PORTAINER.iO

THE CONTAINER PLATFORM THREAT MODEL

What an LLM-assisted attacker could try against a container platform, and what stops each attempt.

Oct 2026

Executive summary

This document assesses six container runtimes (Docker Standalone, Docker Swarm, Docker Desktop, Podman, Kubernetes and Portainer-D2K) against an attacker who uses an unrestricted large language model to discover and exploit vulnerabilities. For each runtime it states what an attacker holds after a container escape, which controls the runtime itself enforces, and which controls exist only in tools placed in front of it.

The full model is Appendix 1.

What has changed

Container escapes aren't theoretical. runc, the runtime under Docker and most Kubernetes distributions, has had serious escape vulnerabilities repeatedly since 2019, most recently three disclosed in November 2025. Still, escapes stayed rare in the wild, because finding the flaw, writing the exploit, and chaining an application bug into a runtime or kernel bug took scarce skill. That scarcity, more than the runtime itself, was the real defense.

AI removes most of that limit. An attacker with an unrestricted frontier model gives it an application's source code or dependency list and gets back a working exploit for a flaw in it, then directs the same model to chain that foothold into a runtime or kernel escape. The attacker needs to understand neither step. Most platforms' assumptions about how often escapes happen were calibrated against human attackers, and they no longer hold. This threat model therefore starts from the assumption that an escape will happen.

> **The question for every runtime:** once an attacker has escaped the container, what do they hold, and what stops them from taking more?

Findings

01 Docker Standalone

The daemon runs as root, and container root is host root. An escape yields the host: every other container, the local image cache (which the attacker can rewrite), the host's credentials, and every network the host reaches.

The engine ships no policy. Any user with access to the daemon can run any command, and a request that reaches the socket by any path (CLI, CI runner, compose file, an AI agent holding a developer's token) is executed. The engine has an authorization plugin interface that could refuse requests, but Docker provides no plugin for it, and the best-known third-party plugin, OPA's opa-docker-authz, is a demonstration project with no recent releases. Because the engine cannot refuse a request, a tool in front of it can only act on requests that pass through that tool. This is why Portainer's Docker security settings for bind mounts, privileged mode, host PID, device mappings, capabilities, sysctls and security-opt hide these options from non-administrators rather than block them.

Image signing follows the same pattern. Docker Content Trust was a client-side setting bypassed by any direct API call, daemon-side enforcement existed only in the Mirantis fork, and Docker announced its retirement in July 2025.

Docker ships two native controls on blast radius. Rootless mode runs the daemon and containers as a non-root user, at the cost of AppArmor, checkpoint/restore, SCTP port publishing and overlay networks. userns-remap keeps the root daemon and, when enabled, maps container root to a subordinate UID range, but any request can set --userns=host to opt its container out, because nothing in the engine refuses it.

02 Docker Swarm

The same engine with a manager quorum, mutual TLS between nodes and an encrypted secrets store. Rootless mode does not support overlay networks, so every Swarm is a rootful cluster with a root daemon on each node. That leaves userns-remap as the only control on blast radius, and any request that reaches a node's daemon directly can still opt a container out of it.

An escape on a worker adds the node's certificate and the secrets of the tasks scheduled there. An escape on a manager yields the cluster: every secret, the join tokens, the root CA key, and the ability to schedule work onto every node.

03 Docker Desktop

The engine runs in a Linux VM, so the workstation sits behind a VM boundary, but one with deliberate openings: Docker Desktop shares host directories, by default the user's home directory on macOS, into the VM so bind mounts work. Inside the VM the engine is rootful. An escape yields every container in the VM, the image cache, and whatever host directories are shared into it. CVE-2025-9074 (August 2025) showed the boundary can fail outright: containers could reach the engine API without a socket mount, even with ECI enabled.

Enhanced Container Isolation (ECI) places each container in its own user namespace, and the engine refuses requests to share the VM's namespaces, mount the Docker socket, or mount VM configuration. It is available only on Docker Business, where admins can enforce it through Settings Management. Docker has built a mitigation for the rootful engine, and offers it only in its top subscription tier.

04 Podman

No persistent daemon, rootless when run by an unprivileged user, and an SELinux label or AppArmor profile on every container where the host enforces one. In rootless mode an escape lands as the invoking user. Under a dedicated service account that yields little; on a developer workstation it yields the developer's home directory, SSH keys, cloud and registry credentials, and every other container they run. Where Podman runs rootful, container root is host root and an escape yields the host, as on Docker Standalone.

Podman has no request authorization layer in either mode. Privileged mode, host namespaces and bind mounts are executed on request, bounded only by the invoking user's own rights in rootless mode and reaching the host in rootful mode. The Docker-compatible API socket carries the same rights: the user's socket is user-equivalent, and the system-wide socket enabled as root is root-equivalent. Podman does verify image signatures natively through `policy.json`, but a rootless user can override the system policy with their own.

05 Kubernetes

Every API request passes through the API server, which authenticates, authorizes and admits it before anything runs; access to a node's container runtime socket bypasses all of this. Pod Security Admission can refuse privileged containers, host-path mounts and host namespaces at the API regardless of the submitting client, but it enforces nothing until a namespace or the cluster is configured to, and anyone who can label a namespace can lower its level. User namespaces map container root to an unprivileged node user, and the Kubernetes project has shown them neutralizing CVE-2024-21626 and CVE-2022-0492, but each pod must opt in with `hostUsers: false` unless admission policy requires it. NetworkPolicy bounds lateral movement where the CNI enforces it and policies have been written.

An escape yields the node: every pod on it, the secrets and service account tokens mounted into those pods, the kubelet's credentials, and often the cloud instance's metadata credentials. Kubernetes ships the controls to limit this and enforces them at the API server, the kubelet and the CNI, but almost none are on by default.

06 Portainer-D2K

Presents the Docker Engine API and translates each call into Kubernetes operations in one namespace. The API is served over mutual TLS and is unreachable without a client certificate. Host network, host PID and host-path bind mounts are refused by the translator. A privileged request becomes a pod spec that Pod Security Admission decides. Existing Docker CLI, compose files and Portainer workflows continue unchanged. Enforcement moves to the API server.

Conclusion

Every runtime assessed except Docker Standalone and Docker Swarm can be configured so that an application exploit followed by a runtime or kernel escape does not yield host root, without giving up features the platform depends on. Docker Standalone reaches that state only in rootless mode, which removes overlay networks, AppArmor and checkpoint/restore. Docker Swarm cannot reach it at all: it requires a rootful daemon, its one remaining control, userns-remap, can be opted out of by any request that reaches a node, and the engine's one refusal hook ships with no policy and no maintained plugin. Docker Desktop reaches it only on Docker Business, through ECI. Against an attacker for whom vulnerability discovery is automated, Docker Standalone and Swarm offer the fewest usable security primitives of the six runtimes assessed.

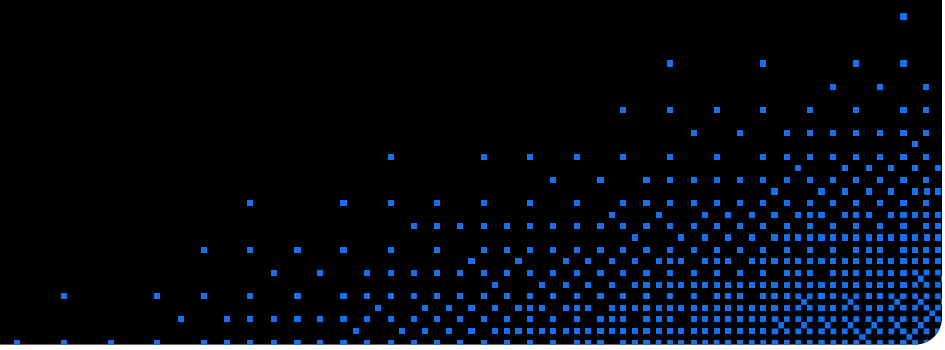
For Portainer, this sets direction. From Portainer 3.0 onward, new features are designed and built against Kubernetes. Docker Standalone, Swarm and Podman environments will continue to receive maintenance and security fixes but no new features. Portainer-D2K is the path for Docker environments that need to be defensible against the attacker described here.

> **Reading the appendix:** Appendix 1 contains the full model: the components of a container platform and their sensitivity, each threat with its STRIDE category, the defense each runtime offers, the residual risk, and a summary table. Each control is marked as Level 1 (shipped with the runtime) or Level 2 (a separately installed product).

Appendix 1

The container platform threat model

What an LLM-assisted attacker could try against a container platform, and what stops each attempt on Docker Standalone, Docker Swarm, Docker Desktop, Podman, Kubernetes and Portainer-D2K.



01. The model

Every container on a Linux node shares that node's kernel. One flaw that lets a process reach kernel or runtime code reaches everything else on the node, and the node itself. Finding such flaws has historically taken scarce skill, and that scarcity has acted as a rate limit on how often a container escape followed an application compromise.

The adversaries this model is designed against, in rough order of likelihood:

01 An attacker with an unrestricted LLM

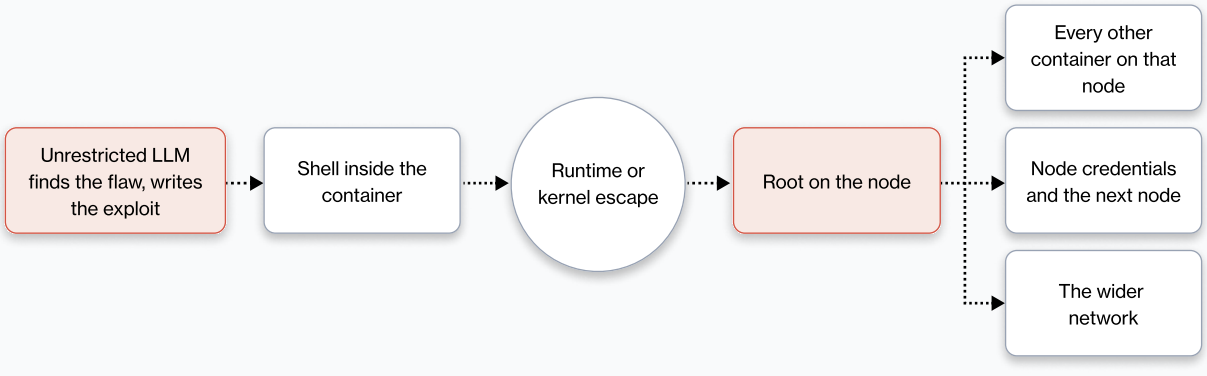
The most realistic attacker. The model reads the application's dependency list, finds the unpatched or undisclosed flaw, writes the exploit, and chains it to a runtime or kernel escape without the attacker needing to understand either. Assume the application will be exploited and design for it. Patch cadence protects you only against flaws that have been disclosed.

02 A poisoned supply chain

A base image, a dependency, or a build step that gives the attacker execution without needing any vulnerability.

03 A legitimate user overreaching

Someone with access to the management socket or API who starts a privileged container, mounts the host filesystem, or shares the host network, because nothing stopped them. With AI agents now driving these APIs on a person's behalf, this adversary includes a prompt-injected agent acting with that person's access.



Every runtime on this page allows the first two steps. The paper is about which of them stop the third, and what the attacker holds if they do not.

The question this model asks of each runtime is the same every time: once the attacker has code execution inside a container, what do they hold, and what stops them from holding more? The answers divide into two levels:

Level 1 is a setting or policy the runtime ships. **Level 2** is a separately installed product.

02. Decomposing the platform

Each entry lists a component, where traffic enters and leaves it, and what it holds that an attacker would want. Sensitivity means what is lost if the component is compromised.

Application container Process

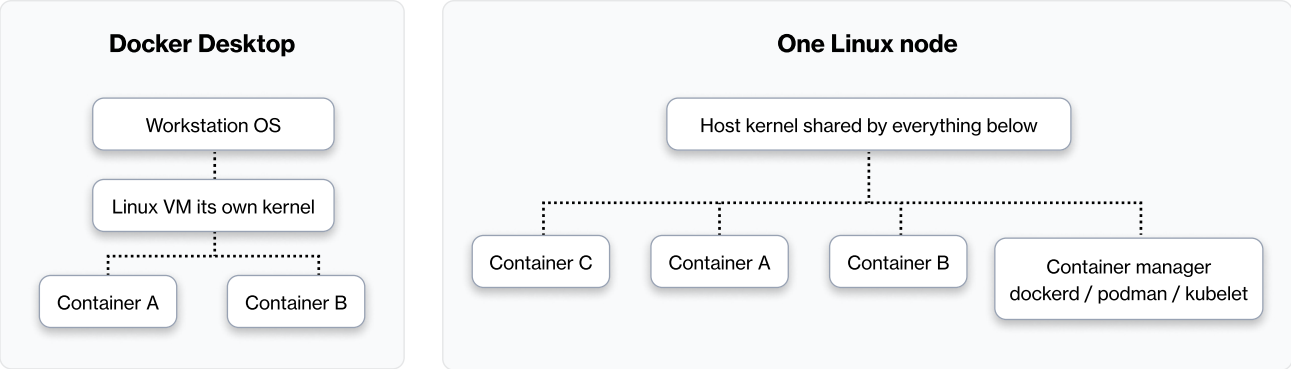
- **Entry points:** whatever network ports it publishes, and whatever content it reads.
- **Exit points:** syscalls to the shared kernel, the container network, any mounted volume.
- **Sensitivity:** the first foothold, plus every credential in its environment, image layers and mounted secrets.

OCI runtime (runc, crun) Process

- **Entry points:** the container manager, when it creates, starts or execs a container.
- **Exit points:** the kernel namespaces, cgroups, mounts and LSM labels it sets up.
- **Sensitivity:** it runs as root on the host during container setup, so a flaw in it is a host escape. The November 2025 runc disclosures covered three container escapes: masked path abuse through mount races, /dev/console mount races, and procfs write redirects (Red Hat advisory RHSA-2025:20957).

Host kernel Data + Process

- **Entry points:** every syscall from every container.
- **Exit points:** none. It is the thing being protected.
- **Sensitivity:** shared by all containers on the node, and by the node itself.



One kernel flaw reaches every container on the node and the manager beside them. Docker Desktop moves that shared kernel into a VM, which is why the workstation sits behind a boundary the other runtimes do not have.

Container manager (dockerd, podman, kubelet, the D2K translator) Process

- **Entry points:** a Unix socket, a TCP endpoint, or the Kubernetes API.
- **Exit points:** the OCI runtime, the registry, the network.
- **Sensitivity:** whoever controls the manager controls every container it runs. On a root daemon that is host root.

Management API Data

- **Entry points:** the socket or endpoint above, plus any tool that fronts it (Portainer included).
- **Sensitivity:** the authorization chokepoint, where one exists. Where none exists, the tool in front of it is the only gate, and the API behind it accepts whatever reaches it.

Image registry Data Store

- **Entry points:** pushes from CI.
- **Exit points:** pulls at container start.
- **Sensitivity:** the supply chain. Writing to it is deploying, eventually.

Container network Data

- **Entry points:** every container on it.
- **Sensitivity:** flat by default on every runtime here, which makes it the lateral movement path.

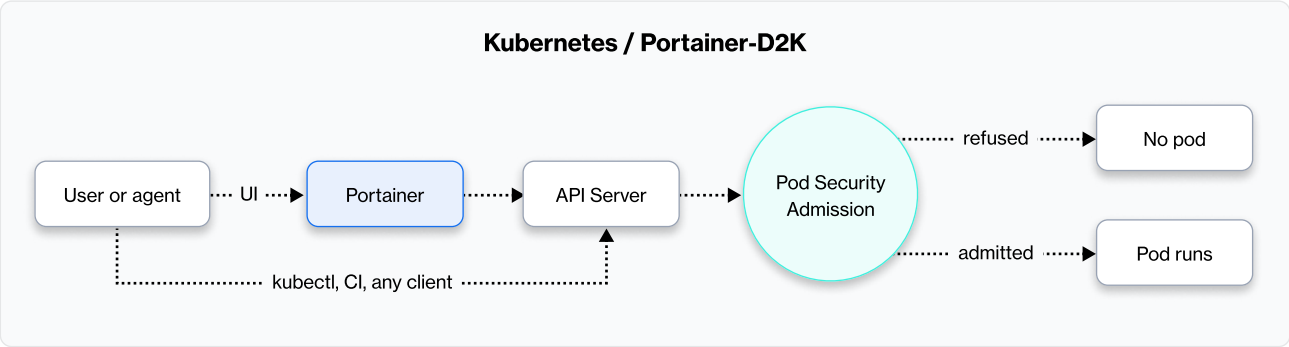
Host filesystem Data

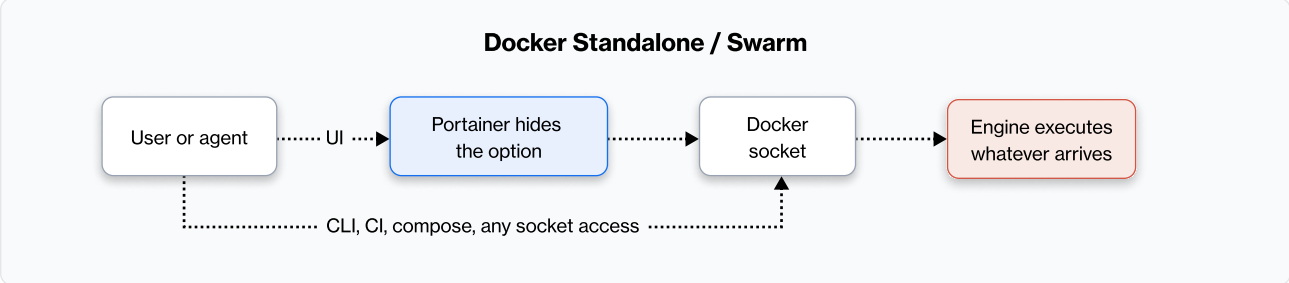
- **Entry points:** bind mounts, hostPath volumes, the management socket itself.
- **Sensitivity:** the node's credentials, its service units, and the image cache.

---

03. Runtime profiles

Each runtime is profiled by its default posture. Every control named below was confirmed current in October 2026. The chart below is the distinction this section keeps returning to: whether the thing in front of the runtime can refuse a request, or only hide the button.



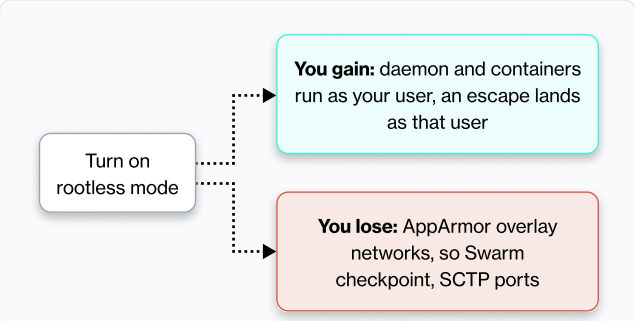


On the left, the front end can only hide a request, because the engine behind it has no gate. On the right, the API server decides, so the client does not matter.

Docker Standalone (Engine 29.x)

- Rootful by default. The daemon requires root privileges unless you opt in to rootless mode, and any user with permission to access the daemon can run any Docker client command (Docker Engine security documentation).
- The engine has an authorization plugin interface that could refuse a request. Docker ships no policy for it and no maintained plugin. The known implementations are archived or commercial, so an operator who wants enforcement writes and runs their own.
- Rootless mode is one of two native controls on blast radius. You give up AppArmor, checkpoint, overlay networks and SCTP ports, and capabilities added with --cap-add apply only inside the container's user namespace.
- userns-remap is the other. The daemon stays root and container root maps to a subordinate UID range by default. Any request can set --userns=host to opt its container out, and the engine accepts it.
- Portainer's Docker security settings (bind mounts, privileged mode, host PID, device mappings, capabilities, sysctls, security-opt) are labeled "hide for non-administrators", because the engine offers no way to refuse these requests, and a user with any other path to the socket can make them regardless of what the UI shows. We used to describe those settings as blocking. We had to rename them to hiding, because that is effectively what they do.

Rootless mode costs the features that make Docker useful as a server runtime. userns-remap keeps them and can be opted out of per container.



Docker Swarm (Engine 29.x)

- The same engine with a Raft-based manager quorum, mutual TLS between nodes, join tokens, and an encrypted secrets store.
- Rootless mode does not support overlay networks, so a hardened Swarm is a rootful Swarm, and every node in it runs a root daemon.
- Whatever the service API accepts is scheduled. There is no admission layer, and the hide-versus-block limitation above applies to Swarm services in the same way.

Docker Desktop (4.x)

- The engine runs inside a Linux VM on the workstation, so the kernel the containers share is the VM's, and the workstation's operating system sits behind the VM boundary.
- Inside the VM the engine is rootful, and containers run as true root in the VM unless Enhanced Container Isolation (ECI) is enabled.
- Docker implements ECI using the Sysbox container runtime, a security-enhanced fork of runc. With ECI on, all user containers run in a Linux user namespace, so root in the container maps to an unprivileged user in the Docker Desktop VM. Each container gets its own exclusive mapping, and the Docker CLI's --runtime flag is ignored, so it cannot be bypassed per container.
- ECI is available to Docker Business customers only. An administrator enables and locks it through Settings Management so developers cannot switch it off. Every later reference to ECI in this document means this paid, administrator-enforced feature. A Docker Desktop installation on a Personal, Pro or Team subscription does not have it.

Podman (6.x)

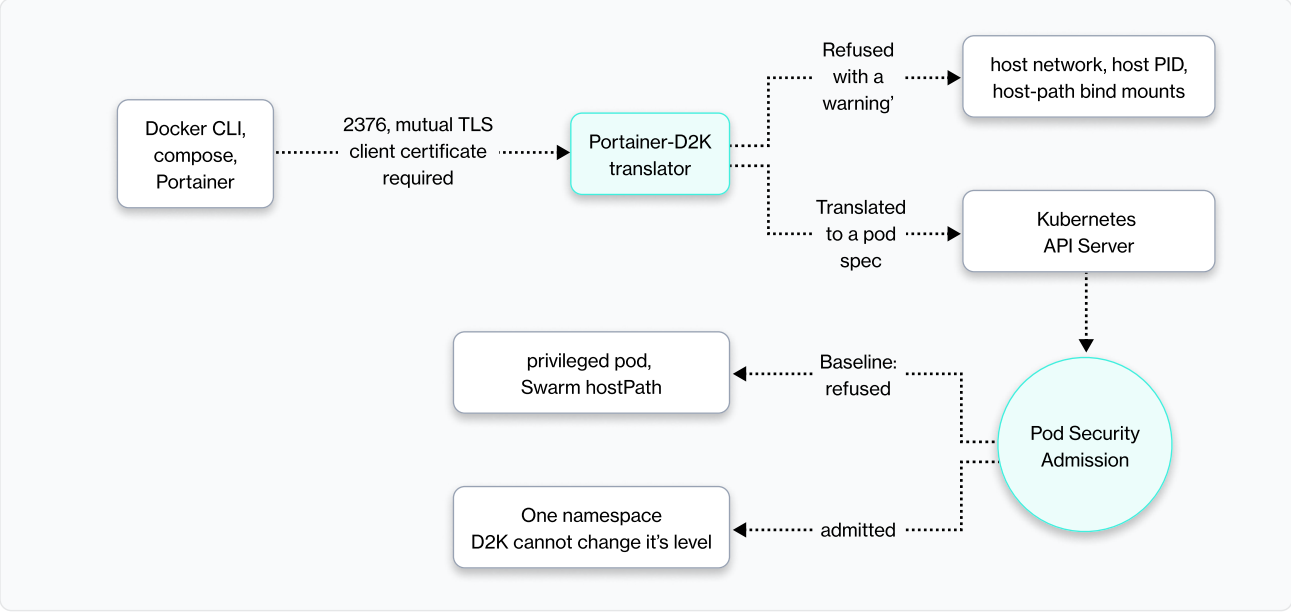
- Daemonless and rootless by default. Every container launches with an SELinux label on RHEL-family hosts or an AppArmor profile on Ubuntu and Debian, where the host enforces one.
- Where Podman is run rootful, container root is host root and an escape yields the host as on Docker Standalone. A system-wide Docker-compatible API socket enabled as root is root-equivalent.
- Policy is per host, in containers.conf. There is no central enforcement point.
- When Podman is managed through its Docker-compatible API, as Portainer does, the same hide-versus-block limitation applies, because that API carries the same requests and Podman accepts them in rootful mode.

Kubernetes (1.36)

- Rootful containers by default, with native controls for every layer except the kernel.
- User namespaces are stable and enabled by default in 1.36, opted in per pod with hostUsers: false.
- Pod Security Admission is built in. NetworkPolicy is an API that an enforcing CNI implements. RuntimeClass selects an alternative runtime per pod.
- Policy here is enforced by the API server, so a management tool in front of it configures enforcement and does not have to impersonate it.

Portainer-D2K

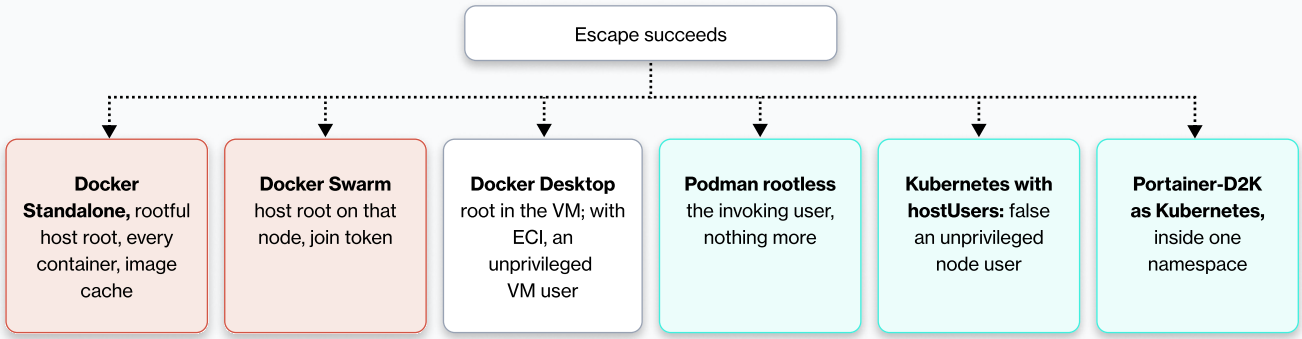
- Exposes the Docker Engine API and translates every call into Kubernetes operations scoped to one namespace, running under a ServiceAccount bound to a namespace-scoped Role.
- Serves the API on 2376 over mutual TLS. Every client presents a certificate issued by the D2K CA, so the certificate is the authentication. d2k pki init issues the CA, server and client certificates, with cert-manager as an alternative issuer. Plaintext 2375 exists only when an operator sets D2K_INSECURE=true.
- In Docker host mode, privileged mode and added capabilities are translated into the pod's securityContext, where Pod Security Admission decides them. Host-path bind mounts are refused with a warning. Host, none and container network modes are refused with a warning. Every other HostConfig field D2K cannot honor is reported back as a warning rather than dropped.
- In Swarm mode, compose bind entries become hostPath volumes.
- D2K sets no Pod Security label on its namespace, and a ValidatingAdmissionPolicy shipped with it (Kubernetes 1.30 or later) prevents its ServiceAccount from changing one. The level the cluster administrator or Portainer sets is the level that applies.
- D2K's own pod runs as a non-root user with a read-only root filesystem, the RuntimeDefault seccomp profile, and all capabilities dropped.



There is no daemon to reach. The translator refuses what Kubernetes cannot express, and the API server decides what it can.

04. Summary by runtime

The same runc flaw yields six different outcomes. Red is the default on that runtime. Green is what the runtime ships once the setting is on.



Threat	Docker Standalone	Docker Swarm	Docker Desktop (ECI = Docker Business)	Podman	Kubernetes	Portainer-D2K
Container root is host root	L1 rootless, or userns-remap with per-container opt-out	L1 userns-remap with per-container opt-out	VM; L1 ECI	Rootless: closed; rootful: host root	L1 hostUsers: false	L1 via cluster
Runtime escape	L1 rootless or userns-remap	L1 userns-remap	L1 ECI	Rootless: invoking user; rootful: host root	L1 user namespaces	L1 via cluster
Kernel zero-day	Seccomp and user namespaces only[1]	Seccomp and userns-remap only[1]	VM boundary	Seccomp and rootless only[1]	Seccomp and user namespaces only[1]	Via cluster[1]
Unsigned or untrusted image	None in engine; UI can restrict	None in engine; UI can restrict	Registry and Image Access Management (Docker Business)	L1 policy.json per host	L2 admission with signature verification	L2 via cluster
Privileged, host namespaces, host mounts	Accepted; authz plugin interface with no shipped policy; UI can hide	Same as Standalone	L1 ECI	Rootless limits effect	L1 Pod Security	Host namespaces refused; privileged and Swarm hostPath decided by Pod Security
Policy enforced in the runtime	None shipped	None shipped	ECI only	Per host, no API gate	API server admission	API server admission; D2K cannot alter the namespace's level
Manager compromise	Root daemon; L1 rootless	Root daemon on every node	Inside VM	No daemon; root API socket is root-equivalent if enabled	API server with RBAC	Mutual TLS required; namespace-scoped
Lateral movement	Per-host manual	Overlay segmentation, no policy	Workstation	Per-host manual	L1 NetworkPolicy with CNI	L1 NetworkPolicy; D2K itself is flat

The sections that follow are the supporting detail for each row.

05. Threats and mitigationsv

Each threat is tagged with its STRIDE categories: Spoofing, Tampering, Repudiation, Info disclosure, Denial of service, Privilege elevation.

Privilege elevation

An application zero-day gives execution inside a container

Attack

The LLM finds a flaw in the application or one of its dependencies and lands a shell inside the container.

Defense

All six runtimes allow this, and the model assumes it has happened. What the runtimes decide is what the shell can reach, which the rest of this page covers. The compensating controls sit above the runtime: minimal base images, image scanning, and a read-only root filesystem with no shell binary. Every runtime here can express all three.

Tampering

A poisoned image or build step

Attack

A dependency, a base image, or a CI step carries the payload in, with no vulnerability required.

Defense

This sits above all six runtimes. The runtimes differ in whether anything at the API refuses an unsigned or unpinned image.

- **Docker Standalone and Swarm:** nothing in the engine. Docker Content Trust was a client-side check controlled by an environment variable, so a direct API call bypassed it. The daemon-side enforcement that refuses unsigned images exists only in Mirantis Container Runtime, the commercial fork of the engine. Docker is retiring DCT and the Notary v1 service entirely, having announced in July 2025 that the upstream Notary codebase is unmaintained and that fewer than 0.05% of Docker Hub pulls used it. Portainer can restrict which registries a Docker user may pull from and which images they may deploy. That restriction is applied in Portainer and bypassed by any direct path to the socket.
- **Docker Desktop:** Registry Access Management and Image Access Management on Docker Business restrict pulls at the engine inside the VM, which is enforcement at the daemon for that one product.
- **Podman:** policy.json gives a per-host signature policy that can require sigstore or GPG signatures per registry, enforced by Podman itself at pull time. Native, maintained, and per host.
- **Kubernetes:** admission policy (Kyverno or Gatekeeper, with Notation or cosign verification) refuses the pod at the API server whatever the client submitted. Level 2, current, and common.
- **D2K:** the same admission path. D2K ignores the registry credentials a client sends and pulls with the namespace's pull secrets, so the namespace, and any policy on it, decides which registries are reachable.

Info disclosure

Credentials reachable from inside the container

Attack

The shell reads environment variables, image layers, a mounted service account token, or the cloud metadata endpoint.

Defense

- **Docker Standalone and Swarm:** secrets are injected as files under /run/secrets, which keeps them out of environment variables and leaves them within a shell's reach.
- **Kubernetes:** a service account token is mounted by default unless automountServiceAccountToken is false, and that token is the usual pivot into the cluster.
- **D2K:** translates Swarm secrets to Kubernetes Secrets and mounts them the same way.
- **All six:** the metadata endpoint is a network control.

Spoofing Privilege elevation

An exposed management API

Attack

The Docker socket mounted into a container, a daemon on TCP without TLS, a reachable kubelet, or a D2K instance started with D2K_INSECURE=true.

Defense

- **Docker Standalone and Swarm:** the socket is root-equivalent on a rootful daemon. The native control is rootless, which narrows socket access to that user's privileges. The engine offers 2376 with TLS and 2375 without, and refuses to start when the API is exposed over HTTP without TLS by accident. A 2375 listener configured deliberately is an unauthenticated root API on the network.
- **Docker Desktop:** the socket is inside the VM.
- **Podman:** no daemon, and no socket unless enabled. A system-wide socket enabled as root is root-equivalent.
- **Kubernetes:** the API server and kubelet authenticate every request, with RBAC behind them.
- **D2K:** mutual TLS on 2376 is required by default and the client certificate is the credential. A pod that can reach the Service and holds no certificate gets a TLS handshake failure. Plaintext 2375 requires D2K_INSECURE=true and reintroduces the same exposure as Docker's own 2375, scoped to one namespace. NetworkPolicy on the Service remains worthwhile as a second layer.

Residual risk. A mounted Docker socket defeats every other control on Docker Standalone and Swarm. It is the most common self-inflicted escape, and only an authorization plugin or review discipline prevents a user from mounting it.

Privilege elevation | Tampering

Relying on a management tool to enforce what the runtime does not

Attack

The organization has configured Portainer, or any other front end, to prevent bind mounts, privileged mode, host PID, device mappings, added capabilities, sysctls and security-opt overrides. The attacker, or the attacker's prompt-injected agent, reaches the Docker API by another route (a CLI with socket access, a CI runner, a compose file deployed outside the UI, or a flaw in the front end itself) and makes the request directly.

Defense

- **Docker Standalone and Swarm:** the engine ships no policy, and the requests are legal API calls the daemon executes. A front end can only remove them from its own screens, which is why Portainer labels these settings "hide for non-administrators". The engine's authorization plugin interface is the one place a refusal could live, and Docker ships nothing for it, so the operator builds and runs that plugin themselves. A growing share of CVEs raised against Portainer describe an engine capability reachable through the UI, where the only remedy available is to remove the capability from the UI. Each such removal costs the people who need the feature and does nothing for anyone who bypasses the UI.
- **Docker Desktop:** with ECI, the engine refuses host namespace sharing and VM mounts regardless of client, and privileged mode is confined to the user namespace. This is enforced in the runtime, so a front end does not have to.
- **Podman:** rootful Podman behaves as Docker. Rootless Podman accepts the request and the kernel denies the host effect.
- **Kubernetes:** Pod Security Admission refuses the pod at the API server, whichever client submitted it. The API behind the front end cannot be talked past.
- **D2K:** the same Docker API request arrives over mutual TLS, host namespaces and host-path binds are refused in the translator, and privileged mode becomes a pod spec the API server decides. The front end has nothing to hide because the translator and the cluster refuse.

Residual risk. Any front end that holds an administrator credential to the Docker socket is itself the escalation path. A flaw in it is a flaw in the daemon's access control, because the daemon ships none of its own.

Privilege elevation

A kernel zero-day reachable through syscalls

Attack

The LLM finds a flaw in a kernel subsystem the container can reach and uses it to escape.

Defense

- **Docker Standalone, Swarm and Podman:** the default seccomp profile, which filters the syscalls a container can make, and user namespaces (rootless or userns-remap), which change which kernel paths an unprivileged container can reach.
- **Kubernetes and D2K:** the same two, with seccomp set to RuntimeDefault per pod and user namespaces via hostUsers: false.
- **Docker Desktop:** the attacker reaches the VM kernel, and the VM boundary protects the workstation.

Residual risk. A reduced surface is still a shared kernel. Sandbox runtimes that interpose a separate kernel or a user-space kernel between the container and the host exist (see the footnote). They are the only control that changes this boundary, and the one on this page that the field has largely declined to adopt.

Privilege elevation

A runtime escape of the runc class

Attack

A race or confused-deputy flaw in the OCI runtime at container start, of the kind disclosed three times in November 2025. One of those three can disable LSM labels, which defeats AppArmor, the main mitigation available for another of the three (runc advisory GHSA-cgrx-mc8f-2prm).

Defense

The flaw is in software shared by every runtime here, so patching is the first control everywhere. What differs is what the escape yields.

- **Docker Standalone:** Level 1, rootless or userns-remap. A container started with --userns=host is outside the remap.
- **Docker Swarm:** userns-remap only, since rootless and overlay networks are incompatible. The same opt-out applies.
- **Docker Desktop:** Level 1, ECI, on Docker Business only. Without it the escaped process is root in the VM, and the workstation is protected by the VM boundary alone.
- **Podman:** rootless, so the escaped process is the invoking user. Rootful yields host root.
- **Kubernetes:** Level 1, user namespaces. Demonstrations show CVE-2024-21626 and CVE-2022-0492 mitigated when they are active (Kubernetes 1.33 user namespaces blog).
- **D2K:** the same, if the cluster sets hostUsers: false.

Privilege elevation

A privileged container, or excess capabilities

Attack

A user, or an agent, starts a container with --privileged or adds a capability that reaches the host.

Defense

- **Docker Standalone and Swarm:** accepted by the daemon. Rootless or userns-remap limits what it buys, and under userns-remap a privileged container needs --userns=host, which the engine also accepts. A front end can hide the option.
- **Docker Desktop:** with ECI, privileged containers work and are privileged only within the container's user namespace, and Docker-in-Docker runs unprivileged inside the VM. Without ECI a privileged container is privileged in the VM.
- **Podman:** rootful Podman accepts it. Rootless limits what it buys.
- **Kubernetes:** Level 1, Pod Security Admission at Baseline forbids privileged containers, enforced by the API server when the namespace is labeled.
- **D2K:** the flag is translated into the pod spec and Pod Security Admission decides it. On a namespace at Baseline the pod is refused. On a cluster whose default is privileged it is admitted.

Privilege elevation Info disclosure

Sharing host namespaces or the host filesystem

Attack

--network=host, --pid=host, a bind mount of /, or a hostPath volume.

Defense

- **Docker Standalone and Swarm:** accepted by the daemon. A front end can hide the option.
- **Docker Desktop:** under ECI, containers cannot share namespaces with the VM and mounts of VM configuration directories are blocked. Without ECI that mount works and gives the container full read and write access to the engine's configuration. Folders shared from the workstation remain reachable either way, by design.
- **Podman:** accepted, with SELinux or AppArmor constraining what the mount can touch on an enforcing host.
- **Kubernetes:** Level 1, hostPath volumes and host namespaces are forbidden under Baseline.
- **D2K:** host, none and container network modes are refused. Host-path bind mounts are refused in Docker host mode. In Swarm mode compose bind entries become hostPath volumes, which Pod Security at Baseline refuses.

Privilege elevation

Container root becomes host root

Attack

Any escape above, on a runtime where UID 0 in the container is UID 0 on the node.

Defense

This row decides blast radius.

- **Docker Standalone:** Level 1, rootless, or userns-remap with --userns=host as the per-container opt-out.
- **Docker Swarm:** Level 1, userns-remap, with the same opt-out.
- **Docker Desktop:** contained to the VM, and ECI closes it inside the VM.
- **Podman:** closed when rootless. Host root when rootful.
- **Kubernetes:** Level 1, hostUsers: false per pod.
- **D2K: the same.**

Privilege elevation

Compromising the manager itself

Attack

The attacker goes after dockerd, a Swarm manager, the API server, or the D2K translator.

Defense

- **Docker Standalone:** the daemon is the most valuable process on the host, and rootless is the only native control that lowers its privilege.
- **Docker Swarm:** a manager holds the Raft store with every secret and the CA that issues node certificates. Autolock encrypts the store at rest with a key you hold, and manager-to-node traffic is mutual TLS. A compromised manager is a compromised cluster.
- **Docker Desktop:** the daemon is inside the VM.
- **Podman:** there is no daemon. Each container is a child of the invoking process.
- **Kubernetes:** the API server authenticates and authorizes every request, with admission and audit behind it. It is the hardest target here and the one that matters most.
- **D2K:** reachable only with a client certificate, running as a non-root process with a read-only root filesystem and no capabilities, under a ServiceAccount with a namespace-scoped Role. Compromising it yields one namespace.

Privilege elevation | Info disclosure

Node credential theft and lateral movement

Attack

Having escaped, the attacker takes kubelet credentials, join tokens, registry credentials, or other workloads' secrets, and moves to the next node.

Defense

- **Docker Standalone:** each host is an island, which limits what can be stolen and leaves nothing to stop the move.
- **Docker Swarm:** a worker holds its node certificate and the join token it used, and the join token lets a rogue node enroll. Rotate it after any suspected escape.
- **Docker Desktop:** one workstation.
- **Podman:** per host, as Docker Standalone.
- **Kubernetes:** a node holds kubelet credentials, and the NodeRestriction admission plugin limits what a kubelet can modify to its own node and pods. Native and on by default in current distributions.
- **D2K:** the same as Kubernetes, with D2K's own account limited to one namespace.

Info disclosure | Tampering

A flat network

Attack

From a compromised container, the attacker reaches every other container and, often, the wider corporate network.

Defense

- **Docker Standalone:** Level 1, user-defined networks and host firewall rules, configured per host.
- **Docker Swarm:** overlay networks give per-stack segmentation and can be encrypted. There is no policy object, so a service on two networks bridges them.
- **Docker Desktop:** workstation scope.
- **Podman:** per host, as Docker.
- **Kubernetes:** Level 1, NetworkPolicy, enforced by a CNI that implements it (Cilium and Calico, both current and common).
- **D2K:** namespace networking is flat, docker network create records a synthetic network, and no isolation is enforced. Cluster NetworkPolicy is the only control.

Tampering Repudiation

Persistence and covering tracks

Attack

A cron entry or systemd unit on the node, a tampered image in the local cache, or deleted logs.

Defense

- **All six:** any runtime that prevents host root prevents host persistence (see the root row). Log tampering is controlled outside the runtime. Ship logs off the node.
- **Docker Standalone and Swarm:** the daemon keeps a local image store that a root attacker can rewrite.
- **Kubernetes and D2K:** images are pulled at schedule time, and D2K stores no images at all.

Denial of service

Speed as an amplifier

Attack

The LLM chains foothold, escape and lateral movement faster than a human responder reads the first alert.

Defense

No runtime rate-limits an attacker. The compensating control is a short path from detection to isolation: halting a node in Swarm or cordoning and draining in Kubernetes, with the halt scripted in advance.

06. What this model concludes

Every runtime here can reach a state where an application exploit plus a runtime or kernel escape yields neither host root nor lateral movement, and that is the bar this model sets. The runtimes differ in what reaching it costs, and in whether the runtime itself enforces the result.

Kubernetes

closes every row except the kernel with settings it ships, and enforces them at the API server, so the client does not matter.

Podman

arrives rootless and daemonless, so most rows are closed on install, with policy held per host. Run rootful, it matches Docker Standalone on blast radius.

Docker Desktop

shields the workstation by construction, and hardens the VM only on Docker Business, where ECI is the control. On any other subscription the VM is a rootful Docker Standalone host with the same exposures as that column.

Docker Swarm

is the harder case again. Rootless and overlay networks are mutually exclusive, so a Swarm is always a set of root daemons, userns-remap with its opt-out is the sole native control on blast radius, and no admission layer exists.

Portainer-D2K

inherits all of that, and adds what the Docker engine lacks at the API itself: mutual TLS is required to reach it, host namespaces are refused in the translator, and a privileged request becomes a pod spec the API server decides. The one setting the operator still owns is the namespace's Pod Security level, which D2K neither sets nor can change. Leave the cluster default at privileged and a privileged request is admitted. Set Baseline and it is refused.

Docker Standalone

ships no policy. Its two native controls on blast radius are a mode that removes server features and a default that any request can opt out of. Its authorization plugin interface could refuse a request, and Docker provides nothing for it. Every policy an organization believes it has applied through a management tool is a view setting the API behind it does not share, which is why Portainer's settings became hides. Image signing, the one engine-side check Docker once offered, was always a client setting on CE and is now being retired.

> The defensible claim is therefore that Docker Standalone and Docker Swarm are the only runtimes that cannot be sufficiently secured with native settings alone without giving something up, and that the only hook for enforcement in the engine is one the operator must build and run themselves. For workloads that AI agents deploy, or that AI-assisted attackers target, that combination leaves the Docker engine with the fewest usable primitives of the six.

07. Limits of this model

This model is built from vendor and project documentation, from the D2K source, and from Portainer's own experience fronting the Docker API. None of the configurations in it were tested for this document. Several entries describe an available control, and on Kubernetes most protections are opt-in per namespace or per pod.

08. Reporting

If you believe you have found an error in this model, or a control it omits, report it to your Portainer contact so the model can be corrected.

[1] Sandbox runtimes exist that change the kernel boundary itself: gVisor (a user-space kernel, release 20261005.0, accepted into the CNCF in September 2026) and Kata Containers (a lightweight VM per pod, 4.0.0, August 2026). Both are current, both integrate with Kubernetes through RuntimeClass, and gVisor can be registered as an alternate runtime on Docker Standalone and Podman. The ecosystem has largely left them on the shelf. Outside managed offerings such as GKE Sandbox and Cloud Run, and some multi-tenant or confidential-computing platforms, they are rarely deployed, Kata's own documentation does not support Podman, and neither is a per-service setting on Swarm. They also narrow what a workload can do. gVisor reaches NVIDIA GPUs through a vetted driver subset (nvproxy) and Kata by passing a whole device into its VM, so GPU-heavy or hardware-adjacent applications tend to be the first reason a team switches them off. They are also attacked in turn: a use-after-free in gVisor before 20260831.0 allowed code execution in the sentry process (CVE-2026-95702). This model treats them as available and unadopted, and does not count them toward any runtime's rating.

PORTAINER.iO