CleanStart

BusyBox Replacement

Memory-Safe Container Userspace Foundations

2026.03.23

Abstract

Modern container security discussions frequently focus on application vulnerabilities and dependency scanning, while the container userspace itself remains largely unexamined. In most Linux container images, core utilities such as sh, ls, grep, sed, and tar originate from BusyBox, a multi-call C binary originally designed for embedded systems. Because BusyBox compiles dozens of utilities into a single executable, a vulnerability in one component affects the entire userspace.

This design introduces structural risk in container environments where images are inherited transitively, rebuilt frequently, and deployed at scale. BusyBox is commonly included through base images such as Alpine Linux and may be present in production containers even when developers did not intentionally select it. When vulnerabilities are discovered, remediation requires rebuilding every dependent image, and propagation delays can leave large fleets exposed.

Conventional approaches attempt to mitigate this risk through patching, runtime scanning, or distribution updates. These controls operate after the userspace has already been assembled and do not change the underlying architecture.

CleanStart replaces BusyBox with a memory-safe Rust workspace named CSU (CleanStart Utils) and enforces its use during image construction through policy-driven builds. Development and production userspaces are separated, utilities are statically compiled, and disallowed binaries are blocked before an image can enter the registry.

By embedding userspace control into the container build foundation, CleanStart converts runtime hardening from a procedural safeguard into an infrastructure property. Production images contain only verified, memory-safe utilities, and legacy C userland components cannot be introduced through dependency inheritance, package installation, or build tooling.

The Container Userspace as a Trust Boundary

Every container image includes a userspace layer containing shells, core utilities, and system tools required for execution. This userspace forms part of the runtime trust boundary between the application and the host. If the userspace contains unsafe or unverified components, the integrity of the container cannot be assured even when the application itself is secure.

In most container ecosystems, the userspace is inherited from a base image rather than built explicitly for the application. Alpine Linux is widely used because of its small footprint, and Alpine ships BusyBox as its default utility implementation. As a result, a large percentage of production containers include BusyBox implicitly.

Because container images are layered and reused across pipelines, a single userspace decision can propagate across hundreds of downstream images. When vulnerabilities exist in shared utilities, the impact extends beyond a single application and affects every container that inherits the base layer.

Userspace utilities are rarely reviewed during security assessment. They are assumed to be safe because they are part of the distribution, even though they execute inside the container with the same privileges as the application.

In modern supply chain threat models, the userspace must be treated as part of the trusted computing base. If its contents are not deterministic, auditable, and minimal, downstream controls such as scanning or runtime monitoring cannot fully restore trust.

Structural control over the userspace is therefore required to ensure that production containers contain only intended components and that unsafe utilities cannot enter the image through inheritance or build tooling.

Failure Patterns in Conventional Container Userspace

Conventional container images prioritize size and convenience rather than deterministic security properties. Several structural weaknesses recur across most distributions that rely on BusyBox or similar general-purpose utility sets.

Multi-call Monolithic Utilities

BusyBox implements dozens of commands as a single compiled binary. Utilities such as sh, ls, grep, awk, and tar are symbolic links to the same executable, which dispatches behavior based on the name used to invoke it.

This design has several consequences:

- all utilities share the same address space
- all utilities share the same compiled code
- a vulnerability in one applet affects every tool
- patching requires rebuilding the entire binary

In container environments, this increases the blast radius of parsing errors, memory corruption, or input validation flaws. A defect in one rarely used utility can expose the shell, file tools, and process execution logic in the same container.

Memory-Unsafe Implementation

BusyBox is written in C and processes untrusted input in multiple utilities, including shell parsing, archive extraction, text processing, and network retrieval.

Memory-unsafe languages allow classes of errors that cannot be prevented at compile time, including:

- buffer overflows
- use-after-free
- out-of-bounds reads
- integer overflow leading to memory corruption

Because BusyBox aggregates many utilities into one binary, these risks accumulate within a single executable that is present in most containers. When such a binary is included in production images, the attack surface expands even if the application itself does not require those utilities.

Transitive Inheritance of Utilities

BusyBox is rarely added directly to an application image.
It is inherited through base images, package manager dependencies, or build tooling.

Common inheritance paths include:

- Alpine base images
- package installation during build
- copied filesystem layers
- toolchains left in runtime images

Because inheritance is implicit, security review often focuses on application dependencies while ignoring the userspace layer. Vulnerable utilities may remain present even when the application does not use them.

This makes it difficult to guarantee that production images contain only intended components.

Patch Propagation Delay

When a vulnerability is disclosed in BusyBox, remediation requires multiple steps across the supply chain:

1. upstream patch release
2. distribution package update
3. base image rebuild
4. application image rebuild
5. pipeline execution
6. deployment rollout

Each stage introduces delay. During that time, vulnerable binaries may remain in production containers.

In large environments with many images, verifying that every container has been rebuilt is difficult without deterministic build controls.

Runtime Surface Drift

Utilities required only during build or debugging frequently remain in the final runtime image.

Examples include:

- shells
- archive tools
- package managers
- compilers
- networking utilities

These tools increase the number of executable paths available inside the container and complicate audit review because the runtime surface is larger than necessary.

Minimal runtime images are widely recommended, but conventional build workflows make it difficult to enforce minimality consistently. Without structural controls, runtime contents depend on developer discipline rather than system guarantees.

Deterministic Userspace Architecture

Eliminating risk from the container userspace requires structural control over which utilities are present, how they are built, and when they are allowed to enter an image. Post-build scanning, patch management, and runtime monitoring do not provide sufficient guarantees because they operate after the userspace has already been assembled.

A deterministic userspace architecture establishes the following properties:

- utilities are selected explicitly rather than inherited implicitly
- production images contain only the minimum required runtime components
- development images may include additional tools without affecting production
- all utilities are compiled from verified source
- disallowed binaries are blocked before the image is published
- the contents of the userspace can be verified at build time

To achieve these properties, the default userspace provided by common base images cannot be accepted as-is. BusyBox and similar multi-call utility sets introduce a shared binary dependency that expands the attack surface and makes deterministic control difficult.

Rather than attempting to restrict BusyBox through patching or configuration, the userspace must be replaced with an implementation designed for container environments and integrated with the image build process.

CleanStart implements this through a Rust workspace named CSU (CleanStart Utils).
CSU replaces BusyBox entirely and is built as part of the image construction pipeline so that legacy utilities cannot be introduced through dependency inheritance, package installation, or copied filesystem layers.

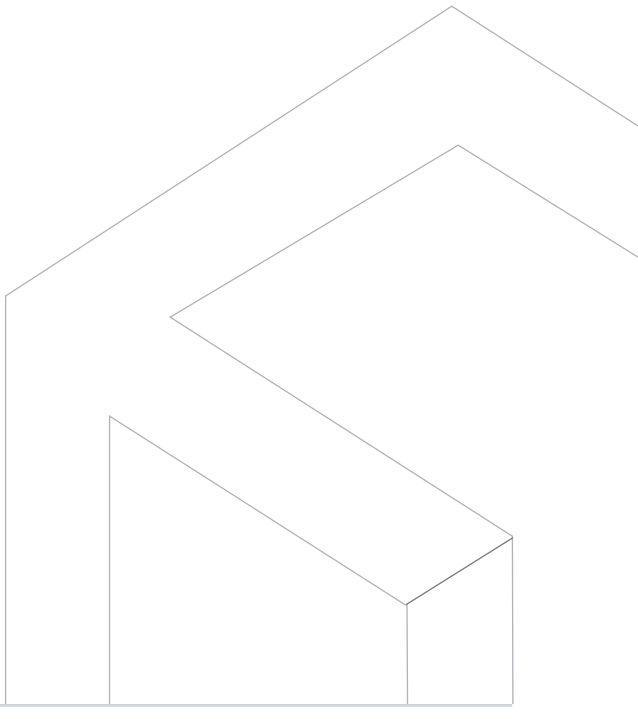
The deterministic userspace model is based on three principles:

- replace monolithic utilities with modular components
- separate development and production userspaces
- enforce policy during image construction

Key properties of the resulting architecture:

- no BusyBox in production images
- no C-based shell in production images
- no dynamically linked userland binaries unless explicitly required
- no build tools present at runtime
- no unverified utilities introduced through base images

Because these properties are enforced during build, they do not depend on developer discipline or manual review.



Conventional Container Userspace

- ⚠ BusyBox inherited implicitly
- ⚠ Memory-unsafe C utilities
- ⚠ Dynamic linking
- ⚠ Tools remain in runtime
- ⚠ Non-deterministic contents

Deterministic Container Userspace (CleanStart)

- ✓ No BusyBox
- ✓ Memory-safe Rust utilities
- ✓ Static linking
- ✓ Minimal runtime
- ✓ Deterministic contents
- ✓ Build-time enforcement

CSU Workspace Design

CSU (CleanStart Utils) is a modular Rust workspace that replaces BusyBox and other general-purpose utility sets. The goal of CSU is not to provide a full Linux distribution userland, but to provide a controlled, auditable set of utilities suitable for container build and execution.

BusyBox combines dozens of utilities into a single C executable using a multi-call architecture. This reduces binary size but causes all utilities to share the same memory space and update cycle. A vulnerability in one component can affect every tool in the image.

CSU uses a different design.

Utilities are implemented as independent Rust modules compiled into a workspace that separates shared logic, development tools, and production runtime components.

The workspace consists of three primary crates.

Component	Purpose	Deployment
libcleanstart	Shared parsing and execution logic	Dev & Prod
cleanimg_init	Minimal runtime process for production	Production
cleanstart-utils	Full utility set and shell	Development

This separation allows development environments to remain usable while production images remain minimal and deterministic.

The workspace design ensures that development convenience does not expand the runtime attack surface.

libcleanstart : shared execution and parsing engine

The libcleanstart crate contains the core logic used by all utilities.
Text processing, environment handling, argument parsing, and execution behavior are implemented in a shared library so that development and production components use the same code paths.

Design characteristics:

- feature-flag controlled compilation
- minimal dependency surface
- no dynamic linking required
- deterministic behavior across builds
- shared implementation between dev and prod

Feature flags allow individual binaries to include only the functionality they require.
For example, the production runtime may include environment expansion but exclude all shell and file utilities.

Because both cleanstart-utils and cleanimg_init depend on the same library, behavior remains consistent across environments. Scripts that work in development do not fail in production due to differences in utility implementations.

This avoids a common failure pattern in container builds where the development environment and runtime environment diverge.

cleanimg_init : Minimal Production Runtime

Production images use cleanimg_init as the container entry process instead of a shell or full utility set.

The purpose of cleanimg_init is limited to:

- environment initialization
- minimal variable expansion
- execution of the application process
- signal handling required for PID 1

Only the required modules from libcleanstart are compiled into the binary.

The following components are intentionally excluded from production images:

- shell interpreter
- archive tools
- networking utilities
- package managers
- build tools
- dynamically linked system utilities

This keeps the runtime surface small and reduces the number of executable paths available inside the container.

Because the runtime binary is statically compiled with feature flags, its contents are known at build time and cannot change at runtime.

cleanstart-utils : Development Utility Set

Development images include cleanstart-utils, which provides a multi-call utility binary similar in interface to BusyBox but implemented as independent Rust modules.

Utilities available in development images may include:

- shell
- ls
- grep
- sed
- tar
- env handling
- file utilities

Each utility is implemented as a separate module compiled into the binary. The dispatcher selects the correct module based on the program name used to invoke the executable.

Although a multi-call interface is used for compatibility with existing scripts, modules do not share mutable state.

Rust's ownership and borrowing rules enforce memory safety at compile time, preventing entire classes of errors that are common in C-based utility implementations.

A defect in one utility cannot corrupt the state of another, even though they are packaged in the same executable.

This preserves the convenience of a multi-call interface without inheriting the security risks of a monolithic C binary.

Static linking and deterministic binaries

All CSU components are statically compiled.

Static linking was chosen to eliminate runtime dependency on shared system libraries and to ensure that the contents of each binary are fully known at build time.

Advantages of static linking include:

- no runtime library substitution
- no dependency on host-provided libraries
- no LD_PRELOAD injection surface
- predictable binary contents
- simplified audit verification

Production images therefore do not contain dynamically linked userland utilities unless explicitly required by the application.

This reduces both attack surface and operational complexity.

Shell Execution Model

Development containers require a shell, but traditional shells introduce a large parsing surface and complex execution behavior that is difficult to audit. BusyBox ash and other C-based shells contain decades of compatibility logic, optional features, and dynamic behavior that are unnecessary in container workflows.

CSU includes a minimal shell designed specifically for container environments.

The shell is implemented in Rust and executes commands using direct process invocation rather than a legacy parser architecture.

Design goals:

- predictable execution behavior
- minimal parsing surface
- auditable command invocation
- compatibility with common container scripts
- no unsafe extensions

Because the shell is implemented in Rust, memory safety violations that affect traditional C shells are eliminated at compile time.

The shell is included only in development images.
Production images do not contain a shell unless explicitly required by policy.

Policy Enforcement at Build Time

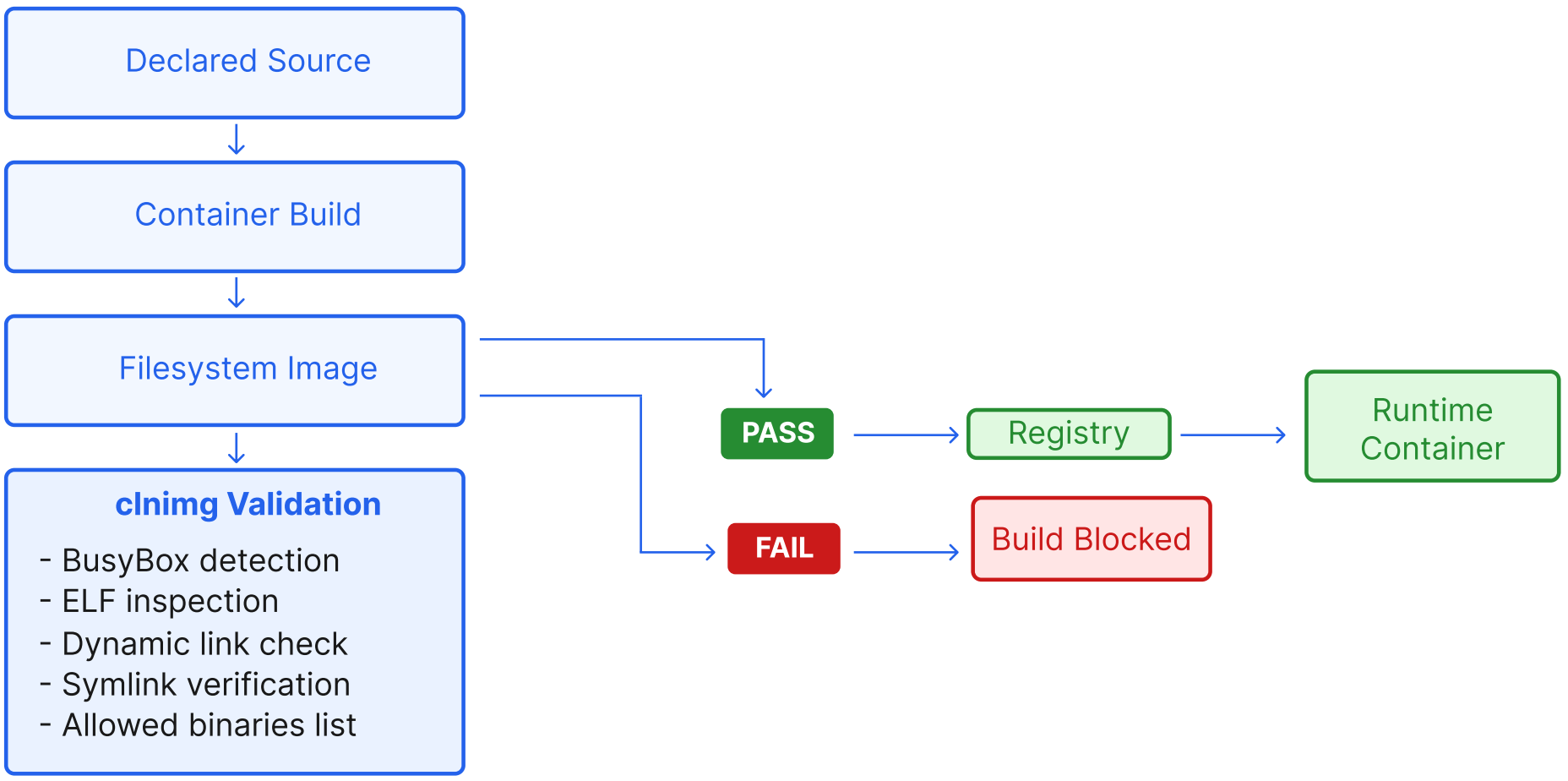
Replacing BusyBox alone does not guarantee a secure userspace.
In conventional container builds, utilities can re-enter the image through package dependencies, copied layers, or build tooling. Without structural enforcement, a single command in a Dockerfile can reintroduce the same components the architecture was intended to remove.

To prevent this, userspace control must be enforced during image construction rather than verified after the image is built.

CleanStart implements build-time enforcement through clnimg, a policy-driven image construction tool that validates the contents of an image before it can be published to a registry.

During the build process, clnimg performs a recursive inspection of the filesystem being packaged. Validation operates at the binary level rather than relying on package metadata, ensuring that renamed or repackaged utilities cannot bypass policy checks.

Figure 2 — Build-Time Image Policy Enforcement



The build fails automatically if disallowed binaries are detected.
An image that violates policy cannot be pushed to the registry.

This converts userspace control from a procedural safeguard into a deterministic property of the build system.

Binary signature detection

BusyBox and similar utilities may appear under different names or may be included as static binaries. Package-level checks are not sufficient to detect all cases.

clnimg inspects executable files directly, including ELF headers and binary contents, to identify disallowed utilities regardless of filename.

Detection includes:

- multi-call dispatch tables
- BusyBox string signatures
- binary format inspection
- executable path validation

Because inspection operates on the compiled filesystem, renamed or statically embedded binaries cannot bypass the policy.

This ensures that BusyBox and other C-based utilities cannot be introduced through base images, dependencies, or manual file copies

Dynamic linking audit

Production images are expected to contain statically compiled utilities unless dynamic linking is explicitly required.

clnimg verifies that executables in system paths are not linked against standard shared libraries such as libc.so.

Dynamic linking increases attack surface because runtime behavior depends on external libraries that may vary between environments.

During validation:

- executables in /bin, /usr/bin, and related paths are inspected
- shared library dependencies are enumerated
- disallowed runtime libraries cause the build to fail

This ensures that the production userspace remains minimal, deterministic, and independent of host-provided libraries.

Symbolic link verification

Multi-call utilities rely on symbolic links to provide different command names.
Policy enforcement must verify that these links resolve only to allowed executables.

clnimg validates symbolic links for system utilities, including:

- /bin/sh
- /bin/ls
- /bin/grep
- /bin/sed

In development images, these links must resolve to cleanstart-utils.

In production images, these links must not exist unless explicitly allowed.

If a symbolic link points to a disallowed binary, the build fails.

This prevents legacy shells or utilities from being introduced through indirect dependencies.

Pipeline policy enforcement

Image validation rules differ between development and production builds. The build pipeline defines which utilities are permitted in each context.

Typical policy rules:

Development pipeline:

- allow cleanstart-utils
- allow CSU shell
- block BusyBox
- block ash, dash, and other C-based shells unless explicitly allowed

Production pipeline:

- allow cleanimg_init
- block BusyBox
- block cleanstart-utils
- block shells
- block general-purpose utilities

Because validation occurs during image construction, developers cannot bypass policy by modifying the Dockerfile or installing additional packages.

The build tool itself becomes the enforcement point.

This makes it programmatically impossible for disallowed utilities to reach a production registry.

Compliance event logging

Each validation result is recorded as part of the build process.

Logged data may include:

- image identifier
- binary hashes
- detected violations
- policy rules applied
- validation outcome

Successful builds record the verified state of the userspace.
Failed builds record the reason for rejection.

These records provide an auditable trail showing that disallowed utilities were blocked before the image entered the registry.

Because validation occurs automatically, audit evidence is generated without manual documentation.

This model supports verification requirements found in modern supply chain and software integrity frameworks.

Development & Production Userspace Separation

Container environments require different utilities during development and runtime.
Development images must include shells and tools, while production images should contain only the components required to execute the application.

Conventional build workflows rely on discipline to keep these environments separate.
In practice, tools used during build frequently remain in the final runtime image.

CleanStart enforces a strict separation between development and production userspaces.

Development images include:

- cleanstart-utils
- CSU shell
- file utilities
- text processing tools
- archive tools

Production images include:

- cleanimg_init
- application binaries
- required runtime libraries only

Utilities present in development images are not permitted in production images unless explicitly allowed by policy.

This separation is enforced during the build process and does not depend on developer behavior.

As a result, production containers remain minimal even when development environments are full-featured.

Operational & Compliance Implications

Deterministic userspace construction improves both runtime security and auditability.

When utilities are selected explicitly and validated during build, the contents of the container become predictable. This reduces the effort required to verify compliance and simplifies vulnerability management.

Operational benefits include:

- smaller runtime images
- fewer executable paths
- reduced vulnerability exposure
- faster rebuild verification
- consistent behavior across environments

From a governance perspective, deterministic userspace control supports requirements found in modern security frameworks, including:

- software integrity validation
- minimal runtime surface controls
- reproducible builds
- verifiable provenance
- controlled dependency inclusion

Because validation occurs automatically during image construction, audit review can rely on build records rather than manual inspection.

Verification shifts from procedural review to artifact validation.

CleanStart Integration

Userspace control is integrated directly into the CleanStart image build foundation.

CSU utilities, policy enforcement, and userspace validation are applied automatically when images are built using CleanStart tooling.

Key properties of the integrated model:

- BusyBox is not present in base images
- CSU replaces default userland utilities
- production images use cleanimg_init
- clnimg enforces policy during build
- validation results are recorded automatically

Conclusion

Container security cannot rely solely on scanning or patching when the userspace itself contains monolithic, memory-unsafe utilities inherited from base images.

BusyBox was designed for embedded systems, but its multi-call architecture and C implementation introduce structural risk when used in modern container environments.

Replacing the userspace with a modular, memory-safe implementation and enforcing that replacement during image construction eliminates this class of risk.

By separating development and production utilities, statically compiling runtime components, and validating image contents at build time, CleanStart establishes deterministic userspace foundations suitable for large-scale container deployments.

Security becomes a property of the build system rather than a post-deployment activity, and production images contain only verified components.

This approach reduces attack surface, simplifies audit review, and provides a consistent runtime baseline across environments.