Docker Security & Production Hardening

Securing Containerized Applications in Real-World Environments

Preface

When Docker revolutionized software delivery over a decade ago, it promised speed, portability, and consistency. It delivered on all three. What it did not deliver—what no technology ever delivers out of the box—was security.

Today, Docker containers power everything from startup MVPs to Fortune 500 production systems. They run in CI/CD pipelines, on bare-metal servers, across cloud platforms, and at the edge. Yet for all their ubiquity, Docker environments remain dangerously under-secured. Containers are shipped with root privileges that aren't needed, images are pulled from public registries without verification, secrets are baked into layers for anyone to extract, and runtime configurations are left at their permissive defaults. The speed that makes Docker transformative is the same speed that lets insecure practices scale.

This book exists to close that gap.

Docker Security & Production Hardening is a comprehensive, practical guide to securing containerized applications in real-world environments. It is written for DevOps engineers, platform teams, security professionals, and software developers who use Docker in production and recognize that convenience without hardening is a liability. Whether you are running a handful of containers on a single host or managing Docker across a sprawling infrastructure, this book provides the knowledge and tools to defend every layer of your Docker environment.

What This Book Covers

The journey begins where security must always begin: with **understanding the threat landscape**. Chapters 1 and 2 establish why Docker container security is fundamentally different from traditional server security and walk you through threat modeling techniques specific to Docker environments.

From there, we move into the **build phase**, where Chapters 3 and 4 guide you through crafting minimal, hardened Docker images and implementing vulnerability scanning into your workflow—because security that starts at deployment starts too late.

The heart of the book addresses **runtime and operational security**. Chapters 5 through 10 cover container runtime protections, resource isolation, Docker network hardening, securing external access points, secrets management, and configuration best practices. These chapters form the defensive core of any production Docker deployment.

No security posture is complete without **visibility and response**. Chapters 11 and 12 tackle logging, observability, and incident response within Docker environments, ensuring that when something goes wrong—and eventually, something will—you can detect it, understand it, and contain it.

Finally, we broaden the lens. Chapters 13 through 16 address Docker host hardening, securing CI/CD pipelines for containerized applications, common Docker security anti-patterns that even experienced teams fall into, and the path from Docker-specific hardening to broader cloud-native security practices.

The appendices provide **immediately actionable resources**: a Docker security checklist, a secure Dockerfile template, runtime configuration examples, an incident response playbook, and a learning roadmap for continued growth.

How to Use This Book

You can read this book cover to cover for a complete security education, or use it as a reference, jumping to the chapters most relevant to your current Docker challenges. Every chapter is designed to be both conceptually grounded and operationally practical, with real configurations, real commands, and real-world context.

Acknowledgments

This book was shaped by the collective wisdom of the Docker and container security community—the engineers who file CVEs, the maintainers who patch them, the practitioners who share hard-won lessons in blog posts and conference talks, and the open-source contributors who build the tools that make container security possible. I am deeply grateful to the technical reviewers whose sharp eyes and honest feedback made every chapter stronger, and to the teams I've worked alongside in production environments where these practices were forged under pressure.

Most of all, this book is for every engineer who has stared at a running Docker container and wondered, *"Is this actually secure?"*

Let's make sure the answer is yes.

Dorian Thorne

Table of Contents

Chapter 1: Why Container Security Is Different

The first time a security breach occurs in a containerized environment, the reaction from most engineering teams is remarkably similar. There is confusion, followed by a scramble to understand what happened, and then the slow, uncomfortable realization that the assumptions they carried over from traditional server security simply did not apply. Containers, and Docker in particular, introduced a paradigm shift in how applications are packaged, deployed, and executed. With that shift came a fundamentally different security landscape, one that demands its own strategies, tools, and mental models.

This chapter lays the foundation for everything that follows in this book. Before we harden Docker images, lock down runtime configurations, or implement production-grade monitoring, we need to understand why container security is its own discipline. We need to appreciate the architectural differences between containers and virtual machines, grasp the unique attack surface that Docker introduces, recognize the shared responsibility between developers and operations teams, and internalize the real-world consequences of getting container security wrong.

Let us begin at the beginning.

The Architectural Reality of Docker Containers

To understand why container security is different, you must first understand what a container actually is at the operating system level. There is a persistent misconception that containers are lightweight virtual machines. This comparison, while useful for initial understanding, is dangerously misleading when it comes to security.

A virtual machine runs a complete guest operating system on top of a hypervisor. Each virtual machine has its own kernel, its own memory space, and its own system processes. The hypervisor mediates all access to the underlying hardware. If an attacker compromises a virtual machine, they are still separated from the host and other virtual machines by the hypervisor boundary, which represents a strong isolation layer.

A Docker container, by contrast, shares the host operating system's kernel. Containers achieve isolation through Linux kernel features, primarily namespaces and cgroups. Namespaces provide the illusion of separation by giving each container its own view of process IDs, network interfaces, mount points, user IDs, and hostnames. Cgroups limit and account for resource usage such as CPU, memory, and disk I/O. Together, these mechanisms create what feels like an isolated environment, but the fundamental truth remains: every container on a host is making system calls to the same kernel.

This distinction has profound security implications. Consider the following comparison:

Characteristic	Virtual Machine	Docker Container
Kernel	Each VM has its own kernel	All containers share the host kernel

Isolation mechanism	Hardware-level hypervisor	Kernel namespaces and cgroups
Attack surface to host	Must escape hypervisor	Must escape namespace/cgroup boundaries
Boot time	Minutes	Seconds or less
Resource overhead	High (full OS per VM)	Low (shared kernel, minimal overhead)
Kernel vulnerability impact	Contained to individual VM	Potentially affects all containers and host
System call access	Filtered through hypervisor	Direct to shared kernel

When a kernel vulnerability is discovered, every container running on that host is potentially affected. There is no hypervisor standing between the container and the kernel. A container escape exploit, where an attacker breaks out of the namespace isolation and gains access to the host, is not a theoretical concern. It has happened in the real world, and it will happen again. The CVE-2019-5736 vulnerability in runc, the container runtime used by Docker, allowed a malicious container to overwrite the host runc binary and gain root-level access to the host system. This single vulnerability demonstrated that the boundary between a container and its host is thinner than many organizations assumed.

You can inspect the namespace isolation of a running Docker container to see this shared kernel reality for yourself:

```
# Start a simple container
docker run -d --name test-container alpine sleep 3600

# Check the kernel version inside the container
docker exec test-container uname -r

# Check the kernel version on the host
uname -r
```

Both commands will return the same kernel version. The container is not running its own kernel. It is using the host's kernel directly. This is not a flaw; it is the fundamental design of container technology. But it means that security strategies designed for virtual machines, where the kernel boundary provides strong isolation, must be rethought entirely for containers.

The Expanded Attack Surface of Docker

Docker introduces several categories of attack surface that do not exist in traditional server deployments or even in virtual machine environments. Understanding these categories is essential before any hardening work can begin.

The first and most critical attack surface is the Docker daemon itself. The Docker daemon, dockerd, runs as root on the host system. It listens for API requests and manages containers, images, volumes, and networks. By default, it communicates over a Unix socket at `/var/run/docker.sock`. Any user or process with access to this socket effectively has root access to the host. This is not an exaggeration. If you can communicate with the Docker daemon, you can mount the host filesystem into a container, run a privileged container, or execute arbitrary commands as root.

```
# This command demonstrates the power of Docker socket access
# An attacker with access to the Docker socket can mount the host
filesystem
docker run -v /:/hostfs -it alpine /bin/sh

# Inside this container, the entire host filesystem is accessible
at /hostfs
# This includes /etc/shadow, SSH keys, and every other sensitive
file
```

The second attack surface is the container image itself. Docker images are built in layers, and each layer may contain software packages, configuration files, credentials, or vulnerabilities. Images pulled from public registries like Docker Hub may contain known vulnerabilities, backdoors, or malicious code. Even images you build yourself can inherit vulnerabilities from their base images. A single outdated library in a base image can become the entry point for an attacker.

The third attack surface is the container runtime configuration. How a container is started matters enormously. Running a container with the `--privileged` flag disables most of the security isolation that Docker provides. Mounting sensitive host paths, running as root inside the container, exposing unnecessary ports, and granting excessive Linux capabilities all expand the attack surface.

```
# This is an example of an insecure container configuration
# Never do this in production without understanding the
implications
docker run --privileged --net=host --pid=host -v /:/host alpine /
bin/sh

# This container has:
# - Full privileges (all capabilities, access to all devices)
# - Host network namespace (can see all host network traffic)
# - Host PID namespace (can see and interact with all host
processes)
# - Host filesystem mounted
```

The fourth attack surface is the orchestration and networking layer. In production environments, Docker containers communicate with each other over Docker networks. By default, all containers on the same Docker bridge network can communicate freely. There is no network segmentation unless you explicitly create it. Service discovery, secrets management, and inter-container authentication all present opportunities for lateral movement if not properly configured.

The fifth attack surface is the build pipeline. Dockerfiles are executable instructions. A compromised Dockerfile or a malicious instruction in a multi-stage build

can introduce vulnerabilities at build time that persist into production. Build arguments, environment variables, and cached layers can all leak sensitive information.

Attack Surface	Description	Example Risk
Docker daemon	Root-level service managing all containers	Unauthorized socket access grants host root
Container images	Layered filesystem with software and config	Vulnerable base images, embedded credentials
Runtime configuration	Flags and options at container start	Privileged mode, excessive capabilities
Networking	Inter-container and external communication	Unrestricted lateral movement between containers
Build pipeline	Dockerfile instructions and build context	Leaked secrets in image layers
Host kernel	Shared operating system kernel	Kernel exploits affect all containers
Volume mounts	Host filesystem paths exposed to containers	Sensitive host files accessible to containers

The Shared Responsibility Model in Docker Security

One of the most significant reasons container security is different is the blurring of traditional security responsibilities. In a conventional infrastructure model, there are clear boundaries. The infrastructure team manages the servers and operating systems. The security team manages firewalls, intrusion detection, and access controls. The development team writes application code. Each team has a defined scope.

Docker dissolves these boundaries. A developer writing a Dockerfile is making infrastructure decisions. They are choosing a base operating system, installing system packages, configuring network exposure, setting user permissions, and defining the runtime environment. Every line in a Dockerfile is a potential security decision.

Consider this Dockerfile:

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y curl wget netcat
COPY . /app
WORKDIR /app
RUN chmod 777 /app
EXPOSE 8080
CMD ["python3", "server.py"]
```

This seemingly simple Dockerfile contains multiple security concerns. The `ubuntu:latest` tag is mutable and may pull different versions at different times, making builds non-reproducible. The installed packages `wget` and `netcat` are common tools used by attackers for downloading payloads and establishing reverse shells. The `chmod 777` command makes the application directory world-writable. The container will run as root by default because no `USER` instruction is specified. None of these issues would be caught by a traditional network firewall or intrusion detection system.

Now compare it with a security-conscious version:

```
FROM ubuntu:22.04@sha256:abc123...
RUN apt-get update && apt-get install -y --no-install-recommends
curl \
    && rm -rf /var/lib/apt/lists/*
RUN groupadd -r appuser && useradd -r -g appuser appuser
COPY --chown=appuser:appuser . /app
WORKDIR /app
USER appuser
EXPOSE 8080
```

```
CMD ["python3", "server.py"]
```

This version pins the base image by digest for reproducibility. It installs only the necessary packages and removes the package cache. It creates a non-root user and runs the application as that user. It sets appropriate file ownership. These are security decisions made by the developer, in the Dockerfile, at build time.

This reality demands a shift in organizational thinking. Security cannot be bolted on after deployment. It must be embedded in the development workflow, in the CI/CD pipeline, and in the image build process. The concept of "shifting security left," moving security practices earlier in the development lifecycle, is not just a best practice in Docker environments. It is a necessity.

Traditional Security	**Docker Security**
Infrastructure team manages OS hardening	Developers define OS in Dockerfile
Security team configures firewalls	Network policies defined in Docker Compose or orchestrator
Operations team manages runtime security	Runtime security defined in container start commands
Patching managed by sysadmins	Base image updates managed by development teams
Clear separation of duties	Shared responsibility across all teams

Real-World Consequences and Case Studies

The theoretical differences between container security and traditional security become starkly concrete when we examine real-world incidents.

In 2018, researchers discovered that thousands of Docker daemons were exposed to the internet with no authentication. Attackers were using these exposed daemons to deploy cryptocurrency mining containers. The attack was trivially simple: the Docker daemon API was accessible on port 2375 without TLS or authentication, and attackers simply issued API calls to pull and run mining containers. The host owners bore the cost of the compute resources while attackers collected the cryptocurrency.

You can check whether your Docker daemon is exposed with a simple command:

```
# Check if Docker daemon is listening on a TCP port
sudo netstat -tlnp | grep dockerd

# If you see dockerd listening on 0.0.0.0:2375, your daemon is
exposed
# The secure configuration uses TLS on port 2376 or Unix socket
only
```

The Docker daemon should never be exposed on an unauthenticated TCP port. The default configuration uses a Unix socket, which is inherently more secure because access is controlled by filesystem permissions. If remote access is required, TLS mutual authentication must be configured.

Another significant incident involved the discovery of malicious images on Docker Hub. In 2018, researchers found 17 Docker images that had been uploaded to Docker Hub containing backdoors and cryptocurrency miners. These images had been downloaded millions of times. Organizations that pulled these images and deployed them in production unknowingly ran malicious code in their environments.

This highlights a critical difference from traditional software deployment. In a traditional model, software is downloaded from known vendors, verified with checksums or signatures, and installed by administrators. In the Docker ecosystem,

pulling an image from a public registry is so easy and so fast that it can become routine, even careless. The command `docker pull` is deceptively simple, but it downloads and trusts an entire filesystem that will run on your infrastructure.

```
# Before pulling any image, verify its source and check for known
vulnerabilities
# Use Docker Content Trust to enforce image signing
export DOCKER_CONTENT_TRUST=1

# Now pulling an unsigned image will fail
docker pull untrusted/image:latest
# Error: remote trust data does not exist

# Only signed, verified images will be pulled
docker pull docker/trusteddimage:latest
```

Building the Security Mindset for Docker

Understanding why container security is different is the first step toward building a security-first approach to Docker deployments. The key principles that emerge from this understanding form the foundation for every subsequent chapter in this book.

First, defense in depth is not optional. Because the isolation boundary between containers and the host is thinner than the boundary provided by a hypervisor, you cannot rely on a single layer of defense. You need security at the image level, the runtime level, the network level, the host level, and the orchestration level.

Second, immutability is your ally. Containers are designed to be ephemeral and immutable. A running container should never be modified; instead, a new im-

age should be built and deployed. This immutability, when enforced, eliminates entire categories of attacks that rely on modifying running systems.

Third, least privilege must be the default. Every container should run with the minimum permissions required to function. This means non-root users, dropped capabilities, read-only filesystems where possible, and restricted network access.

Fourth, trust must be verified, not assumed. Every image, every base layer, every dependency, and every configuration should be verified. Image scanning, content trust, and supply chain verification are not optional extras; they are fundamental requirements.

Fifth, visibility is essential. You cannot secure what you cannot see. Container environments are dynamic, with containers starting and stopping constantly. Logging, monitoring, and auditing must be designed for this dynamic nature.

```
# A practical starting point: audit your current Docker security
posture
# Check Docker daemon configuration
docker info --format '{{.SecurityOptions}}'

# List all running containers and their security-relevant
settings
docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Status}}"

# Check if any containers are running as privileged
docker inspect --format='{{.HostConfig.Privileged}}' $(docker ps
-q)

# Check if any containers are running as root
docker inspect --format='{{.Config.User}}' $(docker ps -q)
```

These commands provide an immediate snapshot of your container security posture. If any container returns `true` for privileged mode or an empty string for user (indicating root), those are immediate areas for improvement.

The journey through Docker security begins with this fundamental understanding: containers are not virtual machines, the attack surface is broader and more nu-

anced than traditional infrastructure, security responsibilities are shared across teams, and real-world consequences are severe and well-documented. With this foundation in place, we are prepared to move into the practical work of securing Docker images, hardening runtime configurations, and building production environments that are resilient against attack.

Every chapter that follows builds on the principles established here. The techniques will become more specific, the configurations more detailed, and the tools more specialized. But the underlying truth remains constant: container security is different because containers are different, and treating them otherwise is the most dangerous assumption an organization can make.

Note: Throughout this book, all commands and configurations are tested against Docker Engine version 24.x and later. While the core security principles apply to all versions, specific command syntax and available features may vary. Always consult the official Docker documentation for your specific version.

Principle	Description	Implementation Starting Point
Defense in depth	Multiple overlapping security layers	Secure images, runtime, network, and host
Immutability	Containers are not modified after deployment	Enforce read-only filesystems, rebuild for changes
Least privilege	Minimum permissions for functionality	Non-root users, dropped capabilities, restricted mounts
Verified trust	All components are verified before use	Image signing, vulnerability scanning, base image auditing
Continuous visibility	Monitor and audit dynamic container environments	Centralized logging, runtime monitoring, security auditing

Chapter 2: Threat Modeling Docker Environments

When organizations adopt Docker, they often focus on the speed of deployment, the convenience of containerization, and the consistency it brings across development and production environments. What frequently gets overlooked, however, is the systematic process of understanding where threats exist within a Docker environment. Threat modeling is not merely an academic exercise or a checkbox on a compliance form. It is a fundamental practice that allows teams to identify, categorize, and prioritize the security risks that exist across every layer of a Docker deployment. Without it, security becomes reactive rather than proactive, and organizations find themselves patching vulnerabilities after they have already been exploited.

This chapter walks through the complete process of threat modeling as it applies specifically to Docker environments. We will examine the Docker attack surface in detail, apply established threat modeling frameworks to containerized architectures, identify the most common threat vectors, build practical threat models for real Docker deployments, and establish a foundation for the security hardening practices covered in subsequent chapters.

Understanding the Docker Attack Surface

Before you can model threats, you must first understand what you are protecting. The Docker attack surface is considerably broader than many engineers initially assume. A Docker environment is not a single monolithic system. It is a layered architecture where each layer introduces its own set of potential vulnerabilities.

At the foundation sits the host operating system. Docker containers share the host kernel, which means that a vulnerability in the kernel can potentially be exploited from within a container to gain access to the host system or other containers. This is fundamentally different from virtual machines, which each run their own kernel. The shared kernel model is one of Docker's greatest strengths for performance and efficiency, but it is also one of its most significant security considerations.

Above the host sits the Docker daemon, which runs as root by default. The daemon is responsible for building, running, and managing containers. It listens on a Unix socket, and if that socket is exposed improperly, whether through network exposure or by mounting it into a container, an attacker can gain full control over the Docker host. The Docker daemon is, in many ways, the crown jewel of a Docker environment from an attacker's perspective.

The container runtime, typically containerd and runc, handles the actual creation and execution of containers. Vulnerabilities in these components, such as the infamous CVE-2019-5736 in runc, can allow container escapes where a malicious process inside a container overwrites the host runc binary and gains root access on the host.

Container images represent another critical part of the attack surface. Images pulled from public registries may contain outdated software with known vulnerabil-

ities, embedded malware, or misconfigured services. Even trusted base images can introduce risk if they are not regularly scanned and updated.

Networking within Docker introduces its own set of concerns. By default, containers on the same Docker network can communicate freely with one another. If an attacker compromises one container, they can potentially pivot to others on the same network. Docker's default bridge network, overlay networks, and exposed ports all present potential entry points.

Finally, volumes and persistent storage create pathways between the container filesystem and the host filesystem. Improperly configured volume mounts can expose sensitive host files to containers or allow containers to write malicious data to the host.

The following table provides a comprehensive overview of the Docker attack surface layers and their associated risks:

Attack Surface Layer	Description	Primary Risk	Example Scenario
Host Kernel	Shared kernel between host and all containers	Kernel exploits leading to container escape	Dirty COW vulnerability exploited from within a container
Docker Daemon	Central management process running as root	Full host compromise if daemon access is obtained	Docker socket mounted inside a container allowing arbitrary container creation
Container Runtime	containerd and runc handling container lifecycle	Container escape through runtime vulnerabilities	CVE-2019-5736 allowing runc binary overwrite
Container Images	Base images, application layers, and dependencies	Vulnerable software, embedded malware, secrets in layers	Public image containing a cryptocurrency miner in a hidden layer

Docker Networking	Bridge networks, overlay networks, port mappings	Lateral movement, network sniffing, service exposure	Attacker pivoting from compromised web container to database container
Volumes and Storage	Bind mounts, named volumes, tmpfs mounts	Host filesystem access, data exfiltration	Bind mount of /etc allowing container to read host shadow file
Docker Registry	Image storage and distribution	Supply chain attacks, image tampering	Compromised registry serving backdoored images
Orchestration Layer	Docker Compose, Docker Swarm configurations	Misconfiguration, secret exposure, privilege escalation	Swarm join tokens exposed in environment variables

Understanding each of these layers is a prerequisite for effective threat modeling. You cannot protect what you do not understand, and in Docker environments, the interconnected nature of these layers means that a weakness in one area can cascade into a compromise across the entire system.

Applying the STRIDE Framework to Docker

STRIDE is a well-established threat modeling framework developed at Microsoft that categorizes threats into six types: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Applying STRIDE to Docker environments provides a structured approach to identifying threats across every component.

Spoofing in a Docker context involves an attacker impersonating a legitimate entity. This could manifest as a compromised Docker registry serving malicious images that appear to be legitimate, or an attacker spoofing a container's identity

within a Docker network. Consider a scenario where an attacker pushes a malicious image to a public registry using a name that closely resembles a popular official image. When a developer pulls what they believe is the legitimate image, they unknowingly deploy a compromised container.

To check the provenance of Docker images and guard against spoofing, you should always use Docker Content Trust:

```
export DOCKER_CONTENT_TRUST=1
docker pull nginx:latest
```

When Docker Content Trust is enabled, Docker will only pull images that have been signed by the publisher. If the signature does not match or is absent, the pull operation will fail.

Tampering refers to unauthorized modification of data or code. In Docker, this includes modifying container images after they have been built, altering container configurations at runtime, or tampering with data in shared volumes. An attacker who gains access to a Docker host could modify running containers, inject malicious processes, or alter the Dockerfile used in a CI/CD pipeline.

You can verify the integrity of a Docker image by checking its digest:

```
docker images --digests
docker pull
nginx@sha256:a8281ce42034b078dc7d88a5bfe6cb63e918f8e65e7b3c
tried4b0a86e81e2d4f
```

Repudiation involves the ability of an attacker to deny their actions. Docker environments that lack proper logging and auditing are vulnerable to repudiation threats. If container activity is not logged, an attacker can compromise a container, exfiltrate data, and leave no trace of their actions.

Configuring the Docker daemon to use a logging driver that sends logs to a centralized system is essential:

```
{
  "log-driver": "syslog",
  "log-opts": {
    "syslog-address": "tcp://logserver:514",
    "tag": "docker/{{.Name}}"
  }
}
```

This configuration in the Docker daemon configuration file (typically located at `/etc/docker/daemon.json`) ensures that all container logs are forwarded to a centralized syslog server where they can be monitored and retained.

Information Disclosure is one of the most prevalent threats in Docker environments. Secrets hardcoded in Dockerfiles, environment variables containing database credentials, and sensitive configuration files baked into images are all common vectors for information leakage. Every layer of a Docker image is stored and can be inspected, which means that even if a secret is deleted in a later layer, it remains accessible in the image history.

To demonstrate how easily secrets can be extracted from image layers:

```
docker history --no-trunc myapp:latest
docker inspect myapp:latest
docker save myapp:latest -o myapp.tar
tar -xf myapp.tar
```

Each of these commands reveals different aspects of the image, and any secrets embedded during the build process will be visible.

Denial of Service in Docker can take many forms. A container without resource limits can consume all available CPU, memory, or disk I/O on the host, effectively denying service to other containers. Fork bombs, memory leaks, and disk-filling attacks within containers can all impact the host and neighboring containers.

Setting resource constraints is a critical defense:

```
docker run -d \
  --name webapp \
```

```
--memory="512m" \
--memory-swap="512m" \
--cpus="1.0" \
--pids-limit=100 \
nginx:latest
```

This command limits the container to 512 megabytes of memory with no swap, one CPU core, and a maximum of 100 processes, preventing it from consuming excessive host resources.

Elevation of Privilege is the most severe category of threat in Docker environments. Running containers as root, granting excessive Linux capabilities, using privileged mode, and mounting the Docker socket into containers are all pathways to privilege escalation. An attacker who achieves elevation of privilege can break out of the container and gain root access on the host.

The following table maps each STRIDE category to specific Docker threats and recommended mitigations:

STRIDE Category	Docker Threat	Mitigation
Spoofing	Malicious images impersonating legitimate ones	Enable Docker Content Trust, use private registries, verify image digests
Tampering	Modification of images or running containers	Use read-only filesystems, sign images, implement integrity monitoring
Repudiation	Untracked container activity	Centralized logging, audit daemon events, enable Docker audit rules
Information Disclosure	Secrets in image layers, exposed environment variables	Use Docker secrets, multi-stage builds, secret management tools

Denial of Service	Resource exhaustion by containers	Set memory, CPU, and PID limits on all containers
Elevation of Privilege	Container escape, root access	Run as non-root, drop capabilities, never use privileged mode

Common Docker Threat Vectors in Production

Moving beyond the theoretical framework, it is important to examine the specific threat vectors that attackers exploit in real production Docker environments. These are not hypothetical scenarios. They are patterns observed in actual security incidents.

The first and most commonly exploited vector is the exposed Docker daemon. When the Docker daemon is configured to listen on a TCP port without TLS authentication, anyone who can reach that port has full control over the Docker host. This is equivalent to giving root SSH access without a password. Attackers routinely scan the internet for exposed Docker daemons on port 2375 (unencrypted) and port 2376 (TLS).

```
# DANGEROUS: Never do this in production
dockerd -H tcp://0.0.0.0:2375

# CORRECT: Use TLS authentication
dockerd --tlsverify \
  --tlscacert=/etc/docker/tls/ca.pem \
  --tlscert=/etc/docker/tls/server-cert.pem \
  --tlskey=/etc/docker/tls/server-key.pem \
  -H tcp://0.0.0.0:2376
```