Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 1, 1243-1261
2025
Publisher: Learning Gate
DOI: 10.55214/25768484.v9i1.4382
© 2025 by the authors; licensee Learning Gate

The role of container security in application development

Zlatan Morić[1*], Jasmin Redžepagić[2], Ivan Bencarić[3], Vedran Dakić[4]
[1,2,3]Algebra University, Department of cybersecurity and operating systems, Croatia; zlatan.moric@algebra.hr (Z.M.) jasmin.redzepagic@algebra.hr (J.R.) ibencar@algebra.hr (I.B.).
[4]Algebra University, Croatia; vedran.dakic@algebra.hr (V.D.).

Abstract: This paper explores the importance of container security in modern ap-plication development, particularly in environments using microservices architectures and DevOps practices. Containers, by providing lightweight and portable virtualization, have transformed application deployment, enabling rapid and consistent transitions across development, testing, and production stages. However, their architecture and ease of use introduced unique vulnerabilities. This paper investigates the implications of these security concerns and outlines best practices to mitigate them. Examining containerization fundamentals, the paper identifies core vulnerabilities, including misconfigurations, runtime attacks, and orchestration layer risks. Techniques like namespace isolation, resource allocation controls, and security tools are evaluated for their effectiveness in hardening containerized environments. Advanced methods, such as role-based access control, vulnerability scanning, and secrets management, are emphasized for securing CI/CD pipelines. The findings underscore the necessity of integrating robust security measures throughout the container lifecycle to protect sensitive data and maintain application integrity. By adopting a comprehensive container security strategy, organizations can balance the scalability and agility of containers and maintain the reliability and safety of their deployment infrastructure.
Keywords: *Containers, Docker, Kubernetes, Microservices, Pipelines, Security.*

1. Introduction

This paper examines the essential function of container security in contemporary application development, highlighting its importance in microservices designs and DevOps methodologies. It thoroughly analyzes containerization basics, highlighting critical vulnerabilities like misconfigurations, runtime assaults, and orchestration layer hazards. It recommends robust security measures, encompassing namespace separation, resource allocation rules, and adopting sophisticated techniques such as role-based access control, vulnerability assessment, and secrets management. The results emphasize the necessity of incorporating stringent security measures throughout the container lifecycle to maintain application integrity and protect sensitive information.

This paper has two main contributions to the field:

- It comprehensively examines fundamental vulnerabilities in containerized settings and their consequences for contemporary application deployment;
- It delineates a thorough framework for enterprises to reconcile containers' scalability and agility with their infrastructure's reliability and safety, thereby solving a significant deficiency in container security research.

The rest of this paper is organized as follows. Section 2 presents the literature review related to the research field, followed by the section about the fundamentals of containerization and its role in modern application development. Section 4 delves into specific container security challenges, exploring

© 2025 by the authors; licensee Learning Gate
History: Received: 4 December 2024; Revised: 9 January 2025; Accepted: 17 January 2025; Published: 22 January 2025
* Correspondence: zlatan.moric@algebra.hr

vulnerabilities and mitigation strategies. Section 5 discusses the broader threat landscape, including image vulnerabilities, runtime attacks, and orchestration layer risks. Sections 6 and 7 detail the essential components of container security, such as runtime, image, and host security. Section 8 highlights security best practices for container development and deployment, focusing on automation and shift-left strategies. Finally, sections 9 and 10 discuss emerging trends in container security, such as zero-trust architectures and AI-driven threat detection, setting the stage for future research and development in the field, as well as the paper's conclusions.

2. Related Works

Containers are a light and portable form of virtualization that can package an application and all its dependencies in the form of an "image," which allows them to run consistently across different computing environments [1]. In contrast to standard virtual machines, which require separate operating system instances for each application, containers share the host operating system kernel but isolate application processes, creating a much more resource-efficient model [2]. Technologies like Docker can build, deploy, and manage these containers [3] and their images, while orchestration platforms like Kubernetes [4] automate containerized applications' deployment, scaling, and management over many machines organized in clusters.

Due to their efficiency, scalability, and flexibility, containers have become popular in application development, especially in environments that follow DevOps and CI/CD principles. Containers can provide faster software delivery by ensuring consistent behavior from development through production. Additionally, they reduce infrastructure costs and enhance application portability, making them ideal for modern, microservice-based architecture.

The popularity of containers has paralleled the rise of DevOps [5] a development approach emphasizing collaboration, automation, and continuous improvement between software development and IT operations.

Containers play a significant role in DevOps by providing rapid but consistent deployments. At the same time, they enable Continuous Integration and Continuous Deployment (CI/CD) pipelines, which are a critical part of DevOps. With containers, applications and all the requirements that form their dependencies can be bundled into isolated, portable units, ensuring they function uniformly across development, testing, and production environments. This consistency minimizes the "it works on my machine" problem, streamlines testing, and can accelerate delivery. Additionally, container orchestration platforms like Kubernetes automate tasks such as scaling and managing deployments, allowing teams to handle complex applications more quickly than when using traditional approaches.

Figure 1.
CI/CD pipeline (from Red Hat [6].

This automation fits perfectly with DevOps principles, providing efficiency and supporting agile methodologies where frequent, reliable software updates are critical. As a result, containers have become an indispensable part of a modern DevOps process, allowing organizations to innovate faster while maintaining high standards of quality and reliability.

As with all new technologies, containers also raise various security concerns that must be addressed [6]. Their security plays a critical role because while offering powerful benefits like scalability and

© 2025 by the authors; licensee Learning Gate

consistency, containers also introduce unique vulnerabilities that can compromise application integrity and data protection. Containers share the host operating system kernel, which can increase the risk of "breakout" attacks—where malicious processes in a container access or exploit the host or other containers [7].

In modern application development, where microservices architectures are common, a single vulnerability in one container can expose the entire system to risks. Additionally, the extensive use of third-party container images from public repositories can present supply chain vulnerabilities. As containerized environments are often deployed rapidly through automated CI/CD pipelines [8] security needs to be integrated into every stage of development to detect vulnerabilities early, enforce configuration best practices, and minimize potential attack surfaces.

Robust container security measures are essential to protecting sensitive data, maintaining trust, and ensuring the stability and reliability of applications in production.

3. Containerization and Application Development

Containers package an application with all its dependencies, such as libraries, binaries, and configuration files, into a single, self-contained unit that can run consistently across various environments. This approach eliminates the need for an individual operating system instance for each application, as is required with virtual machines (VMs), making containers significantly lighter and more efficient. Containers achieve this isolation through kernel features like namespaces and control groups (cgroups) [1].

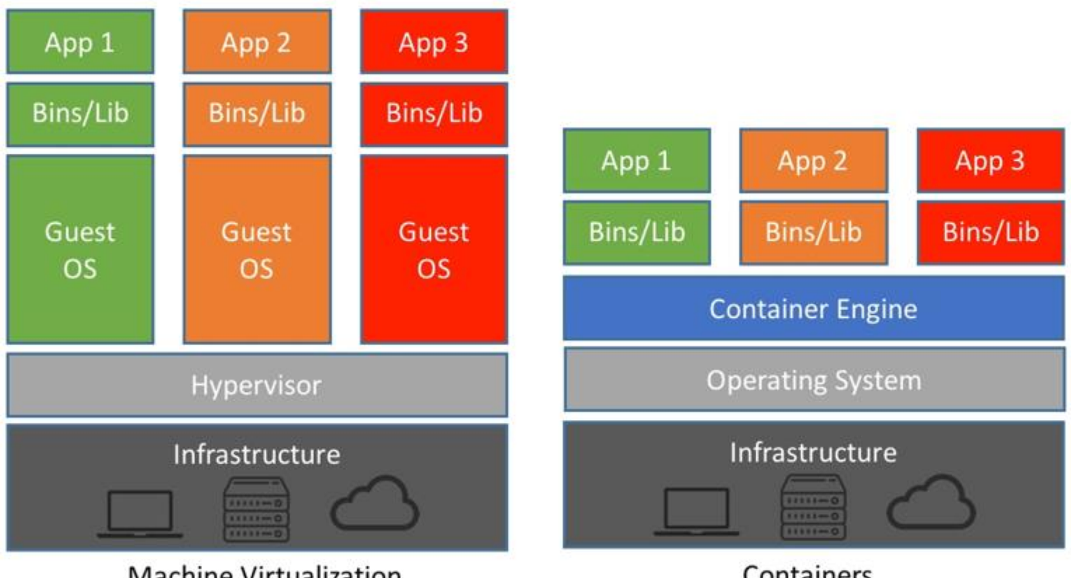

Figure 2.
Virtual Machines and Containers.

Namespaces provide process isolation by limiting what each container can see and access within the system, including its file system, network, process IDs, and user environments.

Control groups, on the other hand, regulate resource allocation, such as CPU, memory, and disk I/O, ensuring that each container remains within predefined resource limits, avoiding interference with other containers or the host. This lightweight isolation allows multiple containers to run on the same host OS without the overhead of running separate guest OS instances, as with VMs.

© 2025 by the authors; licensee Learning Gate

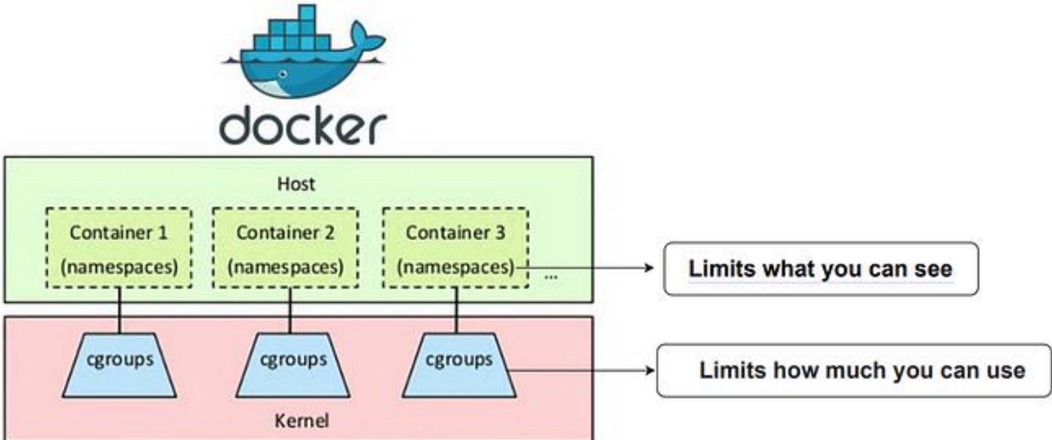

Figure 3.
Namespaces and Cgroups.

At runtime, each container operates as an independent process on the host, sharing its OS kernel but with its own isolated user space, making it portable and consistent across different environments. Tools like Docker enable developers to define container configurations in a Dockerfile [9] specifying the base image, application code, dependencies, and required configuration. This Dockerfile is built into an image, a read-only template for creating containers. Images are portable and can be stored and shared through container registries, such as Docker Inc [10] allowing easy replication across teams and environments.

Orchestration platforms like The Kubernetes Authors [4] Enable the management of containers at scale by handling the deployment, scaling, and monitoring of containerized applications across multiple nodes, making it easier for IT teams to maintain consistency and high availability across large-scale environments. This approach to containerization has redefined application deployment by enabling rapid, consistent, and efficient transitions from development to production, even in complex, distributed systems.

3.1. Benefits of Containers

Containers significantly increase efficiency in modern software development by minimizing overhead and resource usage compared to traditional virtualization. Since containers share the host kernel, they eliminate the need for separate OS instances and drastically reduce the RAM, CPU, and storage footprint. This lightweight nature translates to faster startup times, benefiting CI/CD pipelines. Additionally, containers are often ephemeral, meaning they can be instantiated, scaled, and destroyed rapidly based on demand. This allows for more responsive resource allocation, improved server utilization, and cost-effective operation. From a scaling perspective, containers can be replicated or scaled horizontally as application loads fluctuate, making it simple to manage many container instances using container orchestration systems like The Kubernetes Authors [11]. Because these orchestrators handle load balancing, auto-scaling, and failover, containers can efficiently maintain high availability and performance even under unpredictable usage patterns.

Moreover, portability is a key strength that makes containers extremely important in microservices architectures. By bundling the application and its dependencies into a self-contained image, teams can run these images in development, testing, or production environments without worrying about version conflicts or missing libraries. This approach enables agile, modular design, enabling microservices to be independently developed, deployed, and updated. Each microservice container can be scaled or rolled

© 2025 by the authors; licensee Learning Gate

back without disrupting the rest of the system, supporting faster release cycles and continuous improvement. Orchestration platforms further streamline the communication between services, load-balancing traffic, and managing network policies.

3.2. New Attack Surfaces

Despite their many benefits, containers introduce a range of new attack surfaces and vulnerabilities that developers and operations teams must address. One key concern is that containers share the underlying host OS kernel, potentially enabling a "container breakout" attack [12] if an exploit allows malicious processes to escalate privileges and affect the host. Because multiple containers may share the same kernel, a breach in one container could provide the door for lateral movement across other containers or the host if proper isolation controls aren't in place. Furthermore, container images often pull in libraries and dependencies from public repositories, which might contain vulnerabilities or malicious code; without thorough scanning and verification, organizations risk introducing security gaps directly into their production environments. Misconfigurations such as running containers with overly permissive privileges or allowing unnecessary system calls further widen the attack surface, giving adversaries more pathways to exploit.

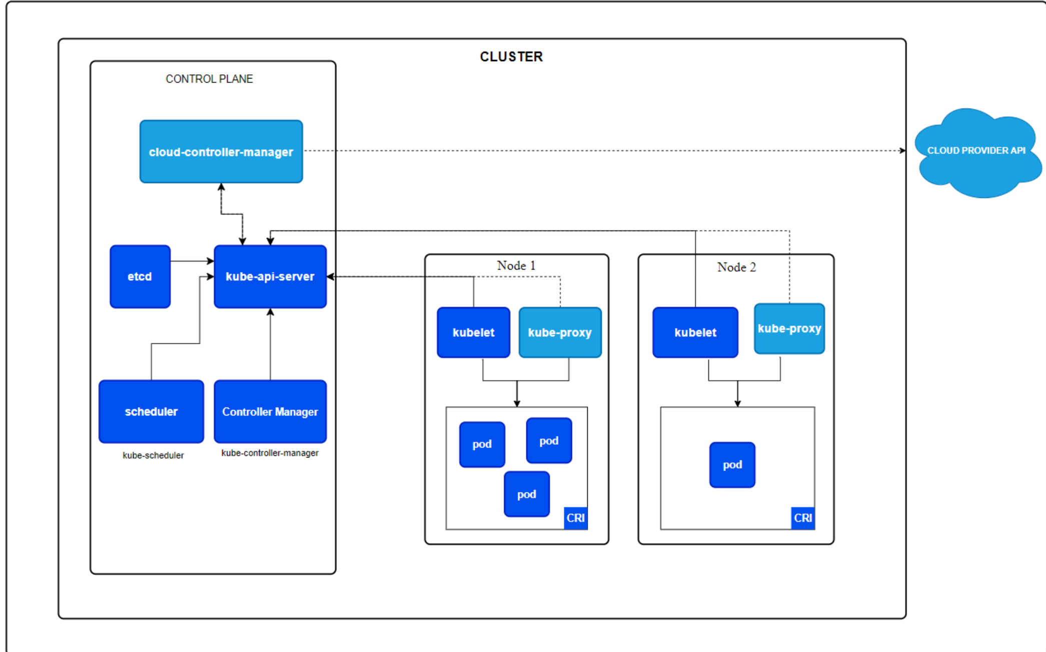

Figure 4.
Kubernetes cluster components (from The Kubernetes Authors [13].

Another challenge is that container orchestration platforms, like Kubernetes, bring additional complexity and, therefore, more potential points of failure (Figure 4). Exposed application programming interfaces (APIs), insecure access controls, and misapplied Role-Based Access Control (RBAC) configurations can all lead to unauthorized control over the containerized environment [6]. Attackers can exploit these weaknesses by launching denial-of-service attacks, intercepting network traffic, or enabling backdoors inside containers.

Additionally, the ephemeral nature of containers means that logs and other forensic evidence might not persist, complicating incident detection and response efforts. As a result, organizations using containers must adopt a complete security strategy—including secure image creation, continuous

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 1: 1243-1261, 2025
DOI: 10.55214/25768484.v9i1.4382
© 2025 by the authors; licensee Learning Gate

vulnerability scanning, robust runtime security policies, and strong governance over orchestration platforms—to mitigate the unique risks associated with container-based deployments [14].

4. Container Security

When comparing container-based security with the security of traditional monolithic applications, one of the most essential differences lies in resource and OS isolation. In traditional environments, mainly when using virtual machines, each VM typically runs its operating system instance, creating a relatively clear boundary for security controls. This helps compartmentalize attacks since a breach in one virtual machine does not necessarily lead to compromise in another. Monolithic applications, which often run on these dedicated servers or VMs, generally have fewer moving parts, simplifying security monitoring and patch management. However, they may also suffer from larger attack surfaces at the application level since all functionality is bundled together in a single, often complex process. Scaling and updating these monolithic systems can be problematic, which can, in turn, delay security patches or updates, leaving known vulnerabilities unpatched for more extended periods.

On the other hand, containers share the host OS kernel while isolating applications at the process level using features like namespaces and cgroups. This approach makes containerized environments lighter and more scalable, creating new security challenges. If an attacker exploits the kernel or a container breakout vulnerability, they can potentially pivot to other containers or escalate privileges on the host. Additionally, the microservices-based architecture that containers commonly support involves numerous interconnected services, each potentially running in multiple containers, which increases the overall number of network endpoints and communication paths. These factors expand the security surface area and complicate monitoring. Developers must implement robust isolation settings (like SELinux Red Hat [15] and AppArmor [16]) and follow best practices such as running containers in non-root or “rootless” modes, regularly scanning images for known vulnerabilities, and automating patching. This more modular ecosystem also means quicker patching and rolling upgrades are possible, but only if proper DevSecOps [17] practices are in place to handle the rapid generation of container versions.

Container security architecture is built around process-level isolation and resource management, leveraging Linux kernel features such as namespaces and cgroups. Namespaces isolate core systems, like process IDs, network stacks, and user IDs—so each container perceives a dedicated view of the operating environment. For example, the PID namespace ensures a container only sees its processes, while the network namespace provides a private network stack. cgroups, however, manage how much CPU, memory, and disk I/O each container can use, preventing one container’s resource-intensive activity from starving others or the host. Namespaces and cgroups enforce firm boundaries between containers, effectively limiting the damage a compromised container can inflict. However, because containers share the host OS kernel, this model still demands kernel patching and tight configuration of kernel security mechanisms to mitigate the risk of container breakout attacks.

Additionally, container images are typically minimalistic, containing only the libraries and dependencies necessary for the application to run, which helps reduce the overall attack surface. You limit the number of potential entry points for malicious activity by omitting unneeded packages or tools. To maintain security, these images should be built from trusted base images and regularly scanned for known vulnerabilities—particularly when pulling from public registries. Implementing a “secure build” process in the CI/CD pipeline, combined with techniques like image signing, can further ensure the integrity and security of container images.

5. Container Threat Landscape

The container threat landscape has markedly expanded due to the increasing popularity of containerized systems in contemporary software development and deployment. Containers provide efficiency and mobility but also present distinct security problems that businesses must confront. The

© 2025 by the authors; licensee Learning Gate

transient and fluid characteristics of containers increase the assault surface, necessitating a thorough comprehension of possible threats. Primary areas of concern encompass image vulnerabilities, wherein insecure or obsolete container images pose hazards; runtime attacks, which target active containers during execution; and breakout assaults, which allow attackers to breach the container barrier and get access to the host system. Moreover, the orchestrator layer—overseeing containerized workloads—presents dangers such as privilege escalation and misconfiguration. Supply chain attacks, which focus on dependencies and third-party components, underscore the interconnectedness of container ecosystems. Ultimately, networking hazards highlight the necessity of safeguarding container-to-container and external communications. The varied threat landscape requires proactive measures to protect containerized settings.

5.1. Image Vulnerabilities

Container image vulnerabilities arise when the base images or included libraries and packages contain unpatched flaws, outdated components, or malicious code. Because containers package the entire application stack—including dependencies and configurations—any vulnerability in these components can become an entry point for attackers. Many teams source their base images from public registries, which might not always be thoroughly vetted, increasing the risk of incorporating hidden security flaws into production environments. Additionally, developers may unintentionally introduce weak configurations (e.g., unnecessary ports, default credentials), further widening the attack surface. Regular image scanning, using reliable sources for base images, and adhering to best practices such as signing and verifying images can help mitigate these risks.

5.2. Runtime Attacks

Container runtime attacks refer to exploits that occur once a container is actively running, taking advantage of vulnerabilities in the application code, misconfigurations, or the underlying container infrastructure. Attackers might escalate privileges through the container's runtime environment, for example, by abusing excessive privileges (running as root) or weak security policies, such as unconfigured Seccomp (Secure Computing Mode) [18] AppArmor, or SELinux. In some cases, attackers leverage vulnerabilities to perform container escapes—gaining access to the host machine's resources or neighboring containers. The ephemeral nature of containers can also complicate detection, as logs and forensic data may disappear when a container is stopped or replaced.

Robust runtime security relies on best practices such as applying the principle of least privilege, continuously monitoring container behavior, strictly enforcing security profiles, and maintaining clear logs even after container termination.

5.3. Breakout Attacks

Container breakout attacks [12] occur when an attacker exploits vulnerabilities within a container or the underlying host to escape the container's isolation and access host-level resources or other containers. Because containers rely on shared kernel features (like namespaces and cgroups) rather than full-blown virtualization, a vulnerability in the kernel or container runtime configurations (e.g., privileged containers) can open a path for these attacks. Misconfigurations—such as running containers with the --privileged flag or granting unnecessary system calls—can further increase the risk of a successful breakout. Once inside the host environment, attackers can move laterally, escalate privileges, and potentially compromise other containers or system services.

Mitigations typically include running containers with the least privilege necessary, restricting system calls using Seccomp or AppArmor, keeping the host and kernel fully patched, and regularly auditing container security policies. Ensuring proper monitoring and logging is also crucial, as container breakouts may go unnoticed without robust observability and incident response processes.

© 2025 by the authors; licensee Learning Gate

5.4. Orchestration Layer Risks

Container orchestration platforms like Kubernetes introduce additional security complexities and risks beyond standalone container deployments. Misconfigurations of critical components—like Kubernetes' API server or the etcd key-value store—can lead to unauthorized access or data exposure. If RBAC is misconfigured, attackers can escalate privileges to run malicious containers or intercept network traffic between services. Additionally, the distributed nature of container orchestration means that multiple nodes, services, and networking components must be adequately secured, including kubelets, ingress controllers, and network policies. Attackers may exploit unsecured API endpoints to manipulate configurations or deploy rogue workloads.

To minimize risks in the orchestration layer, it is critical to ensure secure and encrypted communication among all cluster components, apply least-privilege access controls, keep clusters patched, and actively monitor cluster events.

5.5. Supply Chain Attacks

Container supply chain risks arise from the complex network of dependencies and sources organizations rely on when building container images. Modern development often involves pulling base images and third-party libraries from public registries, where malicious actors might upload compromised or spoofed images to exploit unsuspecting users. Insecure or unverified downloads can introduce backdoors or malicious code directly into production environments. Additionally, vulnerabilities or misconfigurations within the DevOps toolchain—such as insecure CI/CD pipelines—can be leveraged to inject harmful components into otherwise legitimate images.

Organizations should mitigate these risks by scanning images for known vulnerabilities, employing image signing and verification, and restricting downloads to trusted registries or private repositories. Integrating security controls and verifying integrity at every stage of the pipeline—from development to production—ensures that only approved and secure assets make their way into containerized applications.

5.6. Networking Risks

Container networking risks arise from the complex, often ephemeral nature of containers communicating internally and externally. In many orchestration environments, network plugins automatically assign IPs and manage routes, which can create hidden pathways for attackers to exploit if misconfigured or left unsecured. Containers frequently share virtual bridges or overlays, increasing the risk of lateral movement once an adversary gains a foothold in the network. Additionally, traffic between containers may not be encrypted by default, allowing for potential eavesdropping or interception of sensitive data. Microservices architectures can further complicate network security by introducing numerous endpoints and routes for each service, making it harder to maintain consistent policies.

To mitigate these risks, engineers should implement strict network segmentation, restrict unnecessary service exposure, employ mutual TLS [19] where possible, and utilize network policies or firewalls that enforce least-privilege access between containerized workloads.

6. Components of Container Security

Container security is essential for protecting contemporary application environments and guaranteeing the confidentiality, integrity, and availability of containerized workloads. Securing all components inside the container ecosystem is critical for lowering risks, as containers insulate applications from the underlying infrastructure. A holistic strategy for container security encompasses various layers. Image security guarantees container images are devoid of vulnerabilities and adhere to best practices before deployment. Runtime security emphasizes surveillance and safeguarding containers during operation, identifying irregularities, and thwarting illegal activities. Network security protects

© 2025 by the authors; licensee Learning Gate

communication between containers and external systems, implementing controls to inhibit lateral movement. Secrets management secures sensitive information within containers, such as API keys and credentials. Finally, host security guarantees that the foundational architecture is fortified against potential vulnerabilities that could jeopardize the ecosystem. By addressing these elements, businesses may establish strong defenses for containerized environments.

6.1. Image Security

Container image security relies heavily on regular scanning to detect known vulnerabilities in libraries, dependencies, and configurations. Tools like Clair or Trivy can integrate into CI/CD pipelines, automatically scanning images at a build or before deployment and flagging security issues such as outdated packages, CVEs, or misconfigurations.

By integrating these scanners early and often in the development lifecycle, teams can detect and fix vulnerabilities before they make it to production. Many scanning tools offer detailed remediation guidance, helping engineers resolve issues quickly. Beyond scanning, it is essential to ensure that images are rebuilt against patched dependencies and are not inherited from outdated or untrusted base images.

Verifying image authenticity is an equally important aspect of container image security, typically via image signing. Image signing ensures that only trusted images—signed by a recognized entity—can be pulled and run, preventing attackers from injecting malicious images into repositories or tricking users into deploying counterfeit images. Implementing a policy enforcement layer to reject unsigned or untrusted images strengthens this approach. One additional major strategy is to use minimal base images, which include only the essential components needed for the application to run. This practice reduces the overall attack surface by eliminating unnecessary packages and libraries that could contain hidden vulnerabilities.

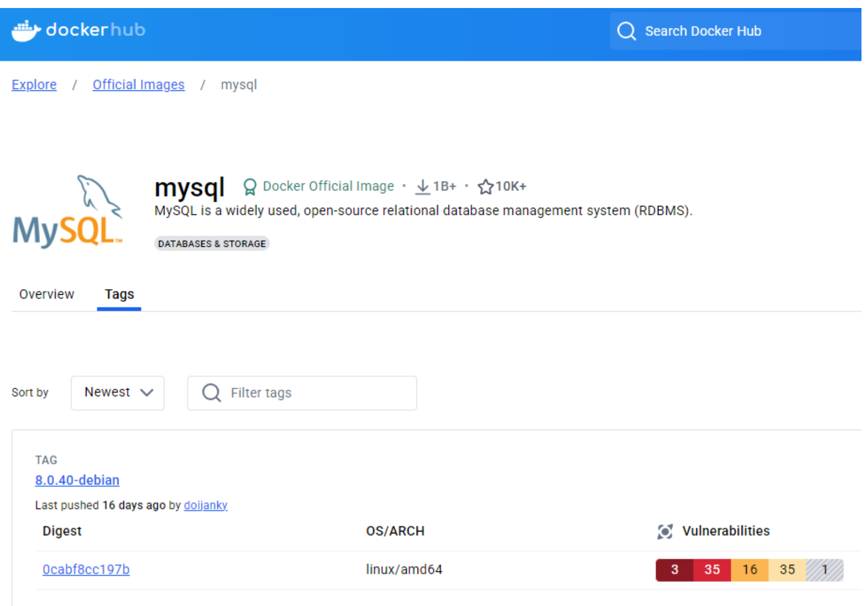

Figure 5.
Vulnerability scanning as a part of Docker Hub (from Docker Inc [10]).

Cloud image repositories like Docker Hub and Quay often provide integrated image scanning and signing features to enhance container security. Docker Hub, for instance, offers vulnerability scanning

DOI: 10.55214/25768484.v9i1.4382
© 2025 by the authors; licensee Learning Gate

(powered by tools such as Snyk [20]) that checks images for known CVEs while supporting Docker Content Trust for image signing to ensure authenticity. Conversely, Quay can integrate with Clair [21] for vulnerability scanning and supports image signing mechanisms.

6.2. Runtime Security

Container runtime security begins with minimizing privileges allocated to each container, ensuring that only the necessary rights are granted for the application to function. For example, when using Docker, containers can be run as a non-root user by specifying the --user flag or by defining a user directive in Dockerfile (e.g., USER 1000). This practice prevents attackers from having unrestricted root access if they manage to compromise the container. Another best practice is to drop unnecessary Linux capabilities, which by default can grant privileges beyond what is needed, further reducing the attack surface. Avoiding the --privileged flag except, in particular, controlled scenarios is also crucial, as privileged containers can effectively operate with host-level permissions, making them a prime target for malicious actors.

Additionally, Linux kernel security modules are essential in restricting container actions at runtime. Docker Inc [18] allows the filtering of system calls, blocking or limiting those containerized applications that should not be invoked. AppArmor [16] and SELinux [15] provide mandatory access control (MAC) frameworks that tightly define what processes can access, from file paths to network interfaces. Enforcing such policies can contain the damage of a breach and prevent unauthorized actions inside the container.

Beyond Docker, alternative runtimes like Podman [22] or CRI-O [23] are often touted for security benefits. Podman, for instance, supports rootless containers by default, letting containers run with user-level privileges rather than root-level, thereby reducing the impact of a potential compromise. CRI-O, purpose-built for Kubernetes, also integrates well with these security modules, providing a streamlined, minimalistic runtime that can be easier to harden than more general-purpose container engines.

6.3. Network Security

Container network security fundamentally relies on network segmentation to limit each container's exposure and control traffic flow between services. By creating logically isolated networks or using technologies like Kubernetes Network Policies [24] operators can define which containers can communicate with one another, effectively locking down routes that should remain inaccessible. This segmentation approach reduces lateral movement risks—if one container is compromised, the attacker's ability to target other containers is contained by strict access controls. Additionally, network segmentation can be implemented at multiple layers, such as isolating microservices within their namespaces or subnets, ensuring that only the minimum required connections are permitted for regular operation.

An essential addition to segmentation is encrypting communications between containerized services. When network traffic traverses potentially untrusted environments—like the public internet or even a shared internal network—encryption safeguards sensitive data and credentials from interception or tampering. Implementing mutual TLS [19] is particularly effective, as it encrypts data in transit and ensures strong authentication between client and server containers. In Kubernetes environments, service meshes like Istio or Linkerd can automate this process by injecting sidecar proxies that handle certificate management and traffic encryption without requiring changes in application code.

6.4. Secrets Management

Secrets management is a critical aspect of containerized application security, as improperly stored or handled credentials—such as API keys, passwords, and tokens—can serve as prime entry points for

© 2025 by the authors; licensee Learning Gate

attackers. Storing secrets directly within container images or as plain-text environment variables makes them easy to expose, whether through runtime logs, container registry leaks, or compromised images.

Instead, centralized solutions like HashiCorp Vault [13] should be used. Vault provides a secure, encrypted store for sensitive data and dynamic secrets that expire after a set time. Vault's integration with container orchestration platforms allows secrets to be injected on-demand, ensuring that applications only have access to the credentials they need for as long as they need them. This limits the potential impact of a breach and makes it easier to rotate secrets or revoke compromised credentials quickly.

In Kubernetes, the native approach to secrets management involves using Kubernetes Secrets [25] which can be mounted as volumes or exposed as environment variables to running containers. Kubernetes Secrets are base64-encoded objects, and while not inherently encrypted by default, they can be protected with encryption at rest in the cluster's etcd store if properly configured. RBAC policies can further secure who can read or modify the secret objects within the cluster. Combining these built-in capabilities with an external secrets manager (such as Vault) can add another layer of security and flexibility, particularly in large-scale or multi-tenant environments. Ultimately, any effective secrets management strategy should emphasize short-lived credentials, automated rotation, and secure storage and retrieval mechanisms to minimize the exposure window if credentials are ever compromised.

6.5. Host Security

The security of a container host is vital because containers share the host operating system kernel. If the host is compromised, attackers can access every container running on it. Consequently, keeping the host operating system up to date with security patches, particularly kernel-level patches, is a top priority. Many distributions offer live patching mechanisms (e.g., Canonical's Livepatch for Ubuntu [26]) that allow kernel updates without downtime, which is especially valuable in production environments where high availability is essential. Access control measures, such as SSH hardening (disabling root logins, using key-based authentication), implementing robust firewall rules, and restricting which users or automation systems can modify container configurations reduce the likelihood of unauthorized changes. Applying the principle of least privilege to host accounts, limiting administrative rights, and routinely auditing login and sudo access fortify the container host.

In this context, operating systems like Fedora CoreOS [27] offer a strong foundation for securing container hosts. Fedora CoreOS is a minimal, automatically updating, container-focused operating system designed to provide a streamlined, secure base for running containerized workloads. By default, it includes only the essential components needed to run containers, reducing the attack surface and simplifying system administration. Automatic updates ensure that Fedora CoreOS frequently applies the latest security patches, mitigating the risk of known vulnerabilities. Additionally, its immutable file system approach limits unauthorized or accidental changes, further securing the underlying OS. When combined with strict access control, running containers as non-root, and other best practices, Fedora CoreOS and similar container-focused operating systems help organizations maintain a secure, consistent, and easily managed environment for their container deployments.

7. Container Orchestration and Security

Kubernetes orchestrates containers by abstracting away individual hosts and grouping them into clusters, then automating tasks like deployments, container scheduling, and scaling. While this process streamlines operational workflows, it also adds complexity to security management because there are multiple layers to consider—pods, services, nodes, and even cluster-level components such as the API server, etcd, and controllers. Kubernetes dynamically assigns IP addresses and routes traffic between containers, making the environment far more ephemeral and distributed than in a traditional monolithic setup. This increases the potential attack surface, as each pod or service endpoint becomes a point that must be secured. Additionally, misconfigurations in cluster components—like insecure Kubernetes API

© 2025 by the authors; licensee Learning Gate

server settings, permissive network policies, or overly broad authentication tokens—can open the door to unauthorized access.

7.1. Role-Based Access Controls

RBAC [28] in Kubernetes is a crucial security measure to mitigate these risks, enabling granular control over what actions users and service accounts can perform on cluster resources. Administrators define roles (or cluster-wide cluster roles) that specify permissions—such as listing pods or modifying deployments—and then bind these roles to subjects (users, groups, or service accounts). This ensures a least-privilege model so that each component or user only has the access necessary for its functions. For instance, a CI/CD pipeline service account might have permission to create or update deployments in a specific namespace but no right to view secrets or interact with other namespaces. By defining the roles, Kubernetes operators reduce the risk of accidental or malicious misuse of privileged operations.

7.2. Pod Security Policies

Beyond RBAC, enforcing security policies at the pod level is vital for maintaining a robust security posture. In newer Kubernetes releases, Pod Security Admission [29] (replacing the older Pod Security Policies) allows administrators to define baseline pod requirements—such as disallowing privileged containers, requiring a read-only root file system, or ensuring containers run as non-root users. For example, a policy might forbid containers from escalating privileges or mounting sensitive host paths containing the blast radius if an attacker compromises a container. Combining pod-level security policies with network policies (to restrict pod-to-pod communication) and RBAC helps build a layered defense strategy. For instance, a development namespace might enforce a more permissive policy to allow debugging, whereas a production namespace enforces a strict policy with no privileged pods.

Pod Security Admission (PSA) in Kubernetes is the newer mechanism for enforcing pod security constraints, replacing the older Pod Security Policies (PSPs). Unlike PSPs, which rely on cluster-wide resources and could be cumbersome to configure, PSA defines standardized enforcement levels (Privileged, Baseline, and Restricted) and uses namespace labels and built-in admission checks, making it more straightforward and more consistent to apply security controls. For example, a "Restricted" level may prohibit privilege escalation or running as root, while a "Baseline" level might allow minimal privileges needed for typical applications. This declarative, namespace-based approach streamlines the security posture across clusters by ensuring that pods deployed in each namespace cannot exceed the defined security level. In addition, PSA integrates more seamlessly into the Kubernetes admission chain, improving compatibility with other security features and providing a clear audit trail when pods are denied or modified. The benefits include a more straightforward configuration model, reduced complexity in administering pod security, and better alignment with modern best practices for Kubernetes security.

7.3. Network Policies

In Kubernetes, network policies [24] define how pods can communicate with each other and external endpoints, effectively implementing a firewall-like mechanism at the pod level. By default, many Kubernetes setups allow all pods to talk to each other without restriction; network policies enable administrators to lock this down and adopt a more fine-grained, zero-trust approach. For example, a network policy might allow incoming traffic to a "backend" pod only from pods labeled "frontend," blocking all other traffic. Another policy could disallow egress from certain pods to external networks, restricting a sensitive service's internet access. This capability is critical for a robust Kubernetes security posture, as it prevents lateral movement by attackers who may compromise one pod and attempt to pivot to other pods or services. Network policies integrate seamlessly with the underlying Container Network Interface (CNI) plugins (like Calico or Cilium [30]), which enforce these rules at the

© 2025 by the authors; licensee Learning Gate

network layer. Network policies help IT professionals isolate workloads, enforce the least-privilege communication, and reduce the attack surface in dynamic, multi-tenant Kubernetes clusters.

7.4. The Role of Service Meshes in Security

Service meshes like Istio [31] and Linkerd [32] enhance container security in Kubernetes clusters by providing a transparent layer for managing and securing inter-service communication.

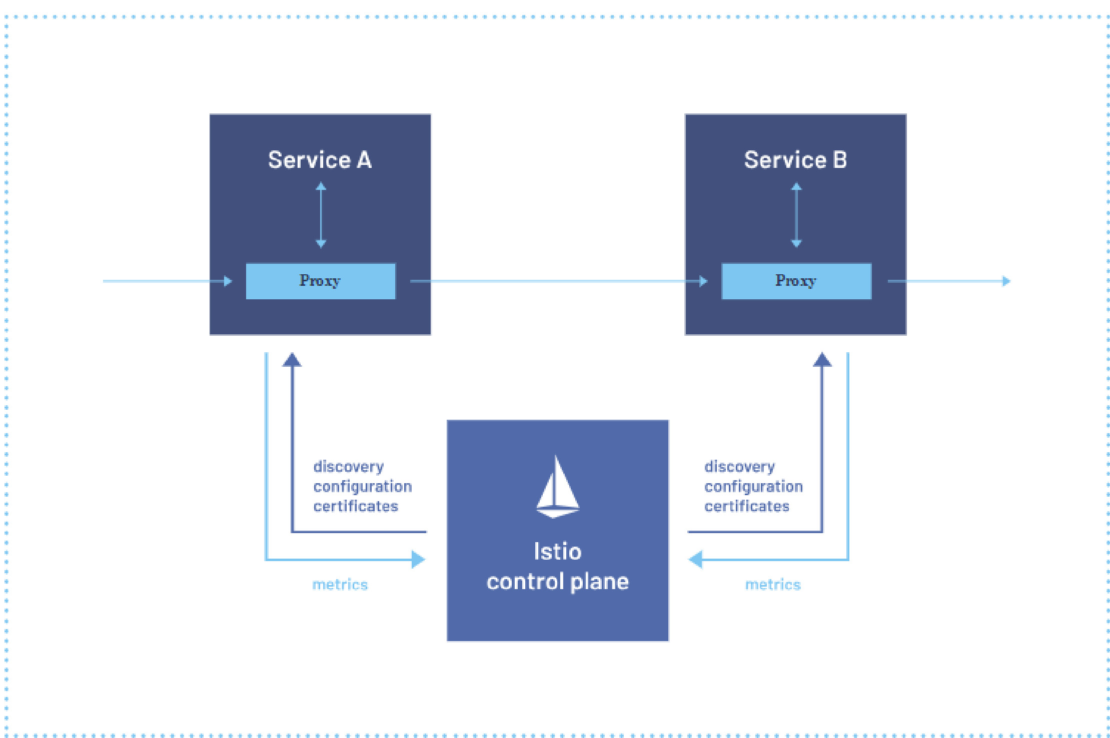

Figure 6.
The Istio service Mesh (from The Istio Authors [30]).

They work through sidecar proxies (e.g., Envoy or linkerd-proxy) injected into each pod, automatically encrypting traffic and enforcing policies without requiring modification to the application code. One core feature is mutual TLS (MLS), which ensures both ends of a connection authenticate each other, and all data in transit is encrypted, significantly reducing the risk of eavesdropping or tampering. Furthermore, service meshes enable fine-grained policy enforcement, allowing only specific services to communicate or setting rate limits to mitigate Denial of Service (DoS) attacks. They also offer powerful observability features by collecting metrics, logs, and traces at the proxy layer, giving IT professionals deep insights into traffic flows, performance bottlenecks, and potential security issues. For example, Istio's policy engine allows custom authorization rules based on attributes like source, destination, and request method. At the same time, Linkerd's "automatic TLS" can secure inter-pod traffic even if applications are not explicitly configured for TLS. By centralizing and automating these capabilities, service meshes streamline the adoption of best practices and bolster the overall security posture of containerized microservices environments.

© 2025 by the authors; licensee Learning Gate

8. Security Best Practices in Container Development

Best practice methodologies for container development are crucial for constructing secure, dependable, and efficient containerized applications. Containers facilitate development and deployment; nonetheless, their dynamic characteristics require meticulous consideration of security and operating measures. A crucial approach is shift-left security, which involves including security measures early in the development process to detect and mitigate vulnerabilities before release. Automated testing and scanning are similarly essential, facilitating ongoing surveillance of container images and configurations for security vulnerabilities and compliance breaches. Consistent patch management ensures that base images and dependencies remain current, mitigating exploitation risk. Furthermore, implementing immutable infrastructure—where containers are substituted rather than altered—guarantees uniformity reduces drift, and streamlines troubleshooting. By adhering to these guidelines, developers can construct resilient, scalable, and secure containerized applications that conform to contemporary DevOps procedures.

8.1. Shift-Left Security

Shift-left security [33] in container development emphasizes embedding security checks and measures as early as possible in the software development lifecycle (SDLC) rather than forcing them on at the end. This approach, often called DevSecOps, aims to detect and fix vulnerabilities before they become production. For instance, developers can incorporate container image scanning (e.g., using Clair [21]; Trivy [34] or Snyk [20]) directly into their CI/CD pipelines. When a developer commits code, automated jobs run to check for known CVEs, outdated libraries, or insecure configurations in the image. If any issues are found, the pipeline can automatically fail or notify the relevant team, prompting quick remediation. This proactive strategy reduces the time and cost of patching vulnerabilities late in the development cycle.

To implement shift-left security, IT professionals can define policy-as-code rules that govern everything from allowed base images to required security tests. A CI/CD platform (e.g., Jenkins [35] GitLab CI [36] GitHub Actions [37]) then enforces these rules at each stage of the build process. For example, if an image fails a vulnerability scan or any dependencies are out of compliance, the pipeline will stop, preventing the insecure container from progressing to later stages. Integration with container registries also helps automate the promotion or rollback of images based on scan results. Moreover, hooking into developer tools like IDE plugins or pre-commit hooks can catch basic misconfigurations (e.g., API keys or passwords present in plain text) before they even hit the repository. Shift-left security ensures containers remain secure, efficient, and compliant throughout their lifecycle by embedding security controls at every step- from code commit to production deployment.

8.2. Automated Testing and Scanning

Automated testing and scanning for container security include three main parts: vulnerability scanning, static analysis, and runtime monitoring. Vulnerability scanning tools (e.g., Trivy [34] and Clair [21] or Anchore [38]) can be integrated into CI/CD pipelines to identify known CVEs or outdated dependencies in container images as soon as they are built. Static analysis tools further enhance this process by examining Dockerfiles, Kubernetes manifests, and other configuration files to detect insecure defaults—such as running containers with root privileges, exposing unnecessary ports, or storing secrets in plain text. These checks can be enforced automatically, preventing misconfigured or vulnerable images from passing critical checks in the build pipeline.

Runtime monitoring provides continuous inspection of containers in staging or production environments. Security-oriented agents or kernel modules, such as Falco [39] Sysdig [40] or even Linux kernel security modules like AppArmor or SELinux, observe real-time container behavior. They watch for anomalous patterns, such as suspicious system calls, privilege escalations, or unexpected

© 2025 by the authors; licensee Learning Gate

network connections, that might indicate an ongoing compromise. For instance, Falco can be configured to alert if a running container suddenly spawns a shell, which could indicate malicious activity.

8.3. Regular Patch Management

Regular patch management in containerized environments is a significant part of container security because multiple software layers, including the base image, application dependencies, and the host operating system, must all remain up to date to prevent known vulnerabilities from being exploited. Containers often inherit layers from base images, meaning if the base image contains outdated libraries or packages with security flaws, every image derived from it will inherit those vulnerabilities. For example, if you use a popular base image from a public repository and a critical CVE is discovered in one of its core libraries, all downstream images derived from that base image are suddenly at risk. To address this, developers should implement automated scanning and rebuild processes in their CI/CD pipelines. Additionally, setting up notifications for critical patches and incorporating "pull and rebuild" policies can ensure that new, patched versions of base images are promptly adopted and even automatically pushed to the testing environment.

8.4. Immutable Infrastructure

Immutable infrastructure in the context of containers refers to treating application environments as disposable, pre-built artifacts rather than mutable systems that are updated in place. Instead of logging into a server to apply patches or modify application configurations, the entire container image is rebuilt with the necessary changes and redeployed. This practice eliminates "configuration drift" issues, where small, manual changes lead to an inconsistent state over time. For example, suppose you discover a vulnerability in a container image rather than patching it in place on a running container. In that case, you rebuild the image (applying the patch), push it to a registry, and redeploy containers from that new, updated image. This approach ensures that every application instance runs a known, validated configuration, reducing the chance of unpredictable behavior and simplifying troubleshooting and auditing.

From an operational standpoint, immutable deployments align well with DevOps and microservices practices. Tools like Kubernetes make rolling updates straightforward: when a new container image version is available, the orchestrator gradually replaces old pods with new ones, verifying health checks along the way. This approach keeps environments consistent and simplifies rollback if something goes wrong—revert to a previous container image version. For developers, it promotes reproducibility and accelerates testing cycles because the same container image is used in development, and QA is the one that runs in production. By combining infrastructure automation with immutable container images, teams can maintain a predictable, resilient deployment pipeline where new features or security patches are confidently rolled out without the risk of unpredictable, manual configuration changes.

9. Compliance and Regulations

Industries such as finance and healthcare operate under strict regulatory requirements (e.g., PCI DSS for payment processing, HIPAA for healthcare data) that dictate rigorous security and auditing requirements that apply similarly to containerized environments. In container deployments, compliance often starts with standard hardening practices and documented security baselines (for example, CIS Benchmarks for Docker [40] or Kubernetes [41]). Organizations can enforce these standards using built-in Kubernetes admission controllers, Pod Security Admission levels, or third-party policy engines (like OPA/Gatekeeper [42]), which ensures that only compliant and securely configured containers can run. Additionally, encrypting data in transit (via mutual TLS) and at rest, employing strict network segmentation, and regularly scanning container images can help organizations maintain compliance. Tools like Falco or Kubescape [43] can provide real-time threat detection in line with governance rules, generating alerts whenever suspicious activity is detected. By integrating these measures into a CI/CD

© 2025 by the authors; licensee Learning Gate

pipeline, companies ensure that every container entering production meets the regulatory guidelines, reducing the risk of costly violations and data breaches.

Logging, auditing, and monitoring for containerized applications differ partly due to containers' ephemeral nature and microservices' distributed nature. Traditional logging might store data on the same host as the application. Still, since containers can be spun up or down quickly, they can potentially lose local logs unless shipped to a centralized logging system (e.g., Elasticsearch, Splunk, or a cloud-based logging service). Furthermore, logs and monitoring data may need to be aggregated in a container ecosystem from multiple pods, nodes, and services, necessitating a robust observability stack (e.g., Prometheus, Grafana, or Datadog). This data becomes essential for incident response, as security teams must be able to quickly reconstruct the timeline of events, track suspicious activity across services, and contain the threat before it spreads laterally through the cluster.

Incident response for containers should include predefined playbooks that detail how to isolate compromised pods (e.g., through Kubernetes Network Policies or by cordoning off nodes), rotate credentials (like Kubernetes Secrets), and redeploy with patched or known-good images. Well-orchestrated responses are essential for meeting regulatory demands in containerized environments.

10. Future Trends in Container Security

Zero-trust architecture in containerized environments extends the principle that no network component or request—internal or external—should be automatically trusted. Traditionally, once inside a trusted network perimeter, systems might communicate with minimal authentication or encryption. In contrast, a zero-trust network requires continuous authentication, authorization, and encryption for every interaction. In container ecosystems, all traffic between containers, microservices, and external endpoints should be treated skeptically and only permitted if it passes strict validation policies. For example, two services might communicate via mutual TLS to verify the other's identity before exchanging data. Strong role-based access controls are also applied to ensure that only the correct containers or service accounts can initiate communication, aligning with the least privilege principle. Zero-trust architecture in Azure is a perfect example of this approach [44] as it offers a comprehensive framework for implementing the least-privilege tenets and automated responses to incoming threats.

This concept is particularly relevant in container and microservices architectures, where multiple lightweight services communicate frequently over dynamic, ephemeral networks. By implementing zero-trust, organizations can create robust micro-segmentation: each service or container runs in an isolated environment and only accepts traffic from authorized sources. Kubernetes Network Policies or service mesh solutions (e.g., Istio, Linkerd) can enforce these rules, denying all other traffic by default. For instance, a frontend service might only be permitted to speak to a backend service on a specific port, and all traffic is encrypted and authenticated at the proxy layer. This level of control both limits the blast radius of a potential breach and ensures regulatory compliance—every request or connection attempt is logged, monitored, and verified.

Serverless computing abstracts away the notion of managing servers, focusing instead on running code in response to events or HTTP requests. While some implementations of serverless (like AWS Lambda [45] or Azure Functions [46]) appear not to involve containers to the end user, most platforms under the hood use container-like environments or sandboxed runtimes to isolate individual functions. This approach eliminates much of the infrastructure provisioning and patch management responsibilities from developers; however, it also centralizes these tasks within the service provider's platform. From a security standpoint, organizations must carefully understand how the serverless provider handles runtime isolation, updating of underlying container images or runtimes, and the data flow between serverless functions and other resources. For example, a function with misconfigured permissions in AWS Lambda could inadvertently grant attackers access to a broader set of resources if its AWS Identity and Access Management (IAM) roles are overly permissive.

© 2025 by the authors; licensee Learning Gate

Despite the convenience of being serverless, there are many security challenges. Functions are often event-driven and short-lived, complicating logging and incident response, as forensics data might vanish quickly. When functions run in containers (as in specific “serverless containers” offerings like AWS Fargate [47] or Google Cloud Run [48]), organizations inherit many of the same concerns as traditional container security, such as image vulnerabilities, runtime isolation, and network segmentation—albeit managed or partially managed by the cloud provider. Implementing strict access policies (e.g., the principle of least privilege), scanning container images for serverless deployments, and adopting strong observability practices with centralized logging and real-time alerting can help mitigate these new risks.

AI and machine learning (AI/ML) are increasingly playing a transformative role in container security, allowing organizations to detect and prevent breaches by analyzing large datasets and identifying patterns indicative of malicious behavior. Traditional security solutions rely heavily on signature-based detection and known threat intelligence, which can adapt slowly to zero-day exploits or new attack vectors. AI/ML-driven solutions, on the other hand, use anomaly detection algorithms or behavior-based models to learn what “normal” looks like for containerized applications—such as expected process trees, network flows, and resource usage. When deviations from these learned baselines occur, the system can issue alerts, trigger automated responses, or quarantine suspicious containers. Tools like Aqua Security [49] and Palo Alto Networks Prisma Cloud [50] increasingly integrate AI-powered functionality, scanning container images more efficiently for hidden malware and tracking behavior in real-time to spot irregularities that signature-based methods may miss. Depending on the environment, tools like Microsoft Sentinel and Logic Apps might be used to improve container security [50] if containers are running in Microsoft Azure.

Another use case of AI/ML in container security is predictive analytics that can prioritize and remediate vulnerabilities. For example, when vulnerability scanners feed data into an AI model, the system can cross-reference findings with exploit databases, code version histories, and system metadata to assess the likelihood of exploiting vulnerability. This helps security and DevOps teams focus on the most critical issues, improving response times and resource allocation. Moreover, AI can help correlate multiple low-level indicators—such as elevated memory usage in one container, unexpected network connections in another, and sudden spikes in CPU across a third—to detect a coordinated attack campaign. By continuously learning from real-world container usage and threat data, AI-powered solutions become increasingly “smarter” at spotting known and unknown threats while strengthening an organization’s resilience against security breaches.

11. Conclusions

Container security is essential to modern application development, especially given the increasing adoption of microservices architectures and DevOps practices. This paper highlights that containers provide unparalleled portability, efficiency, and scalability, enabling rapid deployment and seamless operation across varied environments. However, these advantages come with unique challenges, including vulnerabilities associated with shared kernel architectures, runtime risks, and the orchestration layer complexities. Addressing these risks requires a comprehensive strategy incorporating robust security practices, such as regular vulnerability scanning, secure configuration, runtime monitoring, and implementing access controls like RBAC. Adequate container security ensures the integrity and reliability of applications and enhances trust in sensitive data management and operational consistency.

Looking ahead, the evolution of container security will likely be shaped by advancements in automation, AI-driven threat detection, and a shift toward zero-trust architectures. These trends aim to proactively address vulnerabilities while streamlining the integration of security measures throughout the development lifecycle. Furthermore, the emphasis on compliance, immutability, and innovative technologies like service meshes underscores the need for organizations to remain ever-vigilant. By

© 2025 by the authors; licensee Learning Gate

prioritizing security as a core element of container adoption, businesses can harness the full potential of containerization while minimizing risks, ensuring their applications are resilient, secure, and future-ready in a rapidly evolving technological landscape.

Transparency:

The authors confirm that the manuscript is an honest, accurate, and transparent account of the study; that no vital features of the study have been omitted; and that any discrepancies from the study as planned have been explained. This study followed all ethical practices during writing.

Copyright:

© 2025 by the authors. This open-access article is distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

References

[1] S. Susnjara and I. Smalley, "What is containerization?," Retrieved: https://www.ibm.com/think/topics/containerization. [Accessed 20 May 2024], 2024.
[2] IBM Cloud Team, "Containers versus virtual machines (VMs): What's the difference?, IBM," Retrieved: https://www.ibm.com/think/topics/containers-vs-vms. [Accessed 2021.
[3] Docker Inc, *What is Docker?* Docker Inc. https://docs.docker.com/get-started/docker-overview/, n.d.
[4] The Kubernetes Authors, "Overview," Retrieved: https://kubernetes.io/docs/concepts/overview/. [Accessed 2024.
[5] M. Michalowski, "Top 47 devops statistics 2025: Growth, benefits, and trends, Spacelift," Retrieved: https://spacelift.io/blog/devops-statistics. [Accessed 2025.
[6] Red Hat, "Kubernetes adoption, security, and market trends report 2024, Red Hat," Retrieved: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview. [Accessed 2024.
[7] Check Point Software Technologies, "Top 7 container security issues," Retrieved: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-container-security/top-7-container-security-issues/. [Accessed n.d.
[8] Red Hat, "What is a CI/CD pipeline?, Red Hat," Retrieved: https://www.redhat.com/en/topics/devops/what-cicd-pipeline. [Accessed 2022.
[9] Docker Inc, "Dockerfile reference," Retrieved: https://docs.docker.com/reference/dockerfile/. [Accessed n.d.
[10] Docker Inc, "Docker Hub," Retrieved: https://hub.docker.com. [Accessed n.d.
[11] The Kubernetes Authors, "Autoscaling workloads," Retrieved: https://kubernetes.io/docs/concepts/workloads/autoscaling/. [Accessed 2024.
[12] B. Ben-Michael and Y. Yaakov, "Container breakouts: Escape techniques in cloud environments, Palo Alto Networks," Retrieved: https://unit42.paloaltonetworks.com/container-escape-techniques/. [Accessed 2024.
[13] The Kubernetes Authors, "Secrets," Retrieved: https://kubernetes.io/docs/concepts/configuration/secret/. [Accessed 2024.
[14] A. Wallace and C. Baer, "Exploring container security: Performing forensics on your GKE environment, Google," Retrieved: https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-performing-forensics-on-containers. [Accessed 2019.
[15] Red Hat, "Docker SELinux security policy," Retrieved: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/docker_selinux_security_policy. [Accessed n.d.
[16] AppArmor, "AppArmor Wiki," Retrieved: https://gitlab.com/apparmor/apparmor/-/wikis/home. [Accessed n.d.
[17] Microsoft, "What is DevSecOps?," Retrieved: https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops. [Accessed n.d.
[18] Docker Inc, "Seccomp security profiles for Docker," Retrieved: https://docs.docker.com/engine/security/seccomp/. [Accessed n.d.
[19] D. Warburton, "What is mTLS? F5," Retrieved: https://www.f5.com/labs/learning-center/what-is-mtls. [Accessed 2021.
[20] Snyk, "Snyk Container," Retrieved: https://snyk.io/product/container-vulnerability-management/. [Accessed n.d.
[21] Clair, "Clair is an open source project for the static analysis of vulnerabilities," Retrieved: https://github.com/quay/clair. [Accessed n.d.
[22] Podman, Retrieved: https://podman.io. [Accessed n.d.
[23] CRI-O, "Lightweight container runtime for Kubernetes," Retrieved: https://cri-o.io. [Accessed n.d.

© 2025 by the authors; licensee Learning Gate

[24] HashiCorp, "Manage secrets and protect sensitive data with Vault," Retrieved: https://www.vaultproject.io. [Accessed n.d.
[25] Canonical, "Linux kernel livepatch," Retrieved: https://ubuntu.com/security/livepatch/docs. [Accessed n.d.
[26] Fedora, "Fedora CoreOS documentation," Retrieved: https://docs.fedoraproject.org/en-US/fedora-coreos/. [Accessed n.d.
[27] The Kubernetes Authors, "Using RBAC authorization," Retrieved: https://kubernetes.io/docs/reference/access-authn-authz/rbac/. [Accessed 2024.
[28] The Kubernetes Authors, "Pod security admission," n.d. [Online]. Available: https://kubernetes.io/docs/concepts/security/pod-security-admission/
[29] V. Dakić, J. Redžepagić, M. Bašić, and L. Žgrablić, "Performance and latency efficiency evaluation of kubernetes container network interfaces for built-in and custom tuned profiles," *Electronics*, vol. 13, no. 19, p. 3972, 2024. https://doi.org/10.3390/electronics13193972
[30] The Istio Authors, "The Istio service mesh," Retrieved: https://istio.io/latest/about/service-mesh/. [Accessed 2024.
[31] The Linkerd Authors, "The world's most advanced service mesh," Retrieved: https://linkerd.io. [Accessed 2025.
[32] M. Nair, "Implementing shift left security effectively, Snyk," Retrieved: https://snyk.io/articles/shift-left-security/. [Accessed n.d.
[33] Aqua, "The all-in-one open source security scanner," Retrieved: https://trivy.dev/latest/. [Accessed n.d.
[34] Jenkins, "The leading open-source automation server," Retrieved: https://www.jenkins.io. [Accessed n.d.
[35] GitLab, "Get started with GitLab CI/CD," Retrieved: https://docs.gitlab.com/ee/ci/index.html. [Accessed n.d.
[36] GitHub Inc, "Automate your workflow from idea to production," Retrieved: https://github.com/features/actions. [Accessed n.d.
[37] Anchore, "Developer-friendly scanning tools," Retrieved: https://anchore.com/opensource/. [Accessed n.d.
[38] Sysdig, "Detect security threats in real time," Retrieved: https://falco.org. [Accessed n.d.
[39] Sysdig, "The sysdig platform," Retrieved: https://sysdig.com/products/platform/. [Accessed n.d.
[40] Center for Internet Security, "CIS Docker benchmarks," Retrieved: https://www.cisecurity.org/benchmark/docker. [Accessed n.d.
[41] Community, "A customizable cloud-native policy controller," Retrieved: https://open-policy-agent.github.io/gatekeeper/website/. [Accessed n.d.
[42] The Kubescape Authors, "Comprehensive kubernetes security from development to runtime," Retrieved: https://kubescape.io. [Accessed 2024.
[43] D. Vedran, M. Zlatan, K. Ana, and R. Damir, "Analysis of azure zero trust architecture implementation for mid-size organizations," *Journal of Cybersecurity and Privacy*, vol. 5, no. 1, p. 2, 2025. https://doi.org/10.3390/jcp5010002
[44] Amazon, "Run code without thinking about servers or clusters," Retrieved: https://aws.amazon.com/lambda/. [Accessed n.d.
[45] Microsoft, "Azure functions overview," Retrieved: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview. [Accessed n.d.
[46] Amazon, "AWS Fargate," Retrieved: https://aws.amazon.com/fargate/. [Accessed n.d.
[47] Google, "Cloud run," Retrieved: https://cloud.google.com/run. [Accessed n.d.
[48] Aqua, "Unified cloud security," Retrieved: https://www.aquasec.com. [Accessed n.d.
[49] Palo Alto Networks, "Prisma cloud," Retrieved: https://www.paloaltonetworks.com/prisma/cloud. [Accessed n.d.
[50] V. Dakić, Z. Morić, A. Kapulica, and D. Regvart, "Leveraging microsoft sentinel and logic apps for automated cyber threat response," *Edelweiss Applied Science and Technology*, vol. 8, no. 6, pp. 4319–4348, 2024. https://doi.org/10.55214/25768484.v8i6.2933

© 2025 by the authors; licensee Learning Gate