CleanStart™

Replacing BusyBox in Container Images

Memory-Safe Userspace and Build-Time Enforcement with CleanStart

Built Clean. Run Secure.

1. Executive Summary

BusyBox is one of the most widely deployed binaries in the Linux container ecosystem. It provides the core userspace utilities used in many container images, especially those derived from Alpine Linux. Because BusyBox implements dozens of commands inside a single compiled binary, a vulnerability in one component can affect the entire userspace.

In modern container environments, images are frequently inherited from base layers, rebuilt through automated pipelines, and deployed at scale. As a result, utilities such as BusyBox may be present in production containers even when they were not intentionally selected. When vulnerabilities are discovered, remediation requires rebuilding every dependent image, which can introduce delays and increase operational risk.

Traditional approaches rely on patching, scanning, or runtime controls to manage this risk. These methods operate after the userspace has already been assembled and do not change the underlying structure of the container image.

CleanStart replaces BusyBox with a memory-safe userspace implemented in Rust and enforces that replacement during image construction. Development and production images are separated, utilities are statically compiled, and policy validation prevents disallowed binaries from entering the final image.

This approach reduces runtime attack surface, simplifies compliance verification, and provides a deterministic container foundation suitable for modern software supply chain security requirements.

2. BusyBox in the Container Ecosystem

BusyBox was originally designed for embedded Linux systems where storage and memory were limited. Instead of shipping separate binaries for each utility, BusyBox compiles many commands into a single executable and dispatches behavior based on the program name used to invoke it. This design reduces binary size, but it also means that all utilities share the same compiled code. In container environments, BusyBox is commonly included through base images such as Alpine Linux. Many container images therefore inherit BusyBox even when developers do not explicitly install it. In a typical Alpine-based container, commands such as the following are symbolic links to the same BusyBox binary:

```
/bin/sh
/bin/ls
/bin/grep
/bin/sed
/bin/tar
```

Because these utilities are implemented inside a single executable, a vulnerability in one function can affect all others. Updating a single utility requires rebuilding the entire binary and redistributing every image that depends on it.

In large container fleets, this can make it difficult to ensure that all runtime images contain the intended versions of system utilities.

2.1 Shared binary architecture

BusyBox uses a multi-call design in which multiple commands are implemented inside one executable. All utilities share the same address space and update cycle.

This design increases the impact of vulnerabilities. A defect in one applet can expose the entire userspace inside the container.

2.2 Memory-unsafe implementation

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

Memory-unsafe languages allow errors such as buffer overflows, use-after-free, and out-ofbounds reads. When these errors occur inside a shared binary, the scope of impact is larger than in modular utilities

2.3 Transitive inheritance

BusyBox is often introduced through base images or package dependencies rather than explicit installation.

Common sources include:

1. Alpine base images
2. Package manager installs
3. Copied filesystem layers
4. Build tools left in runtime images

Because inheritance is implicit, security review may not detect the presence of these utilities until after deployment.

2.4 Patch propagation delay

Fixing a BusyBox vulnerability requires several steps:

1. Upstream release
2. Distribution update
3. Base image rebuild
4. Application rebuild
5. Pipeline execution
6. Deployment rollout

In large environments this process can take time, leaving vulnerable binaries in production images.

2.5 Excess runtime surface

Utilities required only during build or debugging may remain in the final container image.

1. Shells
2. Archive tools
3. Package managers
4. Networking utilities

These tools increase the number of executable paths available inside the container and make compliance verification more difficult.

3. Replacing BusyBox Instead of Patching It

Patching BusyBox reduces known vulnerabilities but does not change the structure of the userspace. Scanning images after they are built can detect problems, but it cannot prevent disallowed utilities from being introduced during the build.

To reduce risk, the userspace must be controlled during image construction.

CleanStart replaces BusyBox with a modular userspace called CSU, short for CleanStart Utils. CSU is implemented in Rust and designed specifically for container environments.

The goals of this design are:

1. Minimize runtime surface
2. Keep development environments usable
3. Ensure deterministic image contents
4. Enforce policy during buil
5. Eliminate memory-unsafe utilities from production images

4. CSU Architecture

CSU is organized as a Rust workspace with separate components for shared logic, development utilities, and production runtime.

Component	Purpose	Used In
libcleanstart	Shared parsing and execution logic	Dev and Prod
cleanstart-utils	Development utilities and shell	Dev
cleanimg_init	Minimal runtime process	Prod

This separation allows development images to include tools while production images remain minimal.

All CSU components are statically compiled. Static linking removes runtime dependency on shared libraries and makes binary contents predictable

5. Development and Production Images

Development images include utilities required for building and testing applications.
Production images include only the components required to run the application.

Development images may contain:

1. Cleanstart-utils
2. Shell
3. File utilities
4. Archive tools

Production images contain:

1. Cleanimg_init
2. Application binaries
3. Required runtime libraries

This separation reduces attack surface without affecting developer workflow

6. Build-Time Policy Enforcement with clnimg

Replacing BusyBox alone is not sufficient. Utilities can re-enter the image through dependencies or manual changes.

CleanStart enforces userspace policy during image construction using clnimg.

During the build, clnimg inspects the filesystem that will become the container image. Validation is performed on the compiled binaries rather than package metadata.

Checks include:

1. BusyBox detection
2. ELF inspection
3. Dynamic linking verification
4. Symbolic link validation
5. Allowed executable list

If a violation is detected, the build fails and the image is not produced.

Because validation occurs before the image is pushed to the registry, disallowed utilities cannot reach production.

7. Compliance and Audit Benefits

Deterministic userspace construction simplifies audit and compliance verification.

Because utilities are validated during build, the contents of the container are known before deployment.

Benefits include:

1. Reduced runtime surface
2. Predictable image contents
3. Easier vulnerability tracking
4. Verifiable build process
5. Consistent runtime behavior

This supports requirements found in modern security and supply chain frameworks that require integrity validation and controlled dependencies.

8. Conclusion

BusyBox was designed for constrained environments, but its multi-call architecture and memoryunsafe implementation introduce risk in modern container deployments.

Replacing the default userspace with modular, memory-safe utilities and enforcing policy during image construction reduces runtime attack surface and improves consistency across environments.

By separating development and production images and validating image contents before registry push, CleanStart provides a deterministic container foundation suitable for secure software supply chains.