WHITEPAPER

Container Hardening on a Minimal Host: The Split-Runtime Security Model

Applying trust-domain separation, service manager sandboxing, and layered runtime policy to harden containerized workloads at the edge

April 2026

nova8 Technologies

nova8.io

Table of Contents

1. Executive Summary

Container technologies have become the dominant model for packaging and deploying application workloads. According to NIST Special Publication 800-190, containers provide a portable, reusable, and automatable way to package and run applications, but they also introduce security concerns that require deliberate architectural countermeasures.

At the edge, these concerns are amplified. Edge devices often operate in physically exposed environments, with limited administrative access and constrained recovery options. A container escape on a cloud server is serious; a container escape on an unattended edge device in a contested environment can compromise the entire deployment.

This whitepaper examines a layered approach to container hardening on minimal host operating systems. It describes the split-runtime architecture pattern (separating trusted management functions from less-trusted application workloads), the use of standard Linux service manager sandboxing to constrain runtime processes, and the application of OCI-standard policy enforcement to validate container configurations before execution.

The paper draws on publicly documented standards and guidance, including NIST SP 800-190, the OCI Runtime Specification, systemd sandboxing documentation, and publicly available container-specific operating system designs. It is intended for platform architects, security engineers, and program managers evaluating container security for edge deployments in defense, critical infrastructure, and industrial environments.

2. The Container Security Challenge at the Edge

Containers provide operating system level virtualization by leveraging Linux kernel features including namespaces (for resource isolation), cgroups (for resource allocation), and security mechanisms such as seccomp and mandatory access controls. Unlike virtual machines, containers share the host kernel, which provides efficiency but also means the kernel itself is part of the attack surface for every container on the system.

NIST SP 800-190 identifies several categories of container-specific risk. Image vulnerabilities can be introduced through outdated base images or unpatched dependencies. Registry compromises can distribute tampered images. Insecure runtime configurations, such as running containers with elevated privileges or mounting sensitive host directories, can allow a compromised container to affect the host. The shared kernel means that kernel-level vulnerabilities can potentially be exploited from within any container.

The container escape threat is well documented. CVE-2019-5736 demonstrated that an attacker with root access inside a container could overwrite the host's runc binary through the /proc filesystem, gaining full host access. CVE-2022-0492 showed that root inside a container could exploit cgroups v1 to escape container boundaries. These are not theoretical risks; they represent demonstrated attack paths against real container runtimes.

At the edge, several factors make these risks more consequential.

- Physical exposure increases the likelihood that an attacker can interact with a device directly, making container escapes a path to host compromise in environments where physical security is limited.
- Remote management constraints mean that incident response is slower and more expensive. A compromised edge device may not be accessible for manual remediation for hours, days, or longer.
- Fleet-scale deployment means that a vulnerability affecting one device likely affects many. The blast radius of a single container escape can extend across an entire fleet if all devices run the same workload stack.
- Long deployment lifetimes mean edge devices must remain defensible against threats discovered well after initial deployment. Hardening measures must remain effective over years of field operation.

For these reasons, edge container security cannot rely solely on the default isolation provided by the container runtime. It requires deliberate architectural choices that reduce the attack surface, limit the blast radius of compromise, and enforce policy at multiple independent layers.

3. Minimal Host Operating Systems: Reducing the Foundation

NIST SP 800-190 recommends using container-specific host operating systems instead of general-purpose ones to reduce attack surfaces. The guidance states: "A container-specific host OS is a minimalist OS explicitly designed to only run containers, with all other services and functionality disabled, and with read-only file systems and other hardening practices employed."

Several publicly available container-specific operating systems demonstrate this principle in practice.

Flatcar Container Linux is designed from the ground up for running container workloads. It features an immutable, read-only filesystem, no package manager, and automated atomic updates. The Flatcar project describes its design philosophy: "Only the essential tools to run your containers, eliminating bloat. No package manager, no configuration drift."

Talos Linux takes minimalism further by removing even SSH access and interactive shells. It is managed entirely through a declarative API, and its filesystem is immutable. The Talos documentation describes it as "designed to be as minimal as possible while still maintaining practicality."

AWS Bottlerocket is a container-optimized operating system that uses a read-only root filesystem, automated updates, and a minimal set of host components. Its API-driven configuration model eliminates the need for SSH access in production.

The common design principles across these systems are consistent.

- Read-only or immutable root filesystems prevent runtime modification of the host.
- No package manager eliminates a major source of configuration drift and reduces the binary surface available to an attacker.
- Minimal installed components reduce the number of potential vulnerabilities and the tools available for post-exploitation.
- Automated, atomic updates ensure the host can be replaced rather than patched in place.

These properties create a stronger foundation for container workloads because the host itself has fewer paths through which it can be compromised or modified. The host becomes a constrained, verifiable substrate rather than a general-purpose computing environment.

4. The Split-Runtime Architecture Pattern

On most container hosts, a single runtime instance manages all containers: both system management workloads (fleet agents, update controllers, monitoring) and user application workloads. This means that a compromise of any application container shares the same runtime context as the management plane. If an attacker escapes a user workload, they land in the same runtime domain that controls the host.

The split-runtime pattern addresses this by running distinct runtime instances for different trust domains (see Figure 1). Management functions that require elevated host access operate in one runtime instance with appropriate privileges. Application workloads operate in a separate, more heavily restricted runtime instance with its own storage, policy, and privilege boundaries.

This separation follows the principle of least privilege and the security concept of blast-radius reduction. The management runtime needs access to host resources for tasks like image updates, fleet coordination, and system health reporting. Application workloads, which may run third-party or less-trusted code, should not have that same access.

The key properties of this pattern are straightforward.

- Separate runtime instances mean that workloads in one domain cannot see or interact with containers in the other domain through the runtime API.
- Separate storage ensures that container images, layers, and state for each domain are isolated. A compromised workload cannot tamper with management container images.
- Different privilege profiles allow the management runtime to operate with the host access it genuinely needs while the workload runtime operates under strict restrictions.
- Trust boundary enforcement means that even if an attacker compromises the workload runtime entirely, they have not gained access to the management plane.

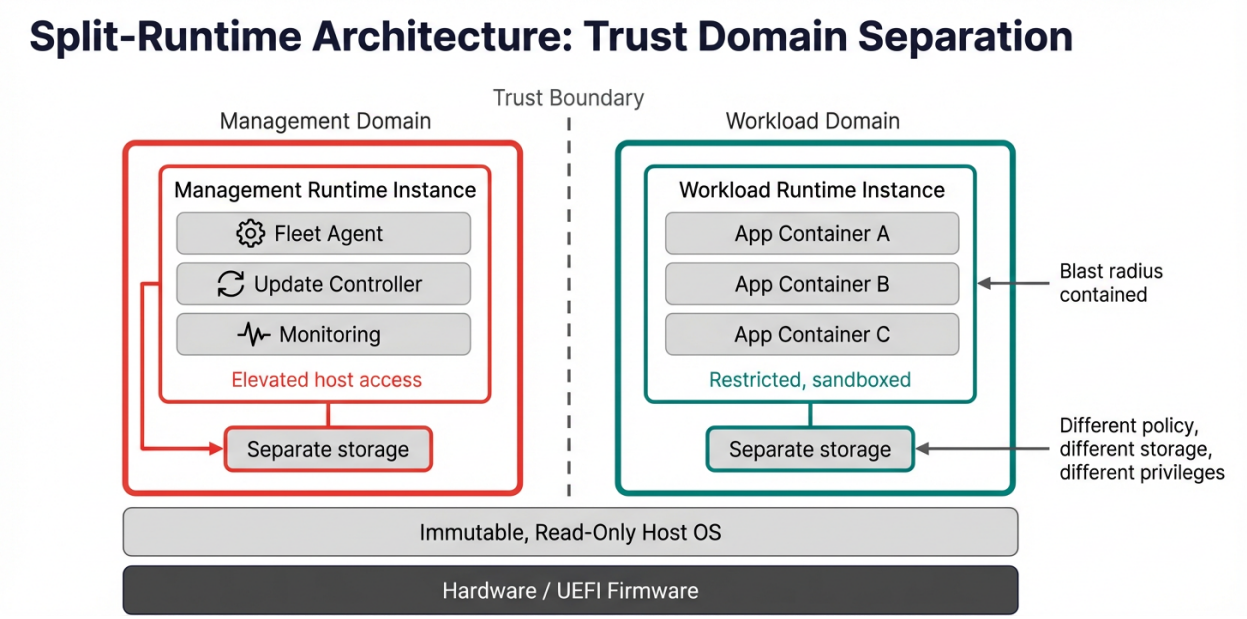

Figure 1: The split-runtime architecture pattern separates management and workload trust domains.

This is not a novel concept in security architecture. It applies the same trust-domain separation principles used in network segmentation, microservice isolation, and defense-in-depth design to the container runtime layer. The difference is that most container deployments do not implement it, because general-purpose server environments have not historically needed this level of host-level isolation.

At the edge, where the blast radius of a compromise is harder to contain through network controls alone, this pattern becomes a practical necessity.

5. Sandboxing Runtime Services with Standard Linux Controls

The systemd service manager, which is the standard init system on most modern Linux distributions, provides a comprehensive set of sandboxing directives that can constrain the behavior of any managed service, including container runtime processes. These directives are publicly documented in the systemd.exec manual and have been described in detail by Red Hat, the Arch Linux Wiki, and the freedesktop.org project.

The Red Hat blog on systemd security notes: "systemd provides a significant number of security features that can be used to isolate services and applications from each other as well as from the underlying operating system. In many cases, systemd provides easy access to the same mechanisms provided by the Linux kernel that are also used to create isolation for Linux containers."

For a container runtime service managing untrusted workloads, several categories of sandboxing controls are particularly relevant.

Read-Only System Protection:

The ProtectSystem=strict directive mounts the entire filesystem hierarchy read-only for the sandboxed service, except for explicitly allowed paths. This means that even if the container runtime itself is compromised, it cannot modify the host operating system, bootloader, or system configuration. Specific writable paths can be allowed through ReadWritePaths for container storage locations that genuinely need write access.

Device Access Restrictions:

The DeviceAllow directive controls which hardware devices a sandboxed service can access. For a workload runtime, device access can be restricted to only the device classes that application containers legitimately need (such as GPUs for compute workloads), while blocking access to raw block devices, system buses, and other hardware that could be used for host compromise.

Execution Controls:

The NoExecPaths directive prevents binary execution in specified directories. When applied to temporary and writable paths, this stops a common attack pattern where a compromised process writes a malicious binary to a writable location and then executes it. Combined with MemoryDenyWriteExecute, which prevents simultaneous write and execute permissions on memory pages, this significantly reduces the available exploitation surface.

Kernel Protection:

Directives like ProtectKernelModules (preventing module loading), ProtectKernelTunables (preventing modification of /proc and /sys), and ProtectControlGroups (preventing cgroup modification) close off entire categories of privilege escalation paths. The systemd documentation recommends enabling these for most services that do not need to modify kernel behavior.

The systemd-analyze security tool provides a quantitative assessment of how well a service unit uses these controls. Red Hat's documentation notes that most default service configurations score as "UNSAFE" on this assessment, not because the services are inherently insecure, but because they are not taking advantage of available sandboxing. Applying even basic sandboxing directives typically reduces the exposure score significantly.

The value of systemd sandboxing for container runtimes is that it provides an independent enforcement layer that operates outside the container boundary. Even if a container escape succeeds, the escaped process finds itself constrained by the systemd sandbox applied to the runtime service. This is defense in depth applied at the service manager level.

6. Policy Enforcement at Container Creation Time

The OCI Runtime Specification defines a set of lifecycle hooks that allow custom logic to execute at specific points during a container's creation and operation. These hooks, documented in the OCI runtime-spec and in the containers/common project's oci-hooks specification, provide a standards-based mechanism for enforcing security policy before a container reaches execution (see Figure 2).

The Red Hat blog on OCI hooks for Podman describes these stages: "precreate hooks are executed after the container configuration is generated, but before the actual start of the container." This timing is critical for security, because it allows a policy engine to inspect and validate the container's requested configuration before any container process runs.

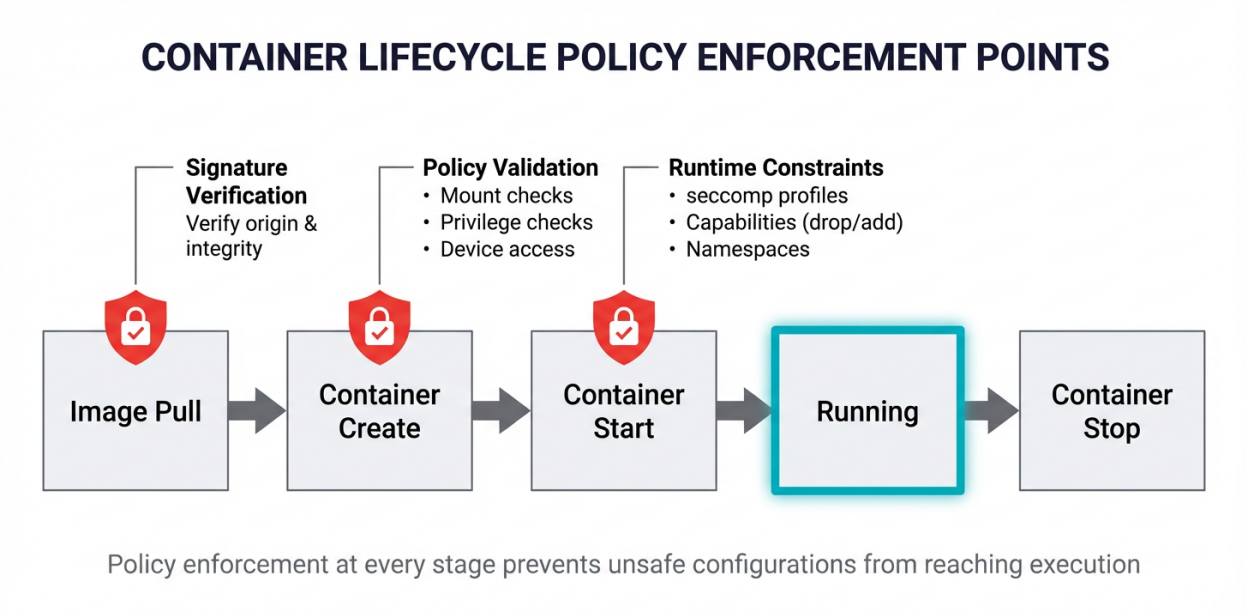

Figure 2: Policy enforcement at each stage of the container lifecycle prevents unsafe configurations from reaching execution.

A precreate hook receives the full container configuration as JSON on standard input and can approve, modify, or reject the configuration before the container runtime proceeds. This provides an admission control mechanism analogous to Kubernetes admission webhooks, but operating at the individual runtime level rather than the orchestrator level.

For edge deployments, precreate policy enforcement is particularly valuable because it can operate without connectivity to a central policy server. The policy logic runs locally on the device as part of the container runtime pipeline, making it effective in disconnected, denied, or intermittent network conditions.

The types of policy checks that can be enforced at container creation time include the following.

- Mount validation: verifying that requested host-path bind mounts are limited to an approved set rather than allowing arbitrary host filesystem access.
- Privilege checks: ensuring containers do not request capabilities, privileged mode, or host namespace access beyond what is authorized for the workload.
- Device access validation: confirming that requested device mounts are limited to approved device classes.
- Image provenance: verifying that the container image comes from a trusted registry and carries a valid signature.

The default-deny approach, where all host-path mounts are blocked unless explicitly approved, is more robust than a blocklist approach that attempts to enumerate and block known-dangerous paths. A blocklist can be bypassed through symlinks, bind mounts to parent directories, or newly created paths that were not anticipated when the policy was written.

NIST SP 800-190 reinforces this principle: "Containers should rarely make changes to the host OS file system and should almost never make changes to locations that control the basic functionality of the host OS." Policy enforcement at creation time provides a mechanism to ensure this guidance is followed consistently across an entire fleet.

7. Defense in Depth: Layered Container Security

No single security mechanism is sufficient to protect containerized workloads. The defense-in-depth model (see Figure 3) combines multiple independent security layers so that the failure of any one layer does not result in complete compromise. For container hardening on a minimal host, these layers operate from the outside in.

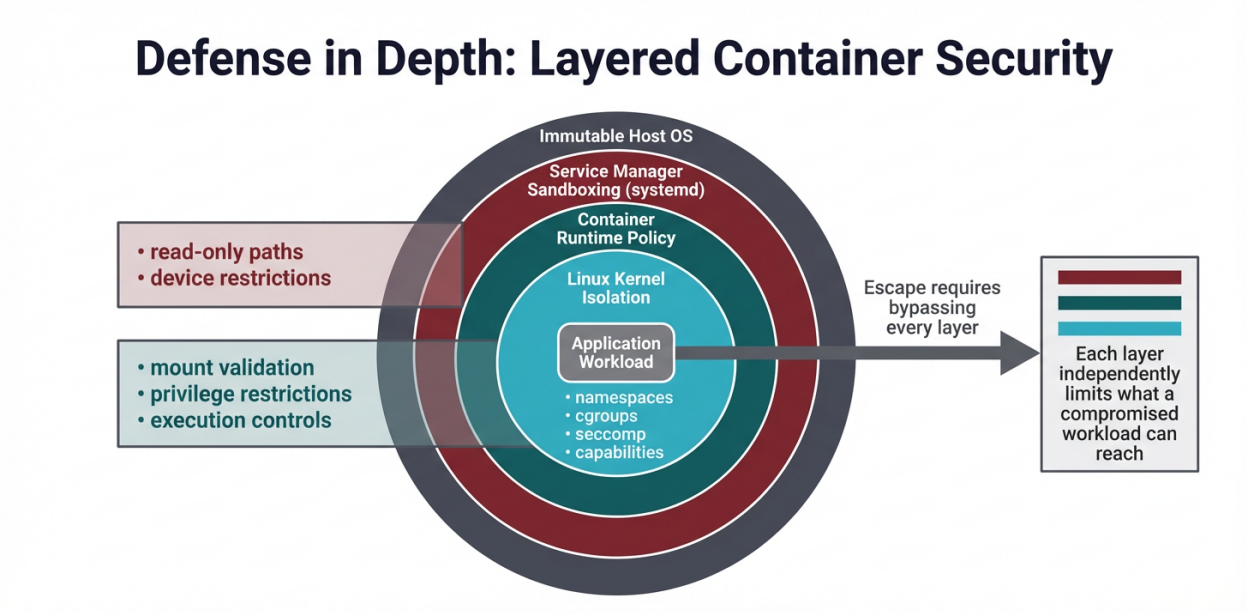

Figure 3: Defense in depth for container security, with each layer independently limiting what a compromised workload can reach.

Layer 1: Immutable Host Operating System

The outermost layer is the host itself. A minimal, immutable host with a read-only root filesystem eliminates entire classes of persistent compromise. An attacker who escapes a container finds a host with no package manager, no compiler, no interactive shell, and no writable system paths. The tools typically used in post-exploitation scenarios are simply not present.

Layer 2: Service Manager Sandboxing

The container runtime process itself is sandboxed by the service manager (systemd). Read-only system paths, device restrictions, execution controls, and kernel protection settings constrain what the runtime process can do, even if it is compromised. This layer operates independently of the container isolation mechanisms.

Layer 3: Container Runtime Policy

OCI lifecycle hooks and runtime configuration enforce policy at container creation time. Mount restrictions, privilege limits, and device access controls prevent unsafe container configurations from reaching execution. This layer ensures that policy violations are caught before the container starts, not after.

Layer 4: Linux Kernel Isolation

The innermost layer uses kernel-provided isolation mechanisms. Namespaces separate filesystem, network, process, user, and IPC visibility. Cgroups limit CPU, memory, and I/O consumption. Seccomp profiles restrict available system calls. Linux capabilities are dropped to the minimum required set. Mandatory access control systems (AppArmor or SELinux) provide additional filesystem and process restrictions.

Each of these layers operates independently. A container escape bypasses Layer 4 but still encounters Layers 2 and 3. Compromising the runtime process bypasses Layers 3 and 4 but still encounters the immutable host and service manager sandbox at Layers 1 and 2. The more layers an attacker must bypass, the higher the cost and complexity of a successful compromise.

NIST SP 800-190 recommends this approach: "mandatory access control technologies like SELinux and AppArmor provide enhanced control and isolation for containers. Organizations are encouraged to use the MAC technologies provided by their host OSs in all container deployments." When combined with the other layers described above, MAC enforcement becomes part of a comprehensive defense rather than a standalone control.

8. Evaluating Container Hardening in Edge Platforms

For teams evaluating edge platforms, the following questions can help assess whether a platform's container hardening architecture provides the layered protection needed for high-consequence deployments.

Runtime Architecture:

- Does the platform separate management and workload runtime domains?
- Can a compromised application container access the management plane?
- Are runtime instances isolated with separate storage and privilege profiles?

Host Hardening:

- Is the host operating system minimal and purpose-built for containers?
- Is the root filesystem read-only or immutable?
- Are interactive shells, package managers, and unnecessary utilities removed from production images?

Service Sandboxing:

- Is the container runtime process itself sandboxed by the service manager?
- Are system paths protected from modification by the runtime?
- Are kernel modules, tunables, and control groups protected?

Policy Enforcement:

- Is there admission control that validates container configurations before execution?
- Does mount policy use a default-deny approach?
- Can policy enforcement operate without connectivity to a central server?

Kernel Isolation:

- Are seccomp profiles applied to workload containers?
- Are Linux capabilities dropped to the minimum required set?
- Is mandatory access control (AppArmor or SELinux) enabled and enforced?
- Are containers running as non-root users by default?

9. How nova8 Technologies Aligns with These Principles

nova8 Technologies applies publicly documented security and systems-design principles, including minimal host design, immutable operation, trust-domain separation, and layered workload isolation, in its edge platform. This paper describes the public architectural pattern and evaluation criteria, not product implementation details.

nova8OS is designed around the concept that the host operating system should be minimal, verifiable, and replaceable as a single unit. The platform applies container hardening principles consistent with NIST SP 800-190 guidance, OCI runtime standards, and industry best practices for container-specific operating systems.

The container security architecture of the platform reflects the defense-in-depth model described in this paper: an immutable host foundation, service manager sandboxing of runtime processes, policy enforcement at container creation time, and kernel-level isolation mechanisms operating as independent, complementary layers.

nova8 Technologies is aligning infrastructure and development practices with CMMC Level 2 expectations. The company holds U.S. Provisional Patent Applications 63/897,352, 63/897,609, 63/903,132, 63/903,161, 63/903,164, and 63/903,168 related to edge computing and container security innovations.

For more information, visit nova8.io or contact the team at contact@nova8.io.

10. References

1. NIST, "SP 800-190: Application Container Security Guide," September 2017, csrc.nist.gov/pubs/sp/800/190/final

2. Open Container Initiative, "OCI Runtime Specification," specs.opencontainers.org/runtime-spec/

3. containers/common, "OCI Hooks Specification," github.com/containers/common/blob/main/pkg/hooks/docs/oci-hooks.5.md

4. Red Hat, "Open Container Initiative hooks for admission control in Podman," January 2024, redhat.com/en/blog/open-container-initiative-hooks-admission-control-podman

5. Red Hat, "Mastering systemd: Securing and sandboxing applications and services," redhat.com/en/blog/mastering-systemd

6. Red Hat, "Using systemd features to secure services," redhat.com/en/blog/systemd-secure-services

7. freedesktop.org, "systemd.exec: Sandboxing directives," freedesktop.org/software/systemd/man/systemd.exec.html

8. Arch Linux Wiki, "systemd/Sandboxing," wiki.archlinux.org/title/Systemd/Sandboxing

9. Flatcar Container Linux, "Security by Design," flatcar-linux.org

10. Siderolabs, "What is Talos Linux?," docs.siderolabs.com/talos

11. AWS, "Bottlerocket: A Container-Optimized OS," aws.amazon.com/bottlerocket/

12. CIS, "Docker Benchmark," cisecurity.org/benchmark/docker

13. CVE-2019-5736, "runc container escape," unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/

14. CVE-2022-0492, "cgroups v1 container escape," unit42.paloaltonetworks.com/cve-2022-0492-cgroups/