Understanding the Security Implications of Kubernetes Networking

Francesco Minna and Balakrishnan Chandrasekaran | Vrije Universiteit Amsterdam
Agathe Blaise and Filippo Rebecchi | Thales SIX GTS France
Fabio Massacci | University of Trento and Vrije Universiteit Amsterdam

Container-orchestration software such as Kubernetes make it easy to deploy and manage modern cloud applications based on microservices. Yet, its network abstractions pave the way for "unexpected attacks" if we approach cloud network security with the same mental model of traditional network security.

Microservices have become the template for cloud-native applications: easy to develop, deploy, debug, scale, and share. When an application is decomposed into independent microservices, ensuring that the services can communicate with one another in a secure way introduces new challenges, particularly when the decomposition results in many services. Even rudimentary cloud applications contain a few tens of microservices, and some of the largest (e.g., Netflix and Uber platforms) contain hundreds or thousands of microservices, possibly running on several containers. Container-orchestration software such as Kubernetes (K8s)[1] provide a simplified interface or model to address these challenges.

At the same time, abstractions make it easy to overlook security threats. For example, (in)secure practices concerning use of K8s default configuration have been well studied.[2,3] Security issues in software-defined networking (SDN) solutions used for managing cloud infrastructure have also been investigated.[4,5] Nam et al.[6] present an overview of security challenges in container networks and the limitations of common networking plug-ins.

In particular, the security implications of K8s networking components (e.g., how K8s configures connectivity between services and enforces network-security policies) are largely unexplored. Indeed, when we think about networking between microservices, we have a "mental model" of networking derived from physical networks, with switches and interfaces interconnected with physical cables—a model that we show significantly departs from reality.

As a result, when thinking about (cloud) network security, we may picture "digitally unbridgeable moats" that do not really exist. The correct analogy with traditional networking would be that as one is able to escalate within a switching device, then one can start laying cables between different devices. The key takeaway is not that K8s is insecure, but that it is insecure to apply the "mental extension" of traditional network security terminology to a different world.

A Playground for "Unexpected Attacks"

To understand the issues, consider some typical deployment scenarios in which a company is wishing to use K8s.

A *K8s-single-cluster* consists of one master and one or more (i.e., a customizable number of) worker nodes. The applications aimed at the users are deployed on two clusters—"development" and "production"—both of which contain the same set of applications, but with different levels of security (typically, more restricted for the production than the development cluster). A network-security policy separates the nodes.

A *K8s-multicluster* setup consists of two clusters composed of one master and at least one worker node for each. To add a layer of security, one cluster can be

Digital Object Identifier 10.1109/MSEC.2021.3094726
Date of current version: 27 July 2021

"development" and the other can be in the "production," each deployed in different network subnets not meant to access one another.

A *K8s-custom-multicluster* is a fully customizable setup, which allows the user to specify both the number of master nodes and worker nodes to be used, where etcd database containing the clusters information should be deployed (within the master nodes or as an external cluster for high availability) and other segregation information available through Linux namespaces. It is also possible to specify, for each K8s component, the release version to be installed (this setup is suitable to replicate production-like environments).

Figure 1 provides an overview of the multicluster setup and its main components. Different application services, possibly segregated by security policies at the operating system level, are typically present even in a single-cluster setup:

- *longhorn*: providing distributed storage
- *nfs server*: providing persistent storage
- *development*: three applications—Wordpress (with MariaDB), Nginx, and Guestbook (with a Redis leader, Redis follower, and front end)
- *production*: same applications as for development.

We provide a practical testbed,[7] built on Vagrant for reproducibility reasons, where the above scenarios can be replicated through containers and virtual Machines (VMs). VMs are created and deployed created from a host machine in a private network, not accessible from the internet. VMs can, however, reach the Internet via a network address translator (NAT).

The "Unexpected Threats" Model

In this testbed scenarios, we consider sample attack scenarios from either external or internal attackers. So, we assume that all attacks start by compromising a pod in some way (the *Initial Access* of MITRE ATT&CK framework as adapted by Microsoft for K8s[8]).

With the traditional mental model of network security, such attacks should remain confined to the initial compromised pod: network-security measures are in place. Additional exploits would be needed to move around. Yet, exploiting the connectivity of K8s components, the cluster may still be compromised. We will return to them with more precise details in Table 1 after describing the network functionality.

- FirewallHole *(bypassing security barriers of an overlay network)*: An attacker launches a SYN flood denial-of-service (DoS) attack against a service bypassing an (apparent) firewall by mimicking the encapsulation of the plug-in in charge of networking that at each node mimics the existence of an overlay network.
- Hit&Spread [*container shell through remote code execution (RCE) vulnerability in the web application*]: An attacker can exploit an RCE in a web application to get a reverse shell on a container and then access sensitive information, laterally move within the cluster, and escalate privileges.
- Replace&Propagate (*supply chain attack through malicious container image*): An attacker can deceive developers into deploying a malicious container, which then contacts a

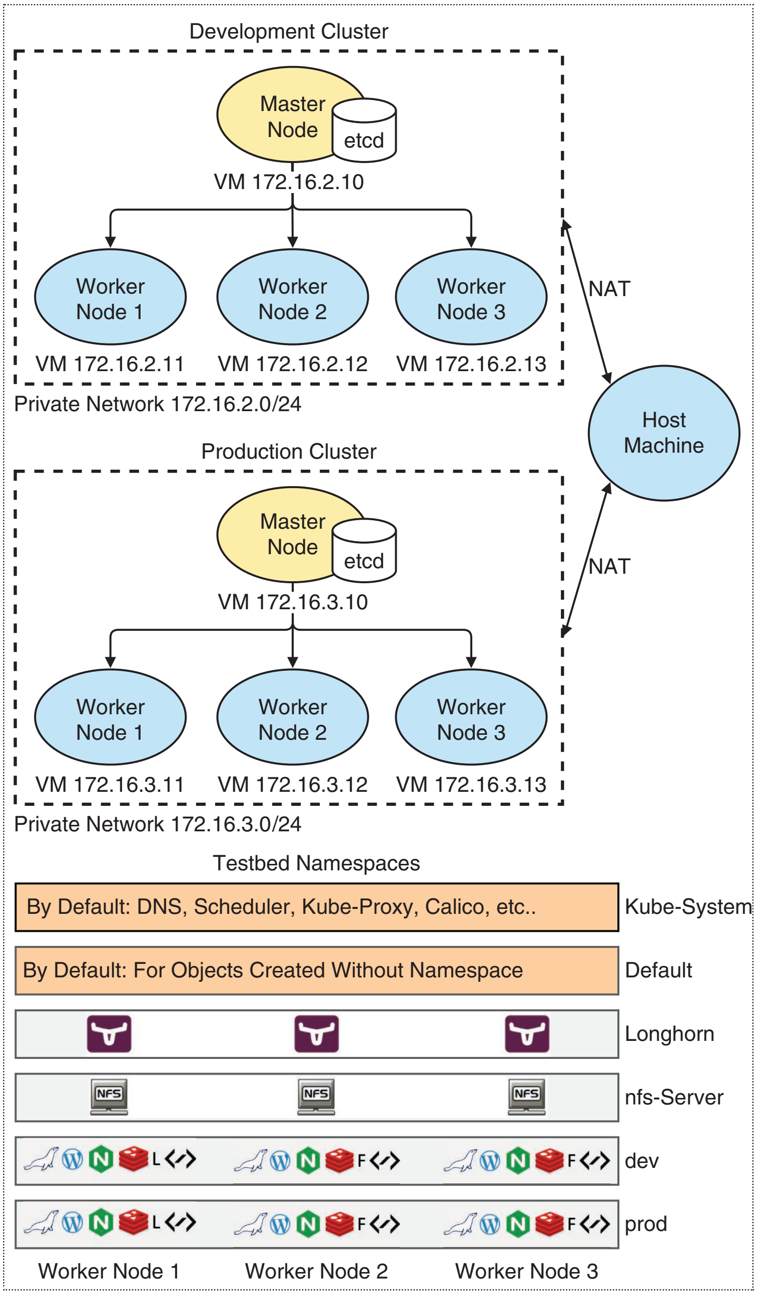


Figure 1. An overview of the multicluster components.

command-and-control server and hijacks the whole cluster including services running on other containers.

A Primer on Containers and Kubernetes

To explain why these attacks are feasible, some background material on containers and key components of K8s is useful.

A container emulates the operating system layers to offer a virtualized and self-contained environment with its own subprocesses and resources. Container isolation in a typical Linux environment is implemented through *namespaces*, which allow a kernel to partition resources among a set of processes. Specifically, a network namespace is a copy of the network stack, including network interfaces, routing and firewall rules, which can be assigned to each process or container. The longhorn, nfs server, dev, and prod namespaces shown in Figure 1 implement a similar resources isolation at a K8s cluster level.

Deployment and management of containers is typically automated with *orchestration engines* such as K8s, Docker Swarm, and AWS ECS. In this article we focus only on K8s, the most widely used orchestration software.[9]

An application running on K8s is deployed within a cluster, a set of machines (either virtual or physical) for running containerized applications. As shown in

Table 1. Details of the example attack scenarios in a K8s cluster.

Attack Scenario	Alternative Steps or Scenarios
FirewallHole: • The target is a web application (front end, database, and back-end server); firewall policies allows only the back end to send packets to the database, and pods without NET_RAW capability (i.e. no source IP address spoofing). • The attacker wants to run a SYN flood DoS attack on the database, by crafting User Datagram Protocol (UDP) packets to mimic Virtual Extensible LAN (VXLAN) encapsulation. • The attacker, miming the Flannel VXLAN encapsulation (i.e., UDP packets with VXLAN header, destination IP set to the node's IP, and destination port set to 8472—default VXLAN UDP port), can bypass the firewall and send packets from the front-end pod to the database.	• Deploying a malicious CNI plug-in, which could allow malicious requests and enable MITM attacks. • K8s objects dynamically created and located in a CIDR not covered by the firewall. • CVE-2020-10749 vulnerability found in affected container networking implementations allowing malicious containers in a cluster to perform MITM attacks. • CNI plug-in that does not handle network policies (e.g., Flannel) or network policies not defined by the user.
Hit&Spread: • Consider a web application containing a remote code execution (RCE) vulnerability in the code or a third-party dependency, that allows obtaining a reverse shell in a container. • The attacker can communicate with the API server via kubectl with tokens and certificates mounted on the compromised pod. • The attacker can also perform malicious actions like mounting the host's file system on new containers, accessing other pods in the cluster, asking the API server to modify containers or intercepting network traffic.	• Transport Layer Security (TLS) authentication disabled for any component on the master node: API server, controller-manager, scheduler, and etcd server. • Interaction with the cloud provider: obtaining the node's credentials from the metadata API, gaining K8s authentication tokens from cloud storage buckets, modifying or creating compute instances, and modifying or duplicating storage. • Exploiting users with a large set of permissions (e.g., for accessing secrets, creating pods or deployments). • Secrets management: accessing secrets stored as environment variables or in other insecure ways.
Replace&Propagate: • Consider the deployment of a malicious image controlled by the attacker and able to open a reverse shell or communicate with a command-and-control server. • The attacker gets a reverse shell on the malicious container and, similarly as before, can install custom scripts or malicious programs, access other pods and secrets, intercept network traffic, and escape on the node.	• The attacker can deceive the developers in deploying the malicious image either by sharing it on public registries (e.g., Docker Hub) with misleading names (i.e., typosquatting attacks), by gaining access to a repository and directly modifying the source code, or exploit a registry's vulnerability and hijack the images (e.g., CVE-2019-16097). • Given attacker's access to the cluster through a compromised container image, which developers can also reuse as a base image for other containers, enlarging the attack surface.

Figure 1, in every cluster there is (at least) one master node and several worker nodes. Master nodes have the task of managing all cluster (i.e., K8s objects and worker nodes) and keeping it at the desired state, scheduling the application containers on the worker nodes, which are the computing units. To provide high availability, both the master and worker nodes can be replicated, either physically or virtually (e.g., through VMs).

The main components of a worker node are:

- *pod*: the smallest deployable object containing at least one container; a pod (or the containers running within it) is attacked and initially compromised in all our scenarios
- *kubelet*: managing and checking running pods
- *kube-proxy*: implementing NAT for new services; this is the component implementing the network policies through iptables rules retrieved from the etcd datastore available in the Master node
- *container runtime*: container engine that runs containers.

Instead, the main components of a master node are:

- *API server*: REST API control manager that controls the whole cluster; K8s users can interact with the cluster through kubectl, a command-line tool, or the web dashboard, by sending commands to the API server
- *controller-manager*: controller loops on cluster objects
- *scheduler*: scheduling pods on worker nodes
- *etcd*: key-value distributed database storing cluster configurations; a faulty network analogy would be a dynamic host configuration protocol server database. More properly, it is an identity database for workers and pods. Should it fail or be compromised, there is no longer a proper distinction between pods and network policies cannot be retrieved anymore.

By default, the K8s network among worker nodes and pods is all flat: to provide network segmentation and to restrict the communication between different objects, K8s allows defining network policies.

A network policy (which is actually a misnomer as will be coming apparent) allows specifying how a pod is allowed to talk to other networking components, such as other pods, services, and so on. Such policy is not enforced by K8s itself, but by network plug-in, a container network interface (CNI) aiming to connect a container engine to a network, providing connectivity specifications for the running containers. Kumar and Trivedi et al.[10] provide an extensive performance comparison of common CNI plug-ins. By default, all policies are stored in the etcd database and retrieved by the plug-in agent running on each node. How these policies are enforced depends on the plug-in (e.g., through iptables rules or admission controllers). In fact, creating a network policy without a CNI plug-in will have no effect on the cluster traffic.

Contrary to common belief, a CNI plug-in is not a K8s component, it is not bound to it in any way, and it does not depend on K8s. In a K8s cluster, a CNI simply acts as a middleware between pods and the container engine being used. Specifically, the kubelet contacts the CNI plug-in providing a JavaScript Object Notation config file containing the network specifications that a worker node should use (e.g., the network subnet) with its pods. This has strong implications on the way networking is implemented using CNIs. A security policy enforced by CNI is only enforced if a K8s component queries the appropriate CNI for policy and interfaces mapping and does what is told.

Kubernetes Networking: Bottom-Up

Within a K8s cluster, every CNI plug-in must guarantee the following properties:

- a container (and pod) can communicate with any other container (and pod) on any worker node without using NAT
- a worker node can communicate with any pod on any worker node without NAT
- each pod is assigned a unique IP address across the entire cluster (i.e., an IP-per-pod model).

In this section, we elucidate various communication scenarios between the key K8s entities (shown in Figure 2) and highlight security issues relevant to each scenario.

Container-to-Container Networking

The simplest scenario consists of communication between containers within the same pod, which is represented by the green line in Figure 2. Containers within the same pod share the same network namespace (abbreviated, henceforth, as netns). They share, hence, the same (virtual) network stack (i.e., network interfaces, routing table, and so on), and they can communicate over localhost. Thus, a compromised container has (network) access to the other containers running in the same pod. The CNI plug-in, invoked by the kubelet, and in charge of setting up network interfaces does not disallow (or even monitor) communication over localhost.

Pod-to-Pod Networking

Moving up one layer, pods can "talk" to each other. We distinguish between two cases: two pods communicate within the same worker node (yellow line in Figure 2), or they are on different nodes (purple line in Figure 2). Nodes may also not be part of the same subnet (e.g.,

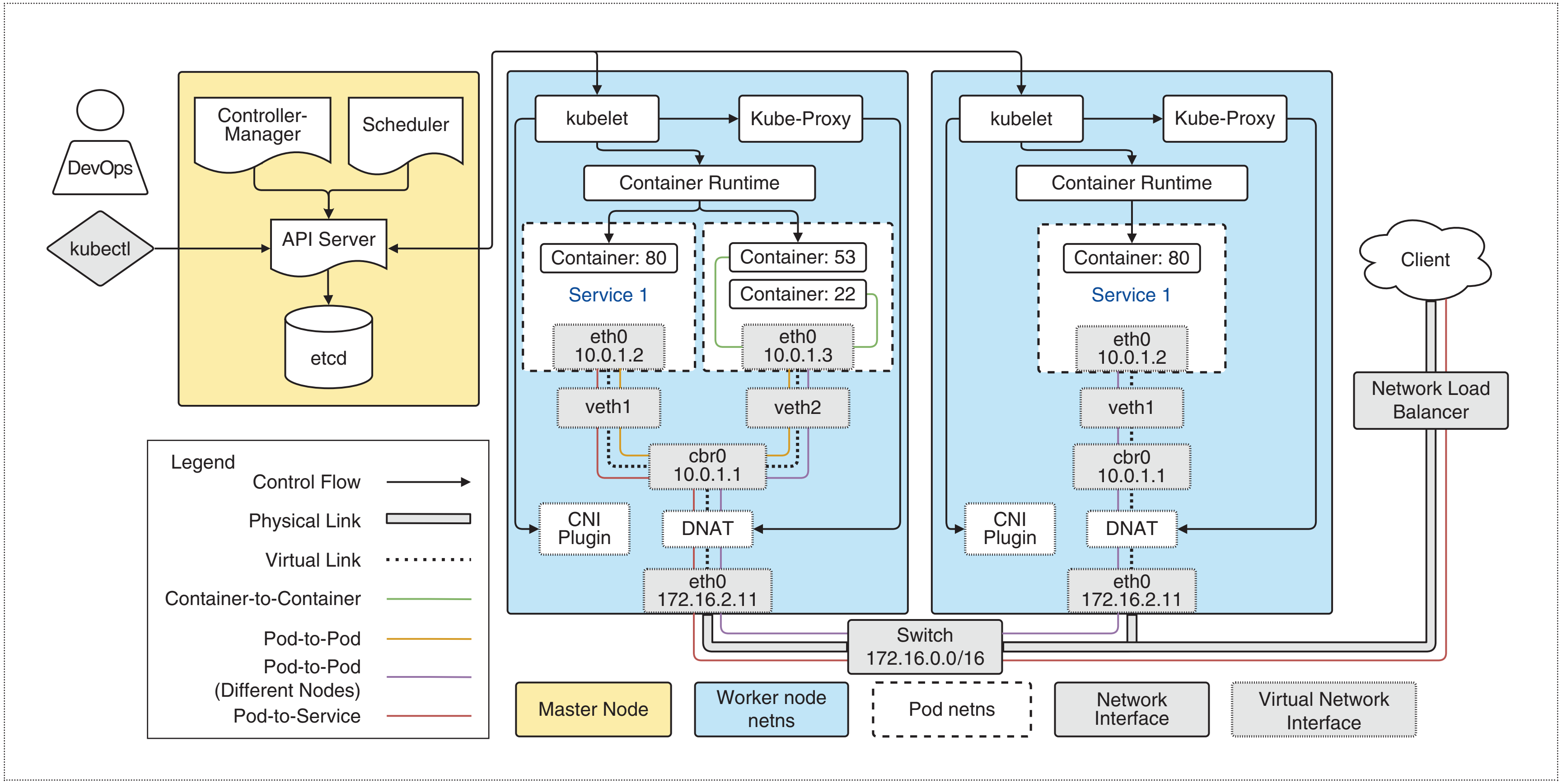


Figure 2. K8s "real" network and possible interactions between different services in a single cluster. This figure shows a more accurate representation of the actual network architecture of a single K8s-managed cluster from the top part of Figure 1. The key takeaway from a security standpoint is that all isolation boundaries (i.e., the namespace abstractions and network-security policies) are implemented in software; these boundaries disappear as soon as one escapes to the right namespace. For example, if an attacker breaks out of the pod netns of "Service 1" to the surrounding worker-node netns, the cluster is laid bare and, thus network-security policies no longer apply (see the "A Primer on Containers and Kubernetes" section). Alternatively, a compromised service could ignore the CNI plug-in and "talk" directly to any reachable interface. Even in the presence of some physical boundaries, an attacker could rewire the logical control flow (through kubelet on top) and provide backdoor access to workers that appear apparently separated.

when nodes are in different datacenters or clouds), in which case they, usually, use an overlay network. The CNI plug-in tracks which pods are on which subnets, and on which nodes, and updates the routing rules in the network namespace of each node such that the pod-to-pod traffic can be forwarded through the right node. Connectivity between nodes is, however, not managed by K8s, and we omit concerned scenarios, since they are beyond the scope of this article. Pod-to-pod communication on the same node is implemented via virtual Ethernet devices (veth pairs in Figure 2) and a bridge (cbr0 in the illustration). Therefore, multiple pods running on the same worker node can exchange network packets via the virtual bridge.

> “Therefore, multiple pods running on the same worker node can exchange network packets via the virtual bridge.”

When a container is compromised, CNI plug-ins using a bridge become vulnerable to common L2 network attacks [such as Address Resolution Protocol (ARP) and Domain Name System (DNS) spoofing]. Other plug-ins, instead of the bridge, use a virtual router in each node or IP in IP encapsulation to avoid such problems.

Pod-to-Service Networking

A K8s service is an abstract way to expose an application running on a set of pods. All pods used by an application share a common label that K8s uses for grouping the pods. K8s also uses labels to automatically keep track of newly instantiated pods and maintains a list of pod IP addresses associated with each service in an EndpointSlice resource. K8s supports three different types of services:

- The *ClusterIP* service assigns the concerned application a cluster-wide unique virtual IP address, only reachable from within the cluster.
- The *NodePort* service assigns the service to a static port on every node in the cluster. It can be accessed from outside the cluster using the node’s IP address and the statically assigned port number. K8s also routes requests to NodePort services to a clusterIP services (to load balance traffic across the pods).
- In the *Load Balancer* case, K8s exposes the service through a cloud-provider’s load balancer (red line in the Figure 2). Requests arriving at the cloud-provider’s load balancer are subsequently routed to a NodePort service, which in turn routes it to a ClusterIP service.

A Summary of Network-Security Implications

In this section, we highlight a list of network-security issues that may arise within a K8s cluster and that every K8s user and developer should keep in mind.

Pod netns by a Pause Container

The pod netns is held by a special container, called a *pause* container. Every container scheduled on a pod will share the netns with the pause container. Thus, escaping from the pod netns means escaping from the pause container netns, ending up in the host netns (the pause container is not shown in Figure 2, but the same can be thought as escaping from the pod square). An attacker who is able to get on the host netns can potentially see network interfaces, routing rules, other pods netns: if the attacker has privileged access, the worker-node netns is fully compromised.

CNI Plug-Ins Jeopardy

CNI plug-ins run as (privileged) programs on worker nodes. Subverting these objects automatically results in privileged access to the worker nodes, compromising the whole network. Also, an attacker can compromise the network interfaces or other components of the CNI plug-in itself. Layer 2 plug-ins that use the Linux bridge may be susceptible to man-in-the-middle (MITM) attacks (e.g., ARP spoofing and DNS spoofing); routing daemons of layer 3 plug-ins (e.g., CVE-2021-26928 affecting Border Gateway Protocol) and eBPF (e.g., CVE-2021—31440) may also be vulnerable.

Software Isolation of Resources

By default, the K8s network is flat. K8s isolates resources in this flat architecture through network policies, while also introducing new security implications. Within a cluster, network policies are enforced by the CNI plug-in and not K8s itself. Subverting the plug-in may result in invalidating all policies. The policies are also usually stored in the CNI-plug-in datastore (e.g., etcd): Compromising this database will result in another point of failure.

Network Policies Limitations

K8s base network policies that do not depend on the particular CNI plug-in do not support logs and drop/block options. There is no support for fully qualified domain name filtering in network rules, limiting the security options available. Furthermore, network policies offer protection for layer 3 network controls between pod IP addresses, but attacks over trusted IP addresses can only be detected with layer 7 network filtering, which requires additional components. Finally, to the best of our knowledge, there is no methodology or tool yet to automatically compare network policies with the business logic of applications, other than manually verifying them.

Multitenant K8s Clusters

CVE-2020-8554, affecting multitenant K8s clusters, is not fully patched in K8s yet. An attacker who has permission to create or edit services and pods can intercept network traffic from other pods or nodes, by creating a ClusterIP service with an arbitrary IP address to which traffic is forwarded. Some plug-ins offer few countermeasures, but an attacker might still be able to succeed.

Dynamic Nature of K8s Objects

It is a common practice to segment a network by assigning subnets and separating them with firewalls in between. This approach cannot cope with the ephemeral nature of K8s objects: network rules based on IP addresses are not very effective, since IP addresses of resources may keep changing; a classic firewall should at least define rules based on classless interdomain routing (CIDR) ranges and not specific addresses. Referring to Figure 2, a firewall has the same network visibility of the switch: It can be used to monitor ingress traffic, but it has no internal visibility over intrazones traffic.

Virtual Network Infrastructure

As we explained in §IV, the K8s network infrastructure is all virtual (i.e., software-defined via veth pairs or the Linux bridge), with no physical interfaces or cables connecting the different components. The attack surface and security issues of SDN have been widely studied (e.g., Dabbagh et al.[4] and Yoon et al.[5]), but, to the best of our knowledge, similar studies on the K8s network do not exist. As an example, the K8s master node, by default, is not replicated, which makes it a single-point-of-failure, affecting other components, like the API server, the controller-manager, the scheduler, and the etcd database. The database is also not replicated, by default: if it becomes unavailable, it may not be possible to retrieve network policies or other settings. An outage or attack on a single master node cluster would not stop the cluster from working, but the cluster itself would become unmanageable (i.e., it would be impossible to change configurations or create new objects).

Distributed Tracing Not Embedded

By default, K8s does not allow distributed tracing of resources usage or networking requests. Keeping track of these events, such as network traffic, system calls, and CPU and memory usages is useful both to identify attacks and to improve the overall performance of the cluster. As an example, sophisticated attacks consist of several steps (e.g., malicious network traffic, CPU overload, and mounting sensitive directories): resources tracing may allow detecting these steps or identifying attack patterns. As of today, correlation of data from different sources remains complex and has to be done with external tools.

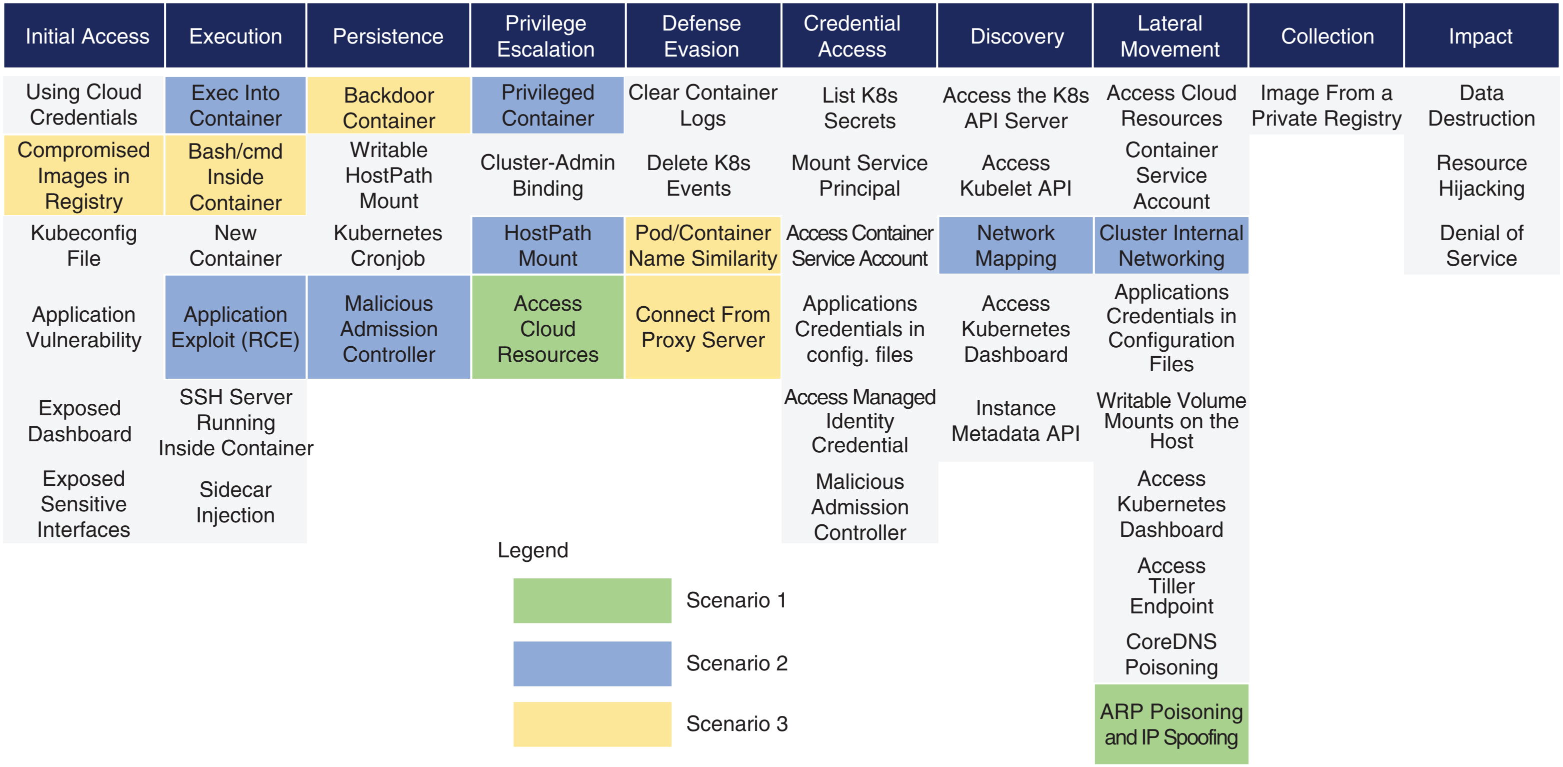
No Audit of the Level of Security of Policies

K8s does not automatically audit the security level of policies in a cluster and the potential risks and vulnerabilities that may result from them. In particular, authentication and authorization such as role-based access control (RBAC) and service accounts, secrets management, network policies, pod security policies, general policies handling the use of namespaces, and security options should be analyzed before deploying the cluster and exposing its services to the outside. As an example, sensitive files to audit include configuration files (/etc/K8s) of both the master and worker nodes and user-defined policies.

Mapping Attacks and Defenses

The ATT&CK (adversarial tactics, techniques, and common knowledge) framework, created by MITRE in 2013, describes common techniques used by attackers to gain access into a system as well as their behavior (e.g., lateral movement and privilege escalation) following the intrusion, based on real-world observations of attacks.MITRE also recently published the D&FEND framework for security defenses. The ATT&CK framework has also been specialized to address security threats relevant to containers.[11] Albeit MITRE has not yet released a more specialized K8s-related matrix, most K8s attack techniques can be mapped to the MITRE framework. Indeed, Microsoft published in April 2020 a K8s threat matrix[8] based on the structure of MITRE's ATT&CK framework that has been widely adopted to study and secure K8s deployments. Given prior collaborations between Microsoft and MITRE, and the overlap between the Microsoft and the MITRE ATT&CK matrices, we suppose that the Microsoft matrix will be included in the MITRE ATT&CK framework, in the near future. We opted, hence, to choose the Microsoft matrix to describe the attack scenarios, as shown in Figure 3. In Table 2, we also summarize the security issues, and propose solutions for hardening K8s deployments and link them to the MITRE D&FEND framework.

In this article, we analyzed the Kubernetes networking infrastructure, highlighting the key low-level abstractions, and offered a glimpse into the security implications



Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Impact
Using Cloud Credentials	🟦 Exec Into Container	🟨 Backdoor Container	🟦 Privileged Container	Clear Container Logs	List K8s Secrets	Access the K8s API Server	Access Cloud Resources	Image From a Private Registry	Data Destruction
🟨 Compromised Images in Registry	🟨 Bash/cmd Inside Container	Writable HostPath Mount	Cluster-Admin Binding	Delete K8s Events	Mount Service Principal	Access Kubelet API	Container Service Account		Resource Hijacking
Kubeconfig File	New Container	Kubernetes Cronjob	🟦 HostPath Mount	🟨 Pod/Container Name Similarity	Access Container Service Account	🟦 Network Mapping	🟦 Cluster Internal Networking		Denial of Service
Application Vulnerability	🟦 Application Exploit (RCE)	🟦 Malicious Admission Controller	🟩 Access Cloud Resources	🟨 Connect From Proxy Server	Applications Credentials in config. files	Access Kubernetes Dashboard	Applications Credentials in Configuration Files		
Exposed Dashboard	SSH Server Running Inside Container				Access Managed Identity Credential	Instance Metadata API	Writable Volume Mounts on the Host		
Exposed Sensitive Interfaces	Sidecar Injection				Malicious Admission Controller		Access Kubernetes Dashboard		
							Access Tiller Endpoint		
							CoreDNS Poisoning		
							🟩 ARP Poisoning and IP Spoofing		

Legend

🟩 Scenario 1

🟦 Scenario 2

🟨 Scenario 3

Figure 3. Mapping the example attack scenarios from Table 1 to the Microsoft K8s threat matrix[8] adapted from MITRE's ATT&CK framework.

Table 2. Main security challenges and mapping possible solutions in K8s networking to MITRE's D&FEND Framework.

Domain	Goal	Key Challenge	Tools (Control Knobs)	MITRE D&FEND
Traffic security	Protect the traffic through mutual authentication both at the control- and data-planes.	Automate the checking and enforcement of such practices both at cluster and service levels so as to reduce human errors and the time to deploy correct security policies.	• Built-in K8s Service Accounts—manage authentication and configuration of containers to access the K8s API server. • Built-in K8s Secrets—store encrypted confidential information within a nonpublic centralized repository. • (Centralized) Secret management to store and grant access to secrets. • Service mesh solutions—use mutual TLS for transport authentication, simplifying the management of keys, certificates and configurations.	• Message authentication • Message encryption • Disk encryption (e.g., for the datastore cluster) • Certificate analysis
Access control	Enforce fine-grained user- and application- access policies to clusters by providing least-privilege principles.	Provide the ability of administrators to define roles and manage control with greater granularity than what is allowed today.	• Built-in K8s RBAC—regulate permissions to access K8s resources (from pods to namespaces). • Service mesh solutions—include a decentralized authorization framework for communications among containers. • Identity providers tools—allow centralized management of roles and access policies, including reporting and auditing features. • Explicit rules inside clusters—restrict access between pods and allow communications only between permitted services.	• Mandatory access control (at network level) • Authentication event thresholding • Authorization event thresholding • Resource access pattern analysis
Traffic segmentation	Achieve defense-in-depth, limiting the lateral movements of a malicious actor.	Provide in a declarative way, the expected model of interaction between the deployed services and then enforce its configuration without breaking complex but desired networks paths.	• NetworkPolicy API—provide firewall-like declarative rules. It presents high complexity and a few limitations (such as the pod-level granularity of policies). • Namespaces—isolate the K8s API resources environments from each other. They do not have any effect on the isolation of network and cluster resources. • Service mesh–work at the application layer and can help to segment traffic; though cannot inspect the traffic. • Virtual networks rules inside clusters–explicit virtual network rules useful to segment traffic between pods.	• Broadcast domain isolation • Inbound/Outbound traffic filtering • DNS allowlisting/denylisting

(continued)

Table 2. Main security challenges and mapping possible solutions in K8s networking to MITRE's D&FEND Framework. (*continued*)

Domain	Goal	Key Challenge	Tools (Control Knobs)	MITRE D&FEND
Network visibility	Provide situation awareness at runtime to discover anomalous and suspicious behaviors.	Develop techniques such as model learning and fingerprinting at runtime represent promising yet not fully investigated approaches to identify drift behaviors	• Built-in K8s audit logger–audit the activities generated by users, applications using K8s API, and the control plane itself • Built-in K8s metrics-server–fetch individual container usage statistics • Open source monitoring and logging engines–aggregate and process logs and metrics from different sources providing a centralized repository for correlating events and detecting problems • Runtime security engines–detects unexpected application behavior and alerts on threats at runtime by monitoring system calls and comparing against security rules	• Administrative network activity analysis • Connection attempt analysis • DNS traffic analysis • Network traffic community deviation • Protocol metadata anomaly detection
Automated remediation	Automated triage and handling security alerts, incidents and policy violations.	Extend deployment automation to respond to security alerts (e.g., intentionally and proactively *stopping then restarting* containers)	• Open source monitoring and logging engines–providing alerts and warnings of runtime events • Complex event process–identify and analyze cause and effect relationship in real-time, connect with the automated actions to be performed • Automation engines and modules–define the playbooks managing how security events are handled and connecting to K8s nodes	• Administrative network activity analysis • Resource access pattern analysis • System call analysis
Compliance & audit	Follow industry best practices, standards, and internal security policies.	Extract evidence to demonstrate compliance, risks indicators, and the efficient automation of such checks and evidence collection process	• Benchmarks and best practice recommendations and certification schemes–a set of recommendations for configuring K8s in a secure way, including the certification of the cybersecurity of deployed services • K8s Audit logs–provide timestamped evidence on the state of the K8s cluster and related user interactions • Open source monitoring and logging engines–provide indicators on runtime events	• Software update • Disk encryption (e.g., for the datastore cluster) • Administrative network activity analysis • Resource access pattern analysis • System call analysis

of these abstractions. Understanding the design choices in implementing these abstractions[7] as well as their ramifications for security is a key first step toward securing a K8s (or any container-based) platform. We present a number of open challenges for the security community and hope that this article spurs the community to address them.■

Acknowledgments

We thank the reviewers and *IEEE Security & Privacy*'s Editor in Chief Sean Peisert for their comments that greatly helped to improve this article. Any remaining error is our fault. This work has received funding by the European Union under the H2020 grant 952647 (AssureMOSS).

References

1. Kubernetes. Accessed: June 15, 2021. [Online]. Available: https://kubernetes.io/
2. M. S. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman, "XI Commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices," in *Proc. IEEE Secure Development (SecDev 2020)*, 2020, pp. 58–64.
3. D. D'Silva and D. D. Ambawade, "Building a zero trust architecture using Kubernetes," in *Proc. 6th Int. Conf. Convergence Technol. (I2CT)*, 2021, pp. 1–8. doi: 10.1109/I2CT51068.2021.9418203.
4. M. Dabbagh, B. Hamdaoui, G. Mohsen, and R. Ammar, "Software-defined networking security: Pros and cons," *IEEE Commun. Mag.*, vol. 53, no. 6, pp. 48–54, 2015. doi: 10.1109/MCOM.2015.7120048.
5. C. Yoon et al., "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3514–3530, 2017. doi: 10.1109/TNET.2017.2748159.
6. J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *Proc. USENIX Annu. Techn. Conf. (ATC 2020)*, 2020, pp. 81–95.
7. AssureMOSS Kubernetes Security Testbed. Accessed: July 19, 2021. [Online]. Available: https://github.com/assuremoss/Kubernetes-testbed
8. "Secure containerized environments with updated threat matrix for Kubernetes," Microsoft, Mar. 23, 2021. Accessed: June 15, 2021. [Online]. Available: https://www.microsoft.com/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/
9. "Kubernetes documentation," Kubernetes. Accessed: June 15, 2021. [Online]. Available: https://kubernetes.io/docs/home/
10. R. Kumar and M. C. Trivedi, "Networking analysis and performance comparison of Kubernetes CNI plugins," in *Proc. ICAS 2019 Adv. Intell. Syst. Comput.*, vol. 1158, pp. 99–109, 2021.
11. "Containers matrix," MITRE. Accessed: June 28, 2021. [Online]. Available: https://attack.mitre.org/matrices/enterprise/containers/

Francesco Minna is a Ph.D. candidate at Vrije Universiteit Amsterdam, 1081 HV, The Netherlands. His research interests include cloud security and dynamic risk analysis for free and open source software. Minna received a double master's in cybersecurity in from the University of Trento and the University of Rennes. Contact him at f.minna@vu.nl.

Balakrishnan Chandrasekaran is a tenure-track assistant professor at the Vrije Universiteit Amsterdam, 1081 HV, The Netherlands. His research focuses on the performance and security aspects of networked systems. Chandrasekaran received a Ph.D. from Duke University. Contact him at b.chandrasekaran@vu.nl.

Agathe Blaise is a research engineer at Thales, Gennevilliers, 92230, France. Her research interests focus on cybersecurity, data analysis applied to networks, and programmable networks. Blaise received a Ph.D. in computer science from Sorbonne University. Contact her at agathe.blaise@thalesgroup.com.

Filippo Rebecchi is a research engineer at Thales, Gennevilliers, 92230, France. His current research interests are in software-defined networking, cybersecurity, and next-generation mobile networking. Rebecchi received a Ph.D. from Pierre & Marie Curie University. Contact him at filippo.rebecchi@thalesgroup.com.

Fabio Massacci is a professor at the University of Trento, Trento, 38123, Italy, and Vrije Universiteit, Amsterdam, 1081 HV, The Netherlands. Massacci received a Ph.D. in computing from the University of Rome "La Sapienza." He received the IEEE Requirements Engineering Conference Ten Year Most Influential Paper Award on security in sociotechnical systems. He participates in the FIRST special interest group on the Common Vulnerability Scoring System and the European pilot CyberSec4Europe on the governance of cybersecurity. He coordinates the European AssureMOSS project. He is a Member of IEEE. Contact him at fabio.massacci@ieee.org.