Bufalino, Jacopo; Di Francesco, Mario; Aura, Tuomas

# Analyzing Microservice Connectivity with Kubesonde

*Published in:*
ESEC/FSE 2023: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering

*DOI:*
10.1145/3611643.3613899

Published: 30/11/2023

*Document Version*
Publisher's PDF, also known as Version of record

# Analyzing Microservice Connectivity with Kubesonde

Jacopo Bufalino
Aalto University
Finland
jacopo.bufalino@aalto.fi

Mario Di Francesco
Eficode and Aalto University
Finland
mario.di.francesco@aalto.fi

Tuomas Aura
Aalto University
Finland
tuomas.aura@aalto.fi

## ABSTRACT

Modern cloud-based applications are composed of several microservices that interact over a network. They are complex distributed systems, to the point that developers may not even be aware of how microservices connect to each other and to the Internet. As a consequence, the security of these applications can be greatly compromised. This work explicitly targets this context by providing a methodology to assess microservice connectivity, a software tool that implements it, and findings from analyzing real cloud applications. Specifically, it introduces Kubesonde, a cloud-native software that instruments live applications running on a Kubernetes cluster to analyze microservice connectivity, with minimal impact on performance. An assessment of microservices in 200 popular cloud applications with Kubesonde revealed significant issues in terms of network isolation: more than 60% of them had discrepancies between their declared and actual connectivity, and none restricted outbound connections towards the Internet. Our analysis shows that Kubesonde offers valuable insights on the connectivity between microservices, beyond what is possible with existing tools.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**.

## KEYWORDS

Microservices, network connectivity, Kubernetes, nework security

**ACM Reference Format:**
Jacopo Bufalino, Mario Di Francesco, and Tuomas Aura. 2023. Analyzing Microservice Connectivity with Kubesonde. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3611643.3613899

## 1 INTRODUCTION

Modern cloud-based applications are composed of several microservices — loosely coupled software components that interact over a network through application programming interfaces (APIs) [20]. Microservices are packaged into software containers; an entire application is deployed and managed by an orchestrator, in practice, with Kubernetes [24] — an open-source software that is now the de-facto standard for container orchestration.



*ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA*

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613899

Typical cloud applications are composed of hundreds or even thousands of microservices [13]. As a consequence, they are very complex distributed systems that pose several challenges to both developers and operators [16, 22], especially in terms of security [3, 12]. A key factor is that microservices are inherently networked, whose implications are poorly understood. The principle of least privilege [26] is a cornerstone for protecting computing systems. In computer networks, it corresponds to restricting connectivity to the bare minimum that allows the application to function; this is accomplished by enforcing adequate access control policies, for instance, through firewalls. Unfortunately, these simple principles are seemingly not being enforced in the cloud: poor isolation between container traffic and unbounded network access are widely recognized risks still today [21, 27]. Securing connectivity between microservices is also challenging for several reasons. The container networking model in Kubernetes hides the complexity behind the cloud infrastructure with a simple, flat address space [17]. Orchestrators do not enforce any restrictions on inter-container communication, defaulting to a permissive "allow all" policy. Developers treat microservices as software libraries: they do not seemingly understand or value the need for isolating connectivity between them, nor the related consequences [23].

There has been abundant research in cloud and container security [7, 14] — from policy management [9, 15, 28, 29] and cloud network security, in addition to a large number of software tools for Kubernetes [1, 5, 6, 30]. However, understanding inter-container traffic to secure microservices has only received marginal attention [8, 19, 31]. This article fills this gap by providing a methodology and a software tool to analyze microservice connectivity.

Our methodology targets assessing the connectivity of microservices in a live cloud application. For this purpose, it establishes probing as an effective means to analyze how the different microservices running on a Kubernetes cluster are able to reach each other and the Internet (Section 2). An attacker typically leverages port scanning to discover services on a remote host from outside the cluster. In contrast, the availability of the orchestrator enables a white-box approach to characterize the actual connectivity between microservices without even requiring their source code. To do so, our methodology specifically addresses the following key questions: How to discover microservices exposing ports to others? When to probe? What should be the sources and the targets of probing? How to interpret the obtained results?

Based on the resulting principles, the Kubesonde tool is then developed (Section 3). Kubesonde allows developers and system administrators to characterize the connectivity between different containers (namely, microservices) in a Kubernetes cluster. Kubesonde has a modern, cloud-native design: it seamlessly integrates with the orchestrator and has minimal impact on live applications, making

it suitable for running even in production. Kubesonde is an open-source software available at https://github.com/kubesonde.

Kubesonde is also leveraged to assess connectivity of microservices in real cloud applications (Section 4). For this purpose, we analyzed 200 widely-used applications packaged for Kubernetes. Our analysis revealed that more than 60% of them had discrepancies between their declared and actual connectivity, and none restricted outbound connections towards the Internet.

## 2 METHODOLOGY

This section discusses the proposed methodology to assess microservice connectivity. It first introduces probing as the key technique to achieve accurate results, then explains how to discover targets, where and when to probe, and how to interpret its outcomes.

### 2.1 Probing Connectivity

Connectivity in this article means the ability to connect to an open TCP or UDP port. Thus, we are interested in the low-level access between microservices below the application and middleware layers. Rejecting an end-to-end request often depends on multiple policies and mechanisms located at different parts of the network or virtualization stacks. These components jointly determine if a connection from one microservice to an open port on another is successful. Therefore, it is impossible to accurately measure connectivity without running the microservices.

Therefore, *probing* between running containers is the most effective way to get an accurate picture of the connectivity between the microservices. The concept of probing is already part of Kubernetes as it is employed to verify the status and readiness of applications. Existing network mapping libraries, provide efficient and accurate means to test connections to TCP and UDP endpoints defined as a combination of protocol, IP address, and port number. We build on these to determine whether a target port responds or not.

This low-level view of connectivity is similar to how a typical attacker approaches network discovery. In contrast, the probing process here is considerably more complete and efficient than typical asset discovery carried out by hackers. In fact, the availability of an orchestrator allows viewing the network connectivity from every possible source to every possible target inside the cluster.

### 2.2 Target Discovery

The availability of the orchestrator enables taking a white-box approach in discovering microservices and their open ports. We classify and describe next the different sources that allow to do so.

**IP Address and Declared Ports**. The Kubernetes API allows to enumerate the nodes in the cluster and the pods therein. In particular, the network-related information associated with the pod include its IP address within the cluster and the *declared ports* where all containers in the pod should be listening for inbound connections. A pod encapsulates one or more containers, depending on the architectural style of the application. These containers share resources in the same namespace, including their IP address; therefore, they cannot use the same port on it. The rest of the discussion in this article refers to pods for preciseness, even though the methodology presented here is valid irrespective of how the application and its individual containers are arranged into pods.

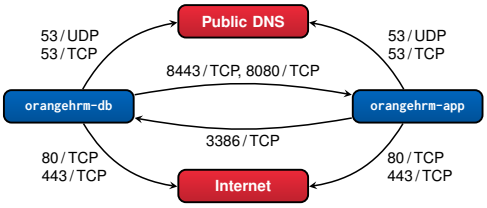



**Figure 1: Sample connectivity graph obtained by Kubesonde for the OrangeHRM human resource management application as packaged by Bitnami.**

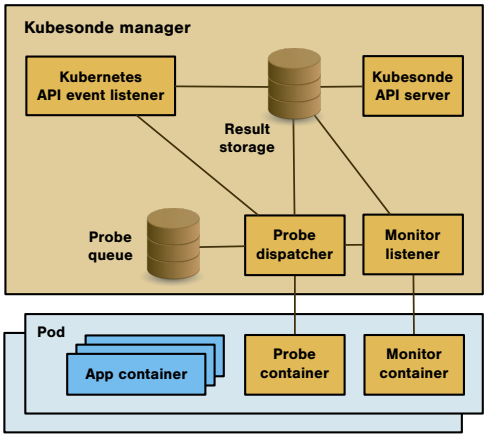
**Network Namespace**. The orchestrator obtains the information discussed above based on the declarative specification of the application, namely, the content of the configuration files associated with its microservices. However, such a declarative specification is only documentation, not an enforced policy: the list of ports the developer reports to the orchestrator are not guaranteed to correspond those actually opened by the software. A different issue is related to processes that are not created through the orchestrator, which is then unaware of their existence. To overcome this incomplete view given by the orchestrator, we monitor the network namespace on each pod for changes to the IP addresses and open ports. This allows to detect all ports actually opened by all containers therein.

**Service Addresses**. The service abstraction within Kubernetes allocates a virtual IP address in the cluster network to scale an application based on incoming traffic. Kubernetes routes requests that were sent to the service address to the pods that implement that service. Access to the service address can also be restricted with a network policy. We treat the service addresses and their ports as probing targets, allowing the developer to verify the presence and effect of the policy – or to become aware of its absence.

### 2.3 Deploying Probes

Now that we have established the targets of probing, it remains to define where the sources should be located. Intuitively, it would seem enough to use a single endpoint within the cluster to probe all open ports therein, for instance, a pod created for the sole purpose of probing the cluster network. Unfortunately, this approach is not enough because there may be already network policies applied in that same location or in isolated parts of the cluster that would never be discovered or probed. A far better option is to instrument *each pod* with probing software that tests the potential connections to *all* other pods. In fact, the probe needs to run in the same network namespace as the source pod to obtain accurate results.

It is possible to integrate the probing software into existing pods without modifying the applications, thereby minimizing the impact on their execution. We achieve this by leveraging the API offered by Kubernetes and inject a new container into a pod, by following the sidecar pattern [10]. This choice has several advantages: it is flexible as it allows a wide range of configuration options; it allows packaging the probe itself as a container; and it makes it easy to establish a control plane for probe containers. Indeed, the orchestrator has already established communication channels to each node in the cluster, therefore, it is not necessary to open new ports or to reconfigure network policies for the control plane.




(a)

```yaml
apiVersion: security.kubesonde.io/v1
kind: Kubesonde
metadata:
  name: Test scenario
spec:
  namespace: wordpress-app
  probe: all
  exclude:
  - fromPodSelector: all
    toPodSelector: component=database
    port: 3306
    protocol: TCP
  include:
  - fromPodSelector: all
    address: api.google.com
    port: 443
    expected: allow
```

(b)

**Figure 2: (a) Kubesonde components in a Kubernetes cluster. (b) Sample Kubesonde configuration file, including a mandatory Kubernetes namespace (to narrow down the pods under consideration) and an optional list of policies and actions to verify; each action is of either `allow` or `deny` type and contains a fixed set of fields to further specify the probe target or source.**

## 2.4 Continuous Monitoring

Pods and open ports are ephemeral and created on demand; consequently, it is not enough to just enumerate the pods, deploy the probes, and run them once. Instead, it is imperative to carry out continuous discovery. This is achieved by listening to events from the Kubernetes API and deploying sidecar containers to any newly created pod. Monitoring the network namespace of the pod also allows us to detect changes in the open ports within that pod.

In addition to continuous discovery of possible new endpoints, re-probing those previously discovered is necessary for two main reasons. First, the ports discovered via the Kubernetes API may start listening for connections at a later stage, or they may only be intermittently open. When the microservice is starting or scaling up, the software in the containers may need time to initialize before it is ready to accept connections. Second, we aim to detect and report variations in service reachability due to changes in the network configuration, such an administrator tightening firewall rules.

## 2.5 Interpreting Results

Probing a port is usually as simple as trying to initiate a TCP connection to it, which can be optimized by SYN scanning [11]. Successful connections from the sidecar container to a target port are thus easy to report. On the other hand, interpreting the results is less straightforward when the connection fails. There may be no response from the target, leading to a timeout. The response might also be an ICMP Destination Unreachable error message — with variable error codes — or a TCP Reset. Most of these failures might be caused by an access policy or permanent isolation from the target, but also due to a temporary glitch in the network or destination service. In our experience, transient failures commonly happened when the target container or service process was not yet ready to accept connections; therefore, we re-probe the unreachable targets to avoid misinterpreting the results, and report eventual success or consistent failure. We do not try to distinguish between different types of failures because firewalls and other filtering policies often return misleading responses.

There are some cases where we might not get a response from an open port and record a false negative result. First, the UDP port at the target might be listening but not responding because we have not sent it a meaningful application-layer message. Second, the target TCP or UDP port might expect packets with a specific source port number, and the probes use random source ports. We may not know the expected source port, and even if we do, the probe container cannot bind to any port already in use by another container in the pod. Conversely, there is also the chance of false positives, for instance, when an application opens a socket without the purpose of actually using it to communicate, but rather to obtain information on the network (e.g., the IP address of a node).

Probing results comprise the open ports found based on the declarative specification of the application (e.g., as Kubernetes resources) and those reported by monitoring the network namespace of pods. Any discrepancies between the two indicates either a need to update the specification or an opportunity to restrict network-layer access. The results also include a timeline of the probing with the source and target and whether the connection was allowed or denied, along with error indicators. This allows to derive network *connectivity graphs* such as the one shown in Figure 1. Probe results can also be leveraged to verify the impact of configuration and policy changes and to track the evolution of the system over time.

# 3 KUBESONDE

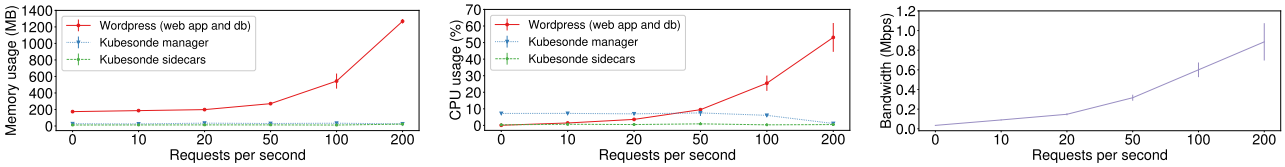
This section discusses the Kubesonde connectivity analysis tool that implements the probing method introduced above (Figure 2). We first introduce the system architecture, then detail its implementation and finally characterize its performance overhead.

## 3.1 Implementation

Kubesonde comprises three types of components: probe containers, monitor containers, and a manager (Figure 2a).

**Table 1: Resource utilization of Kubesonde's components in comparison with the WordPress application and Istio.**

| | Kubesonde | | | | WordPress | | | Istio | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Metric** | **Probe** | **Monitor** | **Manager** | **Total** | **Web app** | **Database** | **Total** | **Istiod** | **Istio-ingress** | **Total** |
| Storage (MB) | 9.40 | 4.70 | 60.69 | 74.79 | 157.46 | 114.37 | 271.83 | 71.32 | 89.40 | 160.72 |
| Memory (MB) | 2.79 | 3.43 | 19.50 | 25.72 | 98.51 | 75.10 | 173.61 | 48.31 | 39.80 | 88.11 |
| CPU utilization (%) | 0.14 | 0.04 | 7.17 | 7.35 | 0.22 | 0.08 | 0.30 | 0.05 | 0.03 | 0.08 |




(a) (b) (c)

**Figure 3: Resource utilization of Kubesonde under varying traffic for different metrics: (a) memory, (b) CPU, and (c) bandwidth.**

The probe and monitor containers are injected into each pod according to the *sidecar* design pattern [4] — as discussed in Section 2.3 — without the need for modifying the application code. We further minimize the impact on running applications by using *ephemeral containers*[1], a special type of short-lived container that runs in an existing pod without requiring its re-deployment, which would otherwise entail some downtime. Ephemeral containers are not guaranteed any resources; they can only take what is allocated to the pod but not used by other containers therein.

The **probe container** initiates outbound connections to other microservices in the same cluster and to selected services in external networks as well as in the Internet. The probe container lies in the same network namespace as the containers in the pod. As a consequence, it initiates outbound connections from the same network as the source container, thereby sharing its logical location therein. The **monitor container** observes the network stack inside the pod it belongs to. In particular, the monitor container accesses the interface offered by the underlying operating system (i.e., the Linux `proc` filesystem) to gain information on the open ports of the containers in that pod, as they all share a single network namespace.

The Kubesonde **manager** is a control-plane component that issues commands to the probe and monitor containers, receives and stores the results, and exposes an API for configuration and data retrieval. The *event listener* subscribes to the events related to the pod lifecycle in the cluster through the Kubernetes API. Once pods are created, it extracts their declarative specification including the declared ports, injects the probe and monitor containers into them, and initiates the actual probing process. The *probe dispatcher* maintains a list of probing sources and targets, along with a queue of probes to be executed. Based on these, the dispatcher issues commands to the probe containers, collects the results, and schedules periodic re-probing. The *monitor listener* receives additional port information from the monitor containers and generates new probing targets. All components in the manager save the relevant information to the *result storage*. Finally, the *API server* allows Kubesonde to communicate with the external world through its own API, which extends the Kubernetes API as a custom resource with its own controller [25]. The API provides endpoints to start and customize the probing process and to retrieve the probing results. Probing is specified in a YAML file that indicates the Kubernetes namespace and selectors for the source and target pods. It is also possible to define external targets in the public Internet, the data center network, or another cluster. Figure 2b shows a sample configuration.

## 3.2 Performance Evaluation

We compared the resource use of an application with and without Kubesonde to characterize the performance overhead incurred by probing. In the first experiment, we measure the CPU and memory usage; in the second one, these and also bandwidth as a function of application traffic. WordPress as packaged by Bitnami[2] for Kubernetes was selected as the representative application, as it is also the one explicitly mentioned in [21]. The application comprises two pods: one for the web application (served by the `nginx` web server) and another for the database (i.e., MariaDB). Experiments were run on Minikube[3] version 1.23.2 and Kubernetes version 1.23.16 with the independent replication method. Measurements were collected through the Metrics[4] API over 25 replications; in each iteration, the application is first given time to stabilize, then metrics are collected over a time frame of half a minute. The results report the related averages along with their standard deviations, when significant.

Table 1 quantifies the average resource utilization of Kubesonde by component, in comparison with the considered WordPress application without application traffic, as a baseline. In the table, storage refers to the size of the corresponding container images. The table clearly shows how the resource utilization of Kubesonde is low, especially in terms of memory. Both the memory and the CPU utilization of the sidecars (i.e., the probe and monitor containers) are insignificant compared to the containers in the actual application. Similar considerations apply to the manager, which consumes less than 30 MB of RAM and only 7% of CPU — the CPU utilization of Wordpress is minimal just because it is not serving any request.

[1] https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/

[2] https://charts.bitnami.com/bitnami/wordpress

[3] https://github.com/kubernetes/minikube

[4] https://github.com/kubernetes/metrics

We also report resource utilization of two selected components in the widely-used Istio service mesh [2] for comparison purposes: ISTIOD, realizing the control-plane of the mesh; and ISTIO-INGRESS, exposing a service outside of the service mesh in Kubernetes. The related storage and memory utilization are significantly higher than those in Kubesonde; particularly, the two Istio components together consume more than three times the memory used by Kubesonde, whereas their CPU utilization is similar to that in Wordpress.

We then performed a different set of experiments by varying the application load. For this purpose, we employed the `httperf` [18] tool to inject client traffic into the web app until a maximum of 200 requests per second, without scaling[5] the application. The obtained results are shown in Figure 3. The figure exhibits a similar trend for memory (Figure 3a) and CPU utilization (Figure 3b): they both significantly increase with traffic for Wordpress, whereas they are constant or decrease for Kubesonde. This trend reflects the design choices of Kubesonde, as we do not specify resources constraints during the installation phase. For this reason, the control plane of Kubernetes reduces the amount of CPU and memory assigned to Kubesonde when other components require more resources. The bandwidth usage of Kubesonde (Figure 3c) is not as stable as the other resources. This is due to the variability in the information exchanged between the monitor probe and the manager to characterize open connections. Still, the overhead is limited: no more than 1.2 Mbps at most, even for the highest application traffic considered.

## 4 ANALYSIS OF CLOUD APPLICATIONS

This section presents the results from a large-scale analysis of popular cloud applications. Specifically, we considered applications defined as Helm charts, similar to [8]. Helm[6] is a widely used software to package and manage applications for Kubernetes clusters; a Helm chart[7] is a collection of files that describe applications as Kubernetes resources, including configuration values and dependencies. Accordingly, we analyzed the 200 most starred charts defined by verified publishers (i.e., the owners of packaged code) in Artifact Hub[8], the largest repository of Helm charts. The related results are summarized in Table 2.

The first part of the table overviews the features on the considered applications in terms of security properties (i.e., role-based access control rules, security contexts, and network policies). The data show that security features were available only to a fraction of these application, primarily in terms of security contexts: 57% and 33% of the applications defined security policies to restrict capabilities for the pods and the individual containers therein, respectively. Role-based access control is available for less than half of the applications; network policies were specified in only 20% of cases, but not applied by default in any of them.

The last part of the table presents findings obtained by deploying all the considered applications with Helm version 3.7 on the same testbed described in the previous section, together with Kubesonde. This allowed us to quantitatively characterize their connectivity in terms of network-related attributes. The results show that 61% of the considered applications exhibited a mismatch between declared and actually used ports. We further delved into the affected applications and found that a large share of discrepancies arises from optional features, such as for debugging and performance optimization (e.g., caching). Moreover, we found that none of the applications imposed restrictions on outbound traffic towards the Internet, namely, all their pods are able to initiate connections to the public Internet. This means that a compromised application container is capable of propagating the attack to other sites, causing outbound denial of service, or downloading a malware payload.

**Table 2: Features of the 200 considered applications.**

| Feature | Value (%) |
|---|---|
| RBAC available | 36 |
| Pod Security Context available | 57 |
| Container Security Context available | 33 |
| Network Policy available | 20 |
| Network Policy enabled | 0 |
| Inconsistencies between declared and used ports | 61 |
| Restrictions imposed on outbound connections | 0 |

## 5 CONCLUSION

This work has introduced a methodology to assess the connectivity of microservices in cloud applications. The proposed approach leverages an insider view of a cluster running a container orchestrator to probe open ports and evaluate whether they can be accessed or not. Based on such a methodology, Kubesonde was designed and implemented as a cloud-native tool for Kubernetes. Kubesonde was also employed to carry out an evaluation of connectivity in 200 popular cloud applications packaged for Kubernetes. Kubesonde allowed to highlight security issues affecting them, based on a characterization of their connectivity in a real deployment environment, without the need for employing service meshes.

This work focused on mapping the connectivity between pods on the same cluster network. It would be interesting to extend our solution to multi-cluster or even multi-cloud scenarios. Moreover, modern cloud applications reuse components at multiple levels, leading to dependencies between microservices. Unfortunately, cloud applications currently lack modular internal security boundaries between the pod and the cluster. Limiting interactions among components in service composition is a challenging problem that calls for further research. Finally, Kubesonde and the analysis in this article target Kubernetes, however, the underlying methodology can be applied to other orchestrators as well. Conversely, Kubesonde does not leverage platform-specific technologies such as eBPF, making it suitable for diverse scenarios at the cost of some loss in efficiency. Extending both the applicability and the performance of Kubesonde are interesting directions for future work.

## ACKNOWLEDGMENTS

This work was partially funded by the Research Council of Finland under grant number 345964. The authors would like to thank Matteo Calabrese for his help with the setup used for the evaluation.

[5] We have verified that the maximum traffic can still be managed by the application deployed as a single replica.
[6] https://helm.sh/
[7] https://helm.sh/docs/topics/charts/
[8] https://artifacthub.io/

## REFERENCES


[1] *Illuminatio: The kubernetes network policy validator.* Retrieved May 05, 2023 from https://github.com/inovex/illuminatio

[2] *Istio.* Retrieved May 05, 2023 from https://istio.io/

[3] Yasemin Acar et al. 2017. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *2017 IEEE Cybersecurity Development (SecDev)*. 22–26. https://doi.org/10.1109/SecDev.2017.17

[4] Akhan Akbulut and Harry G. Perros. 2019. Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing* 23, 6 (2019), 19–27. https://doi.org/10.1109/MIC.2019.2951094

[5] Aquasecurity. *Kube-hunter: Hunt for security weaknesses in Kubernetes clusters.* Retrieved May 05, 2023 from https://github.com/aquasecurity/kube-hunter

[6] Aquasecurity. *Kubescape.* Retrieved May 05, 2023 from https://github.com/kubescape/kubescape

[7] Priyanka Billawa, Anusha Bambhore Tukaram, Nicolás E. Díaz Ferreyra, Jan-Philipp Steghöfer, Riccardo Scandariato, and Georg Simhandl. 2022. SoK: Security of Microservice Applications: A Practitioners' Perspective on Challenges and Best Practices. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '22)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/3538969.3538986

[8] Agathe Blaise and Filippo Rebecchi. 2022. Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 59–69. https://doi.org/10.1109/CLOUD55607.2022.00022

[9] Gerald Budigiri, Christoph Baumann, Jan Tobias Mühlberg, Eddy Truyen, and Wouter Joosen. 2021. Network Policies in Kubernetes: Performance Evaluation and Security Analysis. In *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. 407–412. https://doi.org/10.1109/EuCNC/6GSummit51104.2021.9482526

[10] Brendan Burns and David Oppenheimer. 2016. Design Patterns for Container-based Distributed Systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[11] Marco De Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O De Vivo. 1999. A review of port scanning techniques. In *ACM SIGCOMM Computer Communication Review*. 41–48.

[12] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating System Operators' Perspective on Security Misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada). 1272–1289. https://doi.org/10.1145/3243734.3243794

[13] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters* 17, 2 (2018), 155–158. https://doi.org/10.1109/LCA.2018.2839189

[14] Hyunsu Jang, Jaehoon Jeong, Hyoungshick Kim, and Jung-Soo Park. 2015. A Survey on Interfaces to Network Security Functions in Network Virtualization. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. 160–163. https://doi.org/10.1109/WAINA.2015.103

[15] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. 2021. Automatic Policy Generation for Inter-Service Access Control of Microservices. In *USENIX Security 21*. 3971–3988.

[16] Minghua Ma, Yudong Liu, Yuang Tong, Haozhe Li, Pu Zhao, Yong Xu, Hongyu Zhang, Shilin He, Lu Wang, Yingnong Dang, Saravanakumar Rajmohan, and Qingwei Lin. 2022. An Empirical Investigation of Missing Data Handling in Cloud Node Failure Prediction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 1453–1464. https://doi.org/10.1145/3540250.3558946

[17] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci. 2021. Understanding the Security Implications of Kubernetes Networking. *IEEE Security & Privacy* 19, 05 (2021), 46–56. https://doi.org/10.1109/MSEC.2021.3094726

[18] David Mosberger and Tai Jin. 1998. Httperf—a Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (dec 1998), 31–37. https://doi.org/10.1145/306225.306235

[19] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. BASTION: A Security Enforcement Network Stack for Container Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 81–95.

[20] Sam Newman. 2021. *Building microservices* (2nd ed.). O'Reilly Media.

[21] Open Web Application Security Project – Kubernetes Top 10. *K07:2022 Missing Network Segmentation Controls.* Retrieved May 05, 2023 from https://github.com/OWASP/www-project-kubernetes-top-ten/blob/main/2022/en/src/K07-network-segmentation.md

[22] Xin Peng, Chenxi Zhang, Zhongyuan Zhao, Akasaka Isami, Xiaofeng Guo, and Yunna Cui. 2022. Trace Analysis Based Microservice Architecture Measurement *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1589–1599. https://doi.org/10.1145/3540250.3558951

[23] RedHat Inc. 2022 State of Kubernetes Security Report. Retrieved May 05, 2023 from https://www.redhat.com/en/resources/state-kubernetes-security-report

[24] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale.* O'Reilly. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[25] Gigi Sayfan. 2020. *Mastering Kubernetes* (third ed.). Packt Publishing Ltd. https://www.packtpub.com/product/mastering-kubernetes-third-edition/9781839211256

[26] Fred B Schneider. 2003. Least privilege and more. *IEEE Security & Privacy* 1, 5 (2003), 55–59. https://doi.org/10.1109/MSECP.2003.1236236

[27] Murugiah Souppaya, John Morello, and Karen Scarfone. 2017. *Application Container Security Guide.* NIST Special Publication 800-190. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-190

[28] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. 2015. Security-as-a-Service for Microservices-Based Cloud Applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 50–57. https://doi.org/10.1109/CloudCom.2015.93

[29] Kennedy A. Torkura, Muhammad I.H. Sukmana, and Christoph Meinel. 2017. Integrating Continuous Security Assessments in Microservices and Cloud Native Applications. In *Proceedings of The10th International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 171––180. https://doi.org/10.1145/3147213.3147229

[30] Weaveworks. Scope: Monitoring, visualisation & management for Docker & Kubernetes. Retrieved May 05, 2023 from https://github.com/weaveworks/scope

[31] Hui Zhu and Christian Gehrmann. 2022. Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. 129–137. https://doi.org/10.1109/COMSNETS53615.2022.9668504


Received 2023-05-18; accepted 2023-07-31