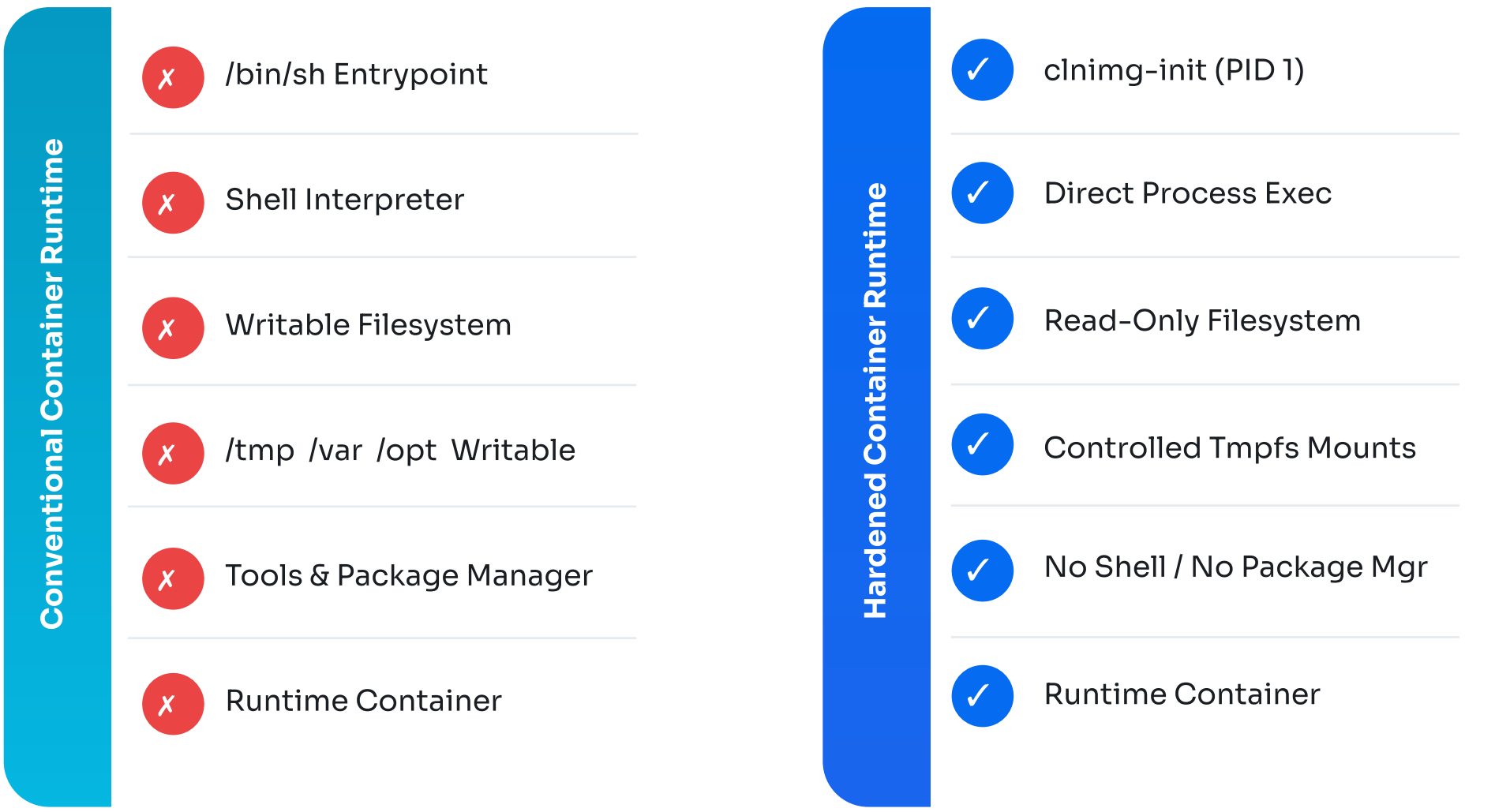
Images that remove the shell often leave the filesystem writable so the application can create temporary files or caches. Images that enforce a read-only filesystem often keep the shell so that initialization scripts and troubleshooting commands continue to work.

This creates a tradeoff between usability and security.

A container that keeps the shell still provides tools that can be used after compromise. A container with a writable filesystem still allows new files to be created at runtime.

Reducing the runtime surface requires solving both problems at the same time without requiring changes to application code.

CleanStart addresses this problem by replacing the shell entrypoint with a hardened init process and by using build-time tracing to determine exactly which paths require write access. The resulting image can run without a shell and without a writable root filesystem while still supporting normal application behavior.



- No Shell
- Read-Only Root
- Tmpfs Mounts
- Minimal Runtime
- Reduced Attack Surface

- Shell Present
- Writable Root
- Tools Available
- Dynamic Behavior
- Larger Attack Surface

Conventional containers often include a shell and writable filesystem, allowing runtime modification and increasing attack surface.

CleanStart containers use a hardened init process, read-only root filesystem, and controlled writable mounts to minimize runtime surface while preserving application behavior.

3. clnimg-init: Replacing the Shell Entry Point

In many containers, the first process started by the runtime is a shell or a script executed through the shell. This is convenient for development, but it introduces unnecessary complexity in production.

To remove the shell without breaking application startup, CleanStart uses a hardened init process called clnimg-init. This binary runs as PID 1 inside the container and performs only the functions required to start and manage the application.

clnimg-init is statically compiled and does not depend on shared libraries. It replaces the traditional shell entrypoint and provides controlled handling of signals, environment variables, and process execution.

When the container starts, clnimg-init prepares the runtime environment and executes the application directly using the operating system process interface. Once the application starts, no shell or interpreter remains in the container.

Signal handling, environment validation, and process execution are handled internally without requiring a shell.

4. Read-Only Root Filesystem Enforcement

Removing the shell reduces the number of tools available at runtime, but it does not prevent modification of the container filesystem. To prevent drift and persistence, the root filesystem must also be mounted read-only.

In the CleanStart model, production containers run with the root filesystem locked. Binaries and configuration files cannot be modified after the image is deployed.

This prevents attackers from writing new files, ensures the runtime matches the built image, and simplifies audit verification.

Applications still require limited write access for temporary files and runtime data. CleanStart identifies required write paths during the build and creates controlled writable mounts at runtime.

5. Build-Time Path Detection

To support a read-only filesystem without modifying application code, CleanStart traces file activity during the build.

The application is executed in a controlled environment while file operations are monitored. The build system records every path that requires write access.

A manifest is generated that defines the minimal writable locations required for the application.

Common paths include:

```
/tmp
/var/run
cache directories
log locations
```

This allows runtime configuration to be created automatically.

6. Controlled Writable Mounts

At runtime, the container starts with the root filesystem mounted read-only. Writable access is granted only to the paths identified during tracing.

Temporary directories use memory-backed mounts. Data is removed when the container stops.

If an application writes to a protected path, the runtime redirects the write to an approved location.

This allows normal behavior without allowing modification of the base filesystem.

7. Development and Production Image Separation

Development images require tools that should not be present in production.

CleanStart produces separate development and production images from the same source.

Feature	Dev	Prod
Shell	Yes	No
Package manager	Yes	No
Debug tools	Yes	No
Filesystem	Writable	Read-only
User	Elevated allowed	Non-root
Runtime surface	Full	Minimal

8. cleanimage-customize Pipeline

The production image is created automatically during the build.

Steps include:

1. tracing runtime behavior
2. removing unused files
3. redirecting writable paths
4. enforcing read-only settings
5. generating SBOM
6. generating runtime manifests

Every production image follows the same rules.

9. Runtime Enforcement in Kubernetes

Runtime configuration enforces the same restrictions used during the build.

Typical settings:

```
readOnlyRootFilesystem: true
runAsNonRoot: true
allowPrivilegeEscalation: false
writable mounts: limited to traced paths only
```

Writable paths use memory storage.

Logging uses stdout instead of files.

Debugging uses temporary containers instead of shells.

10. Operational Impact

Reducing runtime surface results in measurable operational improvements across security, compliance, and operational efficiency.

Metric	Result
Alerts	Reduced
Image size	Smaller
Pull time	Faster
Review effort	Lower
Drift	Eliminated
Compliance	Easier

These improvements come from controlling the image during build.

11. Conclusion

Minimizing container runtime surface requires controlling the image during build and enforcing restrictions at runtime.

By using a hardened init process, read-only filesystem, build-time tracing, and automated image refinement, CleanStart produces containers with predictable and minimal runtime environments.

Applications run without modification, but unnecessary tools and writable paths are removed before deployment.

This approach reduces attack surface and makes container behavior consistent across environments.