Application Security Optimization in Container Orchestration Systems Through Strategic Scheduler Decisions

Yevhenii Voievodin[1] and Inna Rozlomii[1]

[1] *Bohdan Khmelnytsky National University of Cherkasy, 81, Shevchenko Blvd., Cherkasy, 18031, Ukraine*

Abstract
This paper explores the security issues found in Container Orchestration Systems (COS) like Kubernetes, Docker Swarm, and Apache Mesos, which are crucial for running modern cloud-based applications. Although COS makes it easier to deploy and manage applications, it also brings a set of security challenges. This paper discusses common security risks in COS, including unsafe configurations, threats to the supply chain, and setup mistakes that could let attackers gain unauthorized access or disrupt the system. The paper examines the responsibilities of different parts of COS. It points out the dangers of giving too many permissions and not having strong network security measures. The discussion covers how containers and the process of managing them should be securely set up to prevent vulnerabilities. Additionally, the paper looks at the security of microservices, a way of designing applications as a set of small services. It talks about how to safely deploy these services, manage their communication, and secure them using various tools and standards. A major focus of the paper is on how COS decides where to run containers, known as scheduling strategies, and how these can affect security. It reviews different strategies and proposes new ones that focus on the importance of each part of an application. By spreading out important parts across different servers, the system can be made more secure. The paper suggests ways to arrange containers in a way to reduces the chance of a widespread attack if one part gets compromised. In summary, this paper dives into the security aspects of COS, presenting a detailed look at the risks involved and offering guidance on how to secure these systems. It emphasizes the need for careful setup, constant monitoring, and smart strategies to place containers, aiming to protect against the ever-changing security threats in the world of cloud computing.

Keywords
Container orchestration system, Kubernetes, Docker, Docker Swarm, microservices, security.

1. Introduction

COS plays a crucial role in modern cloud-based software ecosystems [1, 2]. The fundamental concept employed by COS is a container, which is a lightweight package of software bundled with all necessary dependencies required for running the software. Utilizing containers significantly streamlines the provisioning of hardware and ensures reproducible application behavior across various platforms and operating systems. As containers encapsulate all dependencies, they eliminate the risk of conflicts or issues with outdated versions. This contributes to the speed of cloud application development and facilitates the continuous delivery [3] process. Often used examples of COS include Kubernetes, Docker Swarm, and Apache Mesos [4].

The popularity of COS is related to the rise of applications that utilize a microservices architecture [5, 6]. However, the transition to COS and microservices is not without its challenges. Organizations must navigate the

CPITS-2024: Cybersecurity Providing in Information and Telecommunication Systems, February 28, 2024, Kyiv, Ukraine
EMAIL: yevhenii.voievodin@vu.cdu.edu.ua (Y. Voievodin); inna-roz@ukr.net (I. Rozlomii)
ORCID: 0000-0002-6415-8566 (Y. Voievodin); 0000-0001-5065-9004 (I. Rozlomii)

complexities of container management, orchestration, and security, which demand a deep understanding of the underlying infrastructure and a careful approach to design and configuration. The shift also necessitates a cultural change, creating an environment of continuous learning, collaboration, and innovation among teams. As businesses increasingly rely on these systems, ensuring the security, reliability, and performance of COS becomes important. This includes not only protecting against external threats but also managing internal risks such as service dependencies, network configurations, and data management practices.

Microservices architecture primarily facilitates faster software delivery [7] by creating bounded contexts or areas of responsibility managed by different teams. Using microservices offers additional advantages. One of these is the containment of application failure scope or "blast radius" [8]. Unlike monolithic applications, those divided into separately deployable pieces are less prone to complete failure due to a single subsystem's issue, as these are independent applications and processes. Scalability is also an essential aspect [9]. The separation into distinct functions provides better granularity and scalability options. COS can dynamically allocate resources to different functions based on the load, optimizing the system's scalability.

While microservices architecture with the use of COS feature allows all these benefits, the application architecture is also important, without proper separation of responsibilities and functions proper scalability, resilience, and delivery speed might be hard to achieve.

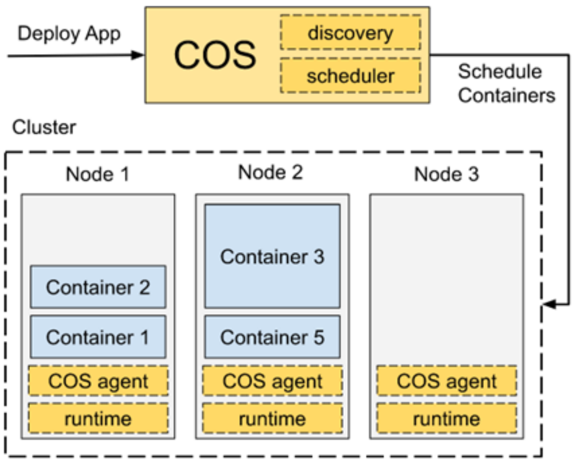

Figure 1: Key components of container orchestration system

Fig. 1 illustrates the key components utilized by COS:

1. **Cluster**: This represents a set of nodes, indicating the state of available hardware or virtual resources.
2. **Nodes**: These are individual physical or virtual units where containers can be deployed.
3. **Container Runtime**: This is the technology used to run containers, such as Docker [10]. Every node must have a container runtime installed to facilitate container operations.
4. **COS Agent**: Typically, this is a program that operates within a node and primarily functions as the communication point to manage operations on that node. While the container runtime focuses on managing the containers themselves, the COS agent delegates such requests to the container runtime.
5. **Discovery Component**: This component is responsible for tracking all the containers and associated applications. It facilitates internal application communication by serving as a central configuration point and may be utilized by a load balancer.
6. **Scheduler**: Playing a crucial role, the scheduler's primary function is to identify a suitable node for container deployment. Selecting a deployment strategy involves analyzing the system's use case, typically balancing between efficient utilization of cluster resources and application resilience or fault tolerance. Common strategies include binpacking and spreading [11]. Binpacking aims to pack as many containers as possible into a single node to maximize resource utilization. Spreading focuses on distributing containers across the emptiest nodes to enhance application fault tolerance [12]. This approach allows for the deployment of multiple instances of the same application component across different nodes, thereby reducing the risk of simultaneous failures due to hardware or other issues.

The adoption of COS does not come without its drawbacks. While COS efficiently organizes the management of running systems, it also adds to the system’s complexity. This complexity is necessary to accommodate the diverse requirements of various applications. For example, Kubernetes alone offers more than ten different configurable abstractions to provide end-users with the needed flexibility. However, this complexity opens the door to potential issues, such as the risk of misconfiguration, which can lead to human errors and security vulnerabilities.

Furthermore, the process of packaging software into container images requires precise attention to ensure that the software is free from vulnerabilities. The architecture of the application significantly influences the overall security of the system. Important considerations include how application secrets are managed, who has access to them, and the level of encryption employed in the communication between container nodes.

The multifaceted nature of COS inherently makes security a more challenging aspect due to the increased number of entry points, exposed APIs, and the amplified potential for errors. This article aims to explore some of the threats associated with COS, with a particular focus on the implications raised from decisions made by the scheduler component, highlighting the critical nature of this aspect in the COS infrastructure.

2. Related Works

The OWASP vulnerabilities [13] provide a comprehensive list of the top ten security risks for Kubernetes-based applications, encompassing issues such as insecure configurations, supply chain vulnerabilities, overly permissive Role Based Access Control (RBAC) configurations, lack of centralized policy enforcement, inadequate logging and monitoring, broken authentication mechanisms, missing network segmentation controls, misconfigured cluster components, and outdated and vulnerable components. Martin and Hausenblas delve into these issues, categorizing threats and proposing defensive measures for various scenarios that apply to Kubernetes COS [14]. They also detail the roles and responsibilities of different Kubernetes components and highlight the importance of isolating the container runtime from the host operating system to increase security. Creane and Gupta extend the discussion on security beyond COS itself, addressing potential threats that may arise during the pre-deployment phase, such as when faulty images are built and deployed [15]. They advocate for robust monitoring techniques, alerting rules, machine learning-based systems, proper network configuration, and secure application exposure methods.

Ugale et al. focus on container vulnerabilities in cloud environments, proposing a security framework that conducts vulnerability scans at various levels [16]. Security challenges in systems reliant on containers are explored by Sultan et al. highlighting the need for enhanced vulnerability management [17]. Rice discusses fundamental security concepts crucial for protecting applications in containers, introducing the “blast radius” concept to limit the impact of threats [18]. Lopens et al. propose mitigating security risks by adapting the seccomp profile, which restricts container system calls to minimize the attack surface, emphasizing the development of these profiles in a fast and scalable manner [19]. Belair et al. focus on the security features of the Linux kernel that are employed at the virtualization boundary between the operating kernel and containers [20].

Dias and Siriwardena comprehensively address the mechanisms utilized in microservice security, including deployments, communication, API gateways, JWT tokens, and the OAuth 2 standard [21]. They delve into the deployment strategies using Kubernetes and discuss methods to ensure secure configurations of Docker or alternative container engines. Berardi et al. explore the significance of the human element in the security of applications employing a microservices architecture [22]. The challenges of microservices security, along with contemporary solutions to key issues, are presented by Driss et al. [23], providing a crucial resource for secure software researchers and practitioners by offering a comprehensive catalog of security solutions and mechanisms for applications based on microservices.

3. Security Threats

3.1. Gaining Access to COS Infrastructure

One common issue in the configuration of COS, as noted in the OWASP top ten security risks for Kubernetes [13], risk of insecure workload configurations. Running application services as the root user, when not necessary, grants the application excessive permissions, potentially enabling the execution of harmful operations such as initiating malicious processes. The use of a file system that allows write operations can inadvertently permit the installation of unauthorized software, leading to unexpected and possibly malicious container behavior. Moreover, the use of privileged containers grants additional kernel capabilities, which, when coupled with other misconfigurations, might pave the way to access the host (node) itself. Gaining control over the host opens up access to other containers on that node, the container runtime, and the COS agent, all of which could be manipulated for malicious purposes. Furthermore, overly permissive configurations of RBAC permissions can provide unintended access to containers, posing a significant security risk. Addressing these vulnerabilities requires strict adherence to security best practices, including minimizing permissions and ensuring robust access controls within the COS environment. Additionally, compromising the supply chain is another avenue through which containers can be made vulnerable. Exploiting the software architecture deployed within the COS system and introducing vulnerabilities can also provide attackers with potential points of exploitation.

Weak authentication mechanisms and absent network security controls can significantly broaden the scope of an attack, potentially extending it from a single container to an entire node, and from one node to others within the network. A lack of encryption within the cluster can enable an attacker who has gained access to a container to exploit network communications by intercepting traffic between containers. Moreover, permitting direct access between hosts could allow an attacker to exploit more critical APIs available within the cluster (Fig. 2).

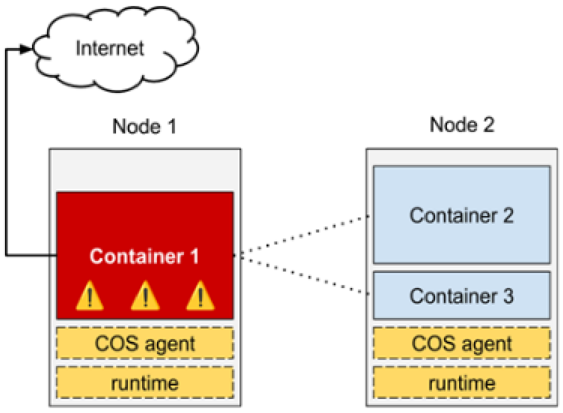

Figure 2: Example of cluster setup where Container 1 is compromised while connecting over the network to Container 2 and Container 3 on the separate node

The outlined approaches reveal that attack vectors typically start with minor threats within the system and progressively expand by accessing additional components. Although adhering to best practices in COS usage can significantly reduce the likelihood of such attacks, it's important to recognize that the risk can never be eliminated. These best practices involve the use of scanning tools and properly configured cloud environments to ensure that permissions are appropriately restrictive. Nonetheless, the specific requirements of individual applications and the urgency of deployment can sometimes lead to more lenient configurations, increasing the vulnerability to security breaches.

Leveraging the capabilities of scheduling strategies in COS can play a crucial role in mitigating the impact of attacks. The scheduler, responsible for deciding which node will host a new set of containers, inherently influences the distribution of containers across the cluster. This distribution, or locality, can be an important factor in system security.

Recent research indicates that scheduling strategies can be aligned with multiple objectives, such as resilience to failures, efficient resource utilization, ensuring application accessibility, and rapid deployment speeds. Security considerations can also be integrated into these objectives. For instance, the binpacking strategy prioritizes nodes that are already heavily loaded, optimizing resource usage. However, this strategy also concentrates on the components of an application, potentially increasing its vulnerability to localized attacks.

Conversely, strategies that spread application components across different nodes can enhance security. If the components are distributed, gaining unauthorized access to one node does not automatically compromise the entire system, as each node operates independently. This approach can make it considerably more challenging for an attacker to inflict widespread damage, underscoring the importance of container distribution in increasing the overall security posture of the COS environment.

The following method aims to minimize the impact of attacks and involves labeling application services according to their importance level. The strategy could distribute application containers such that containers with the same importance level are not deployed on the same node. Essentially, this increases the likelihood that containers of similar importance will be placed on different nodes. Strategies like spreading or usage of affinity controls in Kubernetes already help in distributing instances of the same application across different nodes for enhanced fault tolerance. Considering the importance level of services adds an extra layer of security. This approach effectively reduces the risk of an attacker compromising critical components by gaining access to a limited part of the system. The equation to compute the weight of the node, where weight can be used in the node selection process is the following:

$$W_i = \sum_{k=1}^{m} \frac{1}{R_k} \tag{1}$$

where W_i is the weight of a particular node in the cluster, m is the number of container instances running within a node, R is the rank of the container, the highest rank is 1, and the lowest rank is defined by the system complexity.

The index of a desired node can be found by the following formula.

$$K = \underset{i}{\operatorname{argmin}}\, Wi \tag{2}$$

where K is the desired node, the node having minimal weight, and W_i is the weight of the node with index i within the cluster. A node that for implementation purposes instead of using *min* operator it might be more convenient to sort the node weight values in ascending order, which would allow to pick the next node after the first one, in case the first one is full or does not meet other conditions of the strategy.

3.2. Exploiting Publicly Available Containers

Another risk involves the direct exploitation of vulnerabilities in the application architecture. Consider a scenario where an attacker targets a less important component of the application, possibly because it is less monitored or more prone to oversight. The attacker generates an excessive load on this service. If the container's resource limits are not properly configured, it could consume an increasing share of the node's resources, leading to performance degradation of more critical application components on the same node.

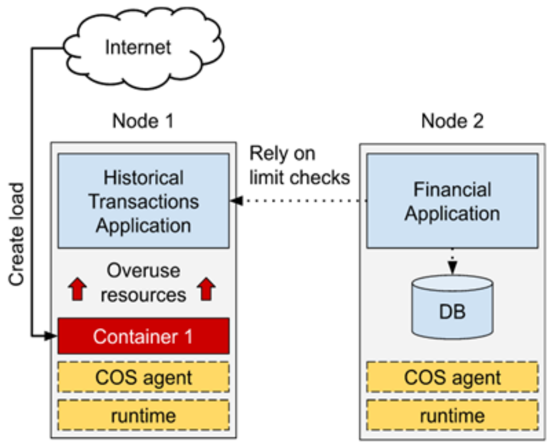

Figure 3. Example of cluster setup Container 1 is overloaded from the public internet, which affects other containers within the node

For example, consider a financial application deployed on one node, while an application handling historical transactions is on another node alongside a less important service, depicted as container 1 in Fig. 3. Although the historical transactions application is not accessible from the public internet and is only used internally within the cluster, overloading container 1 can indirectly impact this service, affecting other applications reliant on it. For instance, if the financial application has a fallback mechanism that relies on a local data copy when the historical transactions service is slow or unresponsive, an attacker, aware of this architecture, might overload container 1. This overload could trigger the fallback mechanism, potentially allowing the attacker to bypass certain controls or limits within the financial application. While this architecture prioritizes availability over consistency (in terms of CAP theorem [24]), which might be contentious for such applications, the issue persists in other cases where systems favor availability.

This scenario introduces an additional potential objective for scheduling strategies: separating publicly accessible applications from those accessible only within the cluster. This separation aims to prevent situations where an attacker might indirectly overload an internal application by exploiting an externally accessible service. Implementing this measure through scheduling strategies can enhance the system's security. The following formula represents the computation of a node weight that along with container ranks also takes into account the public exposure factor:

$$W_i = X\left(\sum_{k=1}^{m} \frac{1}{R_k}\right) + Y\left(\sum_{k=1}^{m} A_k\right) \qquad (3)$$

where W_i is the weight of the node within a cluster, m is the number of container instances running within a node, R is the rank of the container within the node, A is 1 in case the container is publicly available and 0 if it's not, X is the coefficient used to control the impact of containers rank, Y is the coefficient used to control the impact of containers availability.

System malfunction might be triggered not only by direct the external load but also due to time events, such as salaries coming out at the beginning of the month or high load during the import of bank statements.

Additionally, it's crucial to configure resource limits properly for each container, ensuring that there's a defined threshold for resource usage. Alongside setting these limits, robust system monitoring with alerting rules is essential. Prometheus and Grafana are tools that can be used for such systems, providing great flexibility and feature sets, including anomaly detection and alerting [25]. Implementing autoscaling can further improve the system's resilience, enabling it to handle normal usage growth and providing indicators for potential attack attempts. These measures collectively ensure that the application not only remains operational under typical conditions but also offers a level of protection against such attack scenarios.

4. Discussions

The exploration into the impact of scheduler decisions on application security within COS reveals a complex relation between system configurations, scheduling strategies, and security outcomes. This discussion has underscored the multifaceted nature of security within COS, particularly focusing on Kubernetes. Vulnerabilities can arise from various sources, including insecure workload configurations, supply chain risks, and the complex architecture of microservices.

The consideration of service importance and the strategic placement of containers to avoid co-locating critical components on the same node are significant steps toward minimizing the attack surface. However, this strategy is not without challenges. The dynamic nature of COS and the continuous evolution of threats necessitate a proactive and adaptable approach to security. This involves not only intelligent scheduling but also robust monitoring, precise configuration management, and the implementation of responsive mechanisms such as autoscaling to address any anomalies promptly. Suggested methods for weighting nodes, embedded into existing strategies, present a challenge: deciding which is more important—resource consumption or reduced risk.

The insights from related works in the domains of COS security, container vulnerabilities, and microservices security provide a rich context for understanding these challenges. These studies highlight the need for a holistic approach to security that includes all phases of the application lifecycle, from development and deployment to runtime management.

5. Conclusions

This paper has extensively discussed the security intricacies and challenges associated with COS like Kubernetes, Docker Swarm, and Apache Mesos, which play key roles in cloud-based software ecosystems. While COS offers substantial benefits in terms of application deployment, scalability, and management, it also introduces a significant amount of security vulnerabilities and risks. The exploration of COS security highlighted that issues such as insecure workload configurations, supply chain vulnerabilities, and inappropriate permission settings can significantly reduce the integrity and safety of the entire system.

Through an in-depth analysis of related works, the paper emphasized the necessity of using strong security measures, including vulnerability management, robust configuration practices, and the implementation of necessary security frameworks. The role of COS components, such as container runtimes, COS agents, and schedulers, was covered, illustrating how their proper configuration and management are related to maintaining a secure orchestration environment.

Moreover, the paper underscored the critical influence of scheduling strategies on the security of COS. By proposing scheduling strategies that consider the importance level of application services and ensure a strategic distribution of containers across nodes, the research advocated for a proactive stance in mitigating the potential impact of attacks. The nuanced approach to container placement, considering factors like the importance of services and public exposure, provides a mechanism for enhancing system resilience and security.

References

[1] A. Prajapati, D. Patel, Container Scheduling: A Taxonomy, Open Issues and Future Directions for Scheduling of Containerized Microservices, SSRN (2024).

[2] O. Oleghe, Container Placement and Migration in Edge Computing: Concept and Scheduling Models, IEEE Access 9 (2021) 68028–68043. doi: 10.1109/ ACCESS.2021.3077550.

[3] A. Sabau, S. Hacks, A. Steffens, Implementation of a Continuous Delivery Pipeline for Enterprise Architecture Model Evolution, Softw. Syst. Model. 20 (2021) 117–145. doi: 10.1007/s10270-020-00828-z.

[4] L. Mercl, J. Pavlik, The Comparison of Container Orchestrators, International Congress on Information and Communication Technology (2018) 677–685.

[5] A. Saboor, et al., Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives, Appl. Sci. 12(12) (2022) 5793. doi: 10.3390/app12125793.

[6] V. Bushong, et al., On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study, Appl. Sci. 11(17) (2021) 7856. doi: 10.3390/app11177856.

[7] L. Händel, Microservices in the Context of a Fast-Growing Company (2020).

[8] P. Raj, G. David, Engineering Resilient Microservices toward System Reliability: The Technologies and Tools, Cloud Reliability Eng. (2021) 77–116.

[9] G. Blinowski, A. Ojdowska, A. Przybyłek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access 10 (2022) 20357–20374. doi: 10.1109/ACCESS. 2022.3152803.

[10] M. Moravcik, et al., Comparison of LXC and Docker Technologies, 18th International Conference on Emerging eLearning Technologies and Applications (2020) 481–486. doi: 10.1109/ICETA51985.2020.9379212.

[11] C. Cérin, et al., A New Docker Swarm Scheduling Strategy, IEEE 7th International Symposium on Cloud and Service Computing (2017) 112–117. doi: 10.1109/SC2.2017.24.

[12] M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media (2017).

[13] Owasp Kubernetes Top Ten. OWASP. URL: https://owasp.org/www-project-kubernetes-top-ten/

[14] A. Martin, M. Hausenblas, Hacking Kubernetes, O'Reilly Media (2021).

[15] B. Creane, A. Gupta, Kubernetes Security and Observability, O'Reilly Media (2021).

[16] S. Ugale, A. Potgantwar, Container Security in Cloud Environments: A Comprehensive Analysis and Future Directions for DevSecOps, Eng. Proc. 59(1) (2023) 57. doi: 10.3390/engp roc2023059057.

[17] S. Sultan, I. Ahmad, T. Dimitriou, Container Security: Issues, Challenges,

and the Road Ahead, IEEE Access 7 (2019) 52976–52996. doi: 10.1109/ACCESS.2019.2911732.

[18] L. Rice, Container Security: Fundamental Technology Concepts that Protect Containerized Applications, O'Reilly Media (2020).

[19] N. Lopes, et al., Container Hardening Through Automated Seccomp Profiling, 6th International Workshop on Container Technologies and Container Clouds (2020) 31–36.

[20] M. Bélair, S. Laniepce, J. Menaud, Leveraging Kernel Security Mechanisms to Improve Container Security: A Survey, 14th International Conference on Availability, Reliability and Security (2019) 1–6.

[21] W. Dias, P. Siriwardena, Microservices Security in Action, Simon and Schuster (2020).

[22] D. Berardi, et al., Microservice Security: A Systematic Literature Review, PeerJ Comput. Sci. 8 (2022) e779. doi: 10.7717/peerj-cs.779.

[23] M. Driss, et al., Microservices in IoT Security: Current Solutions, Research Challenges, and Future Directions, Procedia Comput. Sci. 192 (2021) 2385–2395. doi: 10.1016/j.procs.2021.09.007.

[24] E. Lee, et al., Quantifying and Generalizing the CAP Theorem, arXiv preprint (2021).

[25] M. Holopainen, Monitoring Container Environment with Prometheus and Grafana (2021).