2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA) | 978-1-6654-1623-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/TPSISA52974.2021.00016

Security Challenges in the Container Cloud

Yutian Yang
Zhejiang University
Hangzhou, China
ytyang@zju.edu.cn

Wenbo Shen†
Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

Bonan Ruan
NSFOCUS Inc.
Beijing, China
ruanbonan@nsfocus.com

Wenmao Liu
NSFOCUS Inc.
Beijing, China
liuwenmao@nsfocus.com

Kui Ren
Zhejiang University
Hangzhou, China
kuiren@zju.edu.cn

***Abstract*—In recent years, containerization has become a major trend in the cloud due to its high resource utilization efficiency and convenient DevOps support. However, the complexity of container system also introduces attack surfaces. This paper aims to summarize security challenges in the container cloud. In particular, we first divide the whole container system into different layers according to their functionalities, including the kernel layer, the container layer, and the orchestration layer. We then summarize security-related technologies. After that, we discuss the security challenges for each layer. Finally, we present the current protection status for the container system and highlight future research directions. Our study shows that to improve the container cloud security, we need to design and implement more robust kernel isolation mechanisms, conduct systematic and thorough security analysis on existing container techniques, and develop comprehensive configuration checking tools.**

***Keywords*-Container security, Namespaces, Control groups, Docker, Kubernetes**

I. Introduction

Cloud computing has proliferated in recent years and has become a fundamental technology for IT companies. Cloud computing relies on sharing computing resources to achieve on-demand scaling, cost-saving, and maintenance reduction. In cloud computing, containerization is becoming a major trend for its convenience in DevOps and high resource utilization efficiency. In containerization, a self-contained user-space runtime environment is called a container. Compared to the virtual machines created by hardware virtualization technique, multiple containers run on the same shared kernel, eliminating the burden of maintaining standalone kernel and achieving high efficient resource utilization. The CNCF Survey 2020 released by the Cloud Native Computing Foundation shows that the use of containers in production has increased to 92%, and up 300% since 2016 [1]. All leading cloud vendors are providing container-based solutions, such as AWS Elastic Container Service and Azure Container Instances.

The architecture of the container cloud is complex and usually consists of multiple layers. More specifically, the bottom layer of the container cloud software stack is the operating system kernel, which manages hardware resources and provides isolation mechanisms for the upper layer. Above the kernel layer, the container layer manages container instances (i.e., user-space runtime environments) and container images. The top layer of a container system is an orchestration tool, which monitors workloads and adjusts the number of containers to achieve dynamic scaling.

Unfortunately, the complexity of the container cloud also introduces large attack surfaces. As the kernel is shared, a malicious container can exploit kernel vulnerabilities to attack the kernel as well as other containers. Moreover, bugs in the container layer allow the attacker to escape from the container. Besides, defects in orchestration tools give the attacker chances to control the whole container system. Different layers in the container cloud are facing different security challenges. While understanding those security challenges is critical for protecting the container cloud.

In this paper, we give a comprehensive study on the security challenges of a container system in the cloud. We first divide the container systems into different layers and introduce the functionalities of each layer, including kernel layer, container layer, and the orchestration layer. Next, we summarize the adopted security techniques of the container cloud. After that, we analyse the weaknesses and summarize the security challenges for each layer. Finally, we discuss the current status for protecting the container system and highlight potential future research directions.

To summarize, more research efforts are needed in the following aspects. First, stronger kernel isolation mechanisms are needed to isolate not only the resources, but also kernel vulnerabilities. Current sandboxed container and virtualized container techniques provide better isolation but with an extra performance overhead. Second, it is necessary to conduct a systematical security analysis on existing container techniques. In particular, kernel isolation mechanisms, such as namespaces and control groups (a.k.a., cgroups), need to be analyzed to fully understand their effectiveness. The container engine and runtime enforcement also need to be analyzed to reduce potential software vulnerabilities. Unfortunately, such analysis studies are still missing. Third, a comprehensive configuration checking tool is required to reduce configuration errors. The container system is complex, requiring proper configurations on multiple layers, which is error-prone. A comprehensive configuration checking tool can reduce most configuration errors and significantly improve the container system security.

†Corresponding author.

This work is partially supported by the NSF of China (Grants No. 62002317, 62032021, and 61772236), by the National Key R&D Program of China (Grant No. 2020AAA0107700), by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. 2018R01005).

978-1-6654-1623-8/21/$31.00 ©2021 IEEE
DOI 10.1109/TPSISA52974.2021.00016

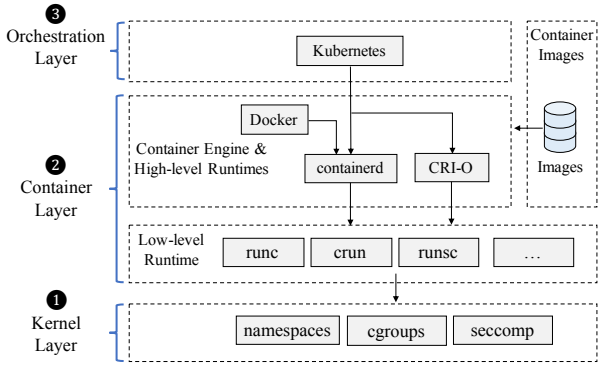

Fig. 1. Container cloud architecture.

II. Container Cloud Architectures

In this section, we introduce the architecture of the container cloud. As shown in Figure 1, the container cloud mainly consists of three layers. ❶ **Kernel layer** is at the bottom of the container architecture and serves as the foundation of the whole container system. In typical native container environments, multiple container instances run on the same shared kernel. While the kernel needs to isolate these container instances properly and coordinate their resource usages. Above the kernel, it is the ❷ **container layer**, including the container runtimes and container engine. Container runtimes are in charge of creating and managing the isolated execution environments (a.k.a., containers) using kernel mechanisms, such as Linux namespaces and cgroups. The container engine, such as Docker, is for user command interpretation, container image composition and management. On the top, we have the ❸ **orchestration layer**, including orchestration tools, such as Kubernetes, which automate the deployment and scaling of containers. Besides above three layers, the container cloud also needs to use container images. A container image is a collection of binaries, packages and other dependencies that are needed to deploy a container instance. The container runtime loads the container image into the isolated execution environment to build a container instance. To better understand the security challenges, we give the necessary background knowledge on those layers in the following.

A. Kernel Layer

In a typical container environment, multiple native containers run on the same shared kernel. The kernel needs to isolate and confine these containers. Traditional resource isolation and limitation mechanisms cannot meet the requirements of container systems. For example, the Linux rlimit confines resource consumption for each process, rather than for each container. Moreover, rlimit cannot account for critical resource usages like CPU and memory. It is also important to restrict system call accesses from containers; otherwise a container can stop another container by calling critical system calls like `reboot`.

Therefore, the Linux kernel introduces building-block features to support isolation and restriction that are necessary for container systems, including namespaces, cgroups, seccomp, and etc. Each of these features provides a functionality needed by container systems. For example, namespaces can be used to isolate the accessible file systems for different processes, while cgroups can be used to account for and limit the usages of CPU and memory. To restrict accessible syscalls for a process, one can use the seccomp. These features need to be combined together to construct a container, and the Linux kernel itself does not have the "container" concept. The job of creating a container is accomplished in the container layer.

B. Container Layer

As mentioned before, the container layer consists of container runtimes and the container engine.

1) Container Runtimes: Container runtimes utilize the kernel mechanisms to manage the full life cycle of containers, including creating, starting, and killing of containers. Most runtimes follow the runtime specification [2] released by Open Container Initiative (OCI), which specifies standard calling interfaces and configuration options so that a user can invoke different runtimes using the same interfaces and configurations.

Many prevalent runtimes, such as runc and crun, leverage namespaces to isolate containers. When creating a container, the runtime first uses namespaces to set up an isolated environment. The runtime then switches into the environment, configures cgroups, drops capabilities, and applies seccomp restrictions. Finally, the container startup process invokes `execve` to start the init process for the container. The Docker default capability and seccomp configurations can be found on Docker documents [3].

Note that different containers still share the same host kernel, leading to information leakages [4] and DoS attacks [5]. Improper configurations of container runtimes can even cause container escapes.

2) Container Engine: Container engines provide user interfaces and image management. They also invoke container runtimes to manage container life cycle. Image management includes building, packaging, transferring, and storing images. For the ease of image management, container engines provide `cp` interfaces to copy files from/to container images. However, malicious users can leverage malicious symbolic links to achieve arbitrary file accessing on the host. Details on container engine weaknesses are discussed in §IV-B.

C. Orchestration Layer

The orchestration tool is responsible for managing containers on a cluster. As mentioned before, it monitors the workload of the whole container cloud and dynamically increases or decreases the number of containers to achieve automate scaling. As Kubernetes has become the de facto orchestration tool, we use it as the example in the following sections.

As shown in Figure 2, the architecture of Kubernetes can be divided into the control plane and the data plane [6]. The

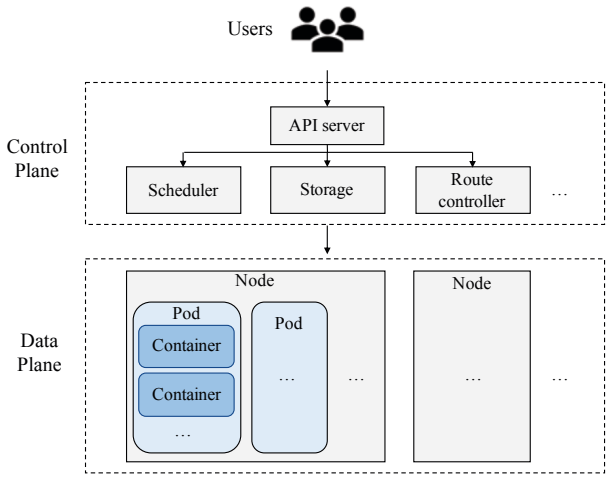

Fig. 2. Kubernetes architecture.

control plane consists of one or multiple controller nodes, such as the API server, the scheduler, the storage, and the route controller. The API server is the core of the control plane. It receives the REST requests from users and configures other components accordingly in the cluster, including control plane components and data plane pods. Therefore, it is critical that the API server authenticates and authorizes users properly, otherwise an attacker can compromise the whole cluster. For security consideration, it is also recommended to run the control plane on different nodes with the data plane.

The data plane receives commands from the control plane and carries out the jobs on data plane nodes. The nodes are actually physical machines or virtual machine instances. Each node can run multiple pods. A pod is the smallest deployable unit in Kubernetes, which contains one or more containers. Containers in the same pod share the same set of namespaces, cgroups, and filesystem volumes. Container life cycle management is delegated to the container layer. Therefore, the Kubernetes data plane faces the same security threats with the container layer. Moreover, data plane containers can be affected by compromised containers on the same node or even in the same cluster, and thus data plane container isolation as well as communication policies need be carefully configured [7].

D. Container Images

A container image is a collection of binaries and packages that are needed by the container instance. Most container images conform to OCI Image Format Specification [8], which specifies the standard image format. According to the specification, an image needs to contain at least a manifest, a configuration, and one or more filesystem layers. The manifest should declare high-level information on layers and the configuration, while the configuration records the detailed status of each layer. The configuration also includes other essential information needed by container engines, like environment variables and the image's entry point.

The content of an image consists of one or more layers. The bottom layer is the basic filesystem that the image is built upon. Each upper layer records the changed files compared to the layer below. For security reasons, all layers, except the topmost one, are set as read-only in current implementations of layered file systems, such as aufs and overlayfs. A container instance can only write to the topmost layer, which is discarded after the instance is killed. The restriction protects images from being manipulated by malicious containers.

The standardized formatting makes it easy to share and reuse images on the network. However, the convenience also brings the security risk that vulnerable or malicious images can propagate wider and faster. The details are discussed in §IV-D.

III. Container Adopted Security Techniques

In this section, we discuss the security techniques that have been adopted by container systems.

A. Namespaces

The Linux kernel provides namespaces to isolate the resource planes [9]. For example, mount namespaces provide isolation on mount points, which further enable the file system isolation. The isolation is implemented by providing different views on mount points for different namespaces. Consequently, one cannot access the file paths that are not mapped to the current mount namespaces.

The Linux kernel currently supports 8 types of namespaces, namely cgroup, ipc, network, mount, pid, time, user, and uts namespaces, which map to 8 types of resources. Each type of namespaces has one or more namespace instances. A namespace instance contains multiple processes, which share the same resource view. Note that the system processes are in the root namespace, which has the full view of the whole-system resources. The Linux kernel provides procfs interfaces for users to check namespace status. Using the pid namespace as an example, a user can access the `/proc/[pid]/ns/` directory to check the namespace IDs for the `pid` process.

B. Cgroups

Linux control groups (a.k.a., cgroups) are used to account for and limit resource usages [10], especially critical hardware resource usages. Cgroups currently have v1 and v2 implementation versions. Although cgroups v1 are still widely used, it is suggested to upgrade to the v2 version by the Linux community. The v2 version is empowered by newer techniques and offers a unified hierarchy organization for cgroup instances.

Cgroups currently supports 13 types of resources, including cpu, cpuacct, cpuset, freezer, perf_event, memory, hugetlb, rdma, blkio, pid, device, net_cls, and net_prio. Each resource type is managed by the corresponding cgroup controller. A cgroup user can choose to activate any of these 13 controllers to account for corresponding resources. For a specific type of resource, the cgroup instance accounts the total usage for all processes belonging to it. Multiple cgroup instances

are organized in a tree hierarchy. Cgroups also provide file-based user interfaces. Similar to procfs interfaces used by namespaces, cgroups have a cgroup `sys` filesystem. Users can manage cgroup instances, check resource usages, and set usage limitations via the cgroup `sys` file interfaces.

C. Seccomp

Seccomp is used to restrict syscalls for a process, including blocking sensitive syscalls and dangerous arguments [11]. It can be enabled by calling `seccomp` syscall on a specified process. Seccomp supports a strict mode and a filter mode. A process can only make `read`, `write`, and `_exit` syscalls when the strict mode is applied. On the contrary, the filter mode allows the user to specify the blocked syscalls and arguments. The filter mode is achieved by hooking syscall entries with BPF filters on the target process. When receiving a syscall from the target process, the kernel first traverses all installed filters to decide whether the syscall is allowed or not, before the syscall is actually handled. Once a process enters the seccomp state, it can no longer disable seccomp. Moreover, seccomp rules of a process also apply to all its children processes. Consequently, seccomp is suitable to restrict containers in the cloud.

D. Mandatory Access Control

The Linux kernel adopts mandatory access control (MAC) to enforce centralized and fine-grained access control. Compared the discretionary access control (DAC), MAC policies can only be modified by the system administrator and thus are tamper-proof. On the other hand, MAC policies are more fine-grained and can specify accessible permissions for each process on each resources. MAC is supported by the Linux security module (LSM), which provides configurable hooks on security-critical paths in the Linux kernel. Therefore, MAC is suitable to protect the cloud system against malicious containers. For example, MAC can enforce that a process within a container cannot access the files outside of containers, even the process has root privilege. Popular MAC frameworks include SELinux, AppArmor, and etc.

E. Sandboxed and Virtualized Containers

Sandboxed runtimes such as gVisor use a user-space kernel to further isolate containers. The user-space kernel handles most of the syscalls from containers and makes as few syscalls as possible on the host system. Using such a defense-in-depth strategy, sandboxed runtimes reduce the attack surface. Virtualized runtimes directly place containers in lightweight VMs. Compared with native runtimes and sandboxed runtimes, virtualized runtimes provide more robust isolation. A typical example of virtualized runtimes is kata-runtime. However, both sandboxed and virtualized runtimes introduce extra performance overhead compared with native runtimes. They also can be escaped [12] if configured improperly.

F. Role-based Access Control

The most widely used orchestration tool, Kubernetes, adopts role-based access control (RBAC) to authorize operations on objects from subjects [13]. Subjects can be users, groups, or service accounts, while objects can be any resources in the Kubernetes cluster including pods, nodes, logs, etc. RBAC uses `Role` and `ClusterRole` objects to represent the rights, including which objects are accessible and how they can be accessed. These role objects can be bound to subjects to achieve authorization. The bindings are specified by `RoleBinding` and `ClusterRoleBinding` objects. The scopes of `Role` and `RoleBinding` are restricted by Kubernetes namespaces [14], while `ClusterRole` and `ClusterRoleBinding` are cluster-wide. By adopting namespaced roles and bindings, the cluster administrator can achieve fine-grained access control. However, fine-grained access control also complicates RBAC rules configuration. As the cluster scale becomes larger, RBAC rules become error-prone and introduce security risks. The details are discussed in §IV-C.

IV. Container Security Challenges

As discussed before, a typical container system usually involves multiple layers, including the kernel layer, the container layer, and the orchestration layer. The container cloud security depends on the security of these layers, and it is challenging to secure all of them. In the following, we will summarize the security challenges of protecting each of them.

A. Kernel Layer Weaknesses

The shared kernel has become the main security concern of adopting native containers. It introduces three main security challenges: vulnerability isolation, resource confinement, and sensitive data protection. More specifically, the current shared kernel design of containerization cannot isolate kernel vulnerabilities. It cannot achieve robust resource confinement either, including both physical and abstract resource confinement. Moreover, the shared kernel design also introduces information leaks, allowing attackers to leak sensitive data or build covert channels.

1) Kernel Vulnerabilities: Container techniques are not designed for sandboxing and thus cannot isolate vulnerabilities. For example, Lin et al. demonstrated that 56.82% of vulnerability exploits could launch successfully from inside the container [15]. As a result, if the kernel has vulnerabilities, the attacker can exploit them easily from inside the container to compromise the host kernel. Therefore, native containers that run on the shared kernel, are vulnerable to kernel vulnerabilities.

2) Resource Confinement: Native containers share the hardware (a.k.a., physical) resources of the host system. Therefore, it is critical to confine and limit the usages of those physical resources for each container. To achieve this, the Linux kernel introduces the cgroups to limit the CPU, memory, and IO resource usages. However, Gao et al. showed that the out-of-band workloads can break the Linux cgroups' confinement [16].

Other than physical resources, researchers further identified that the kernel variables and data structures are also shared between native containers. These kernel variables and data structure instances are termed *abstract resources*. Compared with physical resources, abstract resources are more prevalent but under-protected. Therefore, Yang et al. proposed the abstract resource attacks, which attack other native containers on the same shared kernel by exhausting the shared abstract resources [5]. Their work demonstrated that abstract resource attacks affect all aspects of Linux kernel functionalities, including process management, memory management, storage management, and IO management. Moreover, most mainstream operating systems, including Linux, FreeBSD, and Fuchsia, are also vulnerable to abstract resource attacks.

3) Sensitive Data Leaks: Besides kernel vulnerabilities and weaknesses on resource confinement, the shared kernel may also leak sensitive information, leading to privacy leakage [17] [18]. Gao et al. revealed that the kernel leaks information via pseudo file systems, such as `/proc` and `/sys`. Based on the leaked information, they further demonstrated that the attacker can launch the synergistic power attack that might impact the reliability of data centers [4]. The authors further demonstrated that those information leaks can be exploited to infer private data, detect co-residence, and build covert channels.

Besides the traditional container environments, security researchers also found that the function-as-a-service (FaaS) platform may suffer from information leaks from the memory bus. They further designed covert channels between Amazon lambdas based on these information leaks to demonstrate the feasibility [19].

B. Container Layer Weaknesses

Similar to the kernel layer, the container layer also faces three security challenges: container vulnerabilities, resource confinement, and insecure configurations. In particular, severe CVEs have been identified in Docker engine, runC, containerd, and the virtualized container. Moreover, researchers revealed that the frequent communication between container components introduces resource exhaustion attacks. In addition, the insecure configurations in the container layer may also undermine the container security significantly.

1) Container Layer Vulnerabilities: Dozens of vulnerabilities have been disclosed in the container layer, many of which are severe enough that attackers could leverage them to escape containers.

Docker engine vulnerabilities. Docker copy mechanism caused multiple CVEs that can be triggered via `docker cp` command [20]. CVE-2018-15664 is caused by a time-of-check-to-time-of-use (TOCTOU) bug, which can be triggered by the symbolic link exchanges. CVE-2019-14271 is caused due to the loading of a malicious dynamic libraries within containers. Besides, another container engine named Podman was also vulnerable to the symbolic link-related issues in the copy mechanism (`podman cp`). All of above vulnerabilities could lead to container escapes.

Runtime vulnerabilities. The popular container runtime *containerd* mistakenly exposes containerd-shim API to containers that share the host network namespace. As a result, attackers with root privilege in such containers could execute arbitrary commands on the host via the containerd-shim socket and thus breaking the container isolation.

Another popular low-level runtime named runC also introduces a container escape vulnerability when resolving magic links. This vulnerability (assigned CVE-2019-5736) allows attackers in containers to open the `runc` or `docker-runc` binary on the host and overwrite it with arbitrary commands. Actually, the link resolution in container is error-prone, causing another vulnerability CVE-2021-30465 in directory traversing. By exploiting these CVEs, the attacker can easily escape from the container.

Kata container vulnerabilities. Virtualized containers leverage the hardware virtualization techniques to reinforce the container isolation. Therefore, people consider that they are more secure than native containers. Unfortunately, security researchers demonstrated that it is also possible to escape from the Kata virtualized containers.

More specifically, Kata containers do not enforce the device cgroups, giving the attacker chances to access the `/dev` files on the guest virtual machine, leading to CVE-2020-2023. The attacker can exploit this CVE to overwrite the underlying `kata-agent`. Moreover, the Kata container reuses the corrupted `kata-agent`, leading to CVE-2020-2025. In addition, `kata-runtime` does not check the validity of the mount point in the shared folder; it resolves any symbolic link and conducts the mount operation, leading to CVE-2020-2026. With these vulnerabilities, attackers within Kata containers could masquerade as `kata-agent` and create a symbolic link at the mount point. Root file system of the new container will be mounted to anywhere on the host pointed to by the symbolic link. In this way, the attacker can break the virtualized container, even though the hardware virtualization techniques are in use.

2) Resource Exhaustion Attacks: Besides the software vulnerabilities, the container layer also suffers from resource exhaustion attacks. Zhou et al. revealed that the inter-component communication in Docker is able to break the CPU usage constraint of containers and consume 7$\times$ of CPU than that is allowed [21]. Besides, Xiong et al. also showed that the IP addresses of the host machine in function-as-a-service (FaaS) platforms can be exploited to cause IP-blockages of benign functions [22].

3) Insecure Configurations: Container engines and runtimes could be configured with different options and arguments for various use cases. Some of these configurations might be sensitive and the mis-configuration introduces security problems. For example, Docker daemon configured to listen on remote TCP socket without authentication could be exploited by attackers to control the host by creating new malicious privileged containers. Daemon listening locally tends to be more secure, but it could be exploited by DNS rebinding or host rebinding [23] as well. Even in the case where Docker

daemon only listens on a local UNIX socket, if non-privileged users are added into the docker group, they could exploit this UNIX socket to escalate privilege by creating privileged containers.

Currently, Docker provides different kinds of sensitive configurations via `docker run` commands, as discussed below.

For runtime privilege and Linux capabilities. Docker supports four `docker run` commands for specifying runtime privilege and Linux capabilities. More specifically, `--cap-add` adds Linux capabilities while `--cap-drop` drops Linux capabilities; `--privileged` grants the running instances the root privilege; `--device` adds specified devices to the running container instance. Unfortunately, those commands can be easily misused to grant the container excessive privileges or capabilities, threatening the security of the container systems.

For runtime resources isolation. Docker also provides run commands for supporting Linux namespaces, including `--pid` for the PID namespace; `--uts` for the UTS namespace; `--ipc` for the IPC namespace; `--network` for the network namespace; `--userns-remap` for the user namespace. However, Docker allows a container to share the host namespaces by specifying the parameter as the host one (i.e., `--pid=host`). These sharing configurations bypass the isolation of Linux namespaces, undermining the container security.

For runtime resources constraints. Docker provides 20+ run commands to support Linux cgroups for resource constraints, including `--cpu-shares` for CPU shares, `--memory` for memory limit, `--device-read-bps` for read rate limit from a device, and `--device-write-bps` for write rate limit to a device. It is suggested to use as many cgroups as possible to defend against resource exhaustion attacks from malicious container users. Moreover, these cgroups also add another security defense layer against malicious container. For example, the Kata container did not enforce the device cgroups, giving the attacker chances to write to the hard drive of the guest virtual machine directly (CVE-2020-2023).

For security configuration. Docker provides `--security-opt` commands to allow users to specify additional security configurations for the container instances, including enforcing mandatory access control mechanisms, preventing from gaining new privileges and configuring seccomp profile. Those configurations are critical to the container security. Therefore, it is recommended to enforce those security configurations. However, Docker allows to turn off seccomp by specifying `--security-opt="seccomp=unconfined"`, which undermines the container security.

C. Orchestration Layer Weaknesses

The container orchestration tool faces security threats from multiple aspects, including vulnerabilities, insecure configurations, and weak network isolation. We focus on Kubernetes in this section.

1) Orchestration Tool Vulnerabilities: Vulnerabilities of orchestration tool leads to privilege escalation, container escape, and denial of services.

Privilege escalation vulnerabilities. The attackers may exploit vulnerabilities to execute any commands to escalate their privileges. For example, the API server mishandles the forwarding connections to backend, giving the attacker chances to execute arbitrary commands in the pods (CVE-2018-1002105). Moreover, an attacker might exploit vulnerabilities to escalate the privilege of one node to the privilege of the whole cluster. For instance, the API server fails to validate the redirection of upgrade requests correctly, leading to CVE-2020-8559. As a result, the attacker can redirect and execute commands on other nodes.

Container escape vulnerabilities. Vulnerabilities of Kubernetes may also lead to container escape. For example, the symbolic links are resolved on the host before mounted into containers, allowing the attacker to access directories outside of the container (CVE-2017-1002101). Actually, the patch [24] is not complete and leads to the time-of-check-to-time-of-use (TOCTOU) attacks. A new CVE-2021-25741 was assigned. Moreover, the `kubectl cp` command introduces the similar issues with `docker cp` command (discussed in §IV-B1), leading to CVE-2019-1002101, CVE-2019-11246, CVE-2019-11249, and CVE-2019-11251. By exploiting these vulnerabilities, malicious users in a container can create or overwrite files outside of the container.

Denial of service vulnerabilities. Kubernetes contains several denial of service (DoS) vulnerabilities. The Golang HTTP/2 implementations used by the API server is vulnerable to the ping flood and reset flood attacks, causing CVE-2019-9512 and CVE-2019-9514. Moreover, the API server does not validate the input YAML files properly. As a result, when parsing a specially-crafted YAML files, the API server consumes excessive amounts of CPU and memory, leading to DoS attacks (CVE-2019-11253).

2) Weak Network Isolation: It is challenging to secure network within a Kubernetes cluster. For example, in the multi-tenancy scenario, one compromised pod can send malicious traffic to other pods. Moreover, one compromised node could also send malicious traffic to other nodes. In addition, the cluster network is actually maintained by Container Network Interface (CNI) plugins in different network layers relying on different techniques. As a result, potential threats in all these layers and techniques might undermine the confidentiality, integrity and availability of the network in the cluster. It has been shown that clusters with CNI plugins working on data link layer (layer 2) are potentially vulnerable to ARP or DNS spoofing by default. Plugins utilizing BGP routing protocols suffer from BGP hijacking problem, which is a practical threat to a Kubernetes cluster [25].

D. Container Image Weaknesses

As shown in Figure 1, the container engine loads a container image to the runtime to initiate a container instance. As such, the security of the container environment also depends on container images.

1) Software Vulnerabilities: A container image is a collection of binaries, packages and other dependencies that are needed to deploy a container environment. However, these binaries and packages may contain vulnerabilities. For example, researcher revealed that both official and community Docker images contain more than 180 vulnerabilities on average when considering all versions [26]. Moreover, researchers also showed that the certified and verified repositories introduced by Docker do not improve the overall image security. While the average number of unique vulnerabilities found across repositories are expected to grow with 105 vulnerabilities per year [27].

2) Malicious Images: Beside the unintended software vulnerabilities, the container images may also contain the purposely-introduced malicious software. Security researchers have identified crypto-mining related malicious images on the Docker hub [28]. Researchers also detected 42 malicious images from the Docker hub using VirusTotal malware scanning.

V. Current Protection Status and Future Directions

In this section, we summarize current protection efforts on container security. We further analyze limitations of current protection and point out future research directions.

A. Isolating Kernel Layer

Besides hardening the kernel [29]–[31], researchers and developers in Linux kernel and the container communities have developed different schemes for isolating kernel weaknesses.

1) New Namespaces: Researchers have proposed the security namespace, allowing containers to have an autonomous control over their security mechanisms, such as Integrity Measurement Architecture (IMA) and AppArmor [32]. More recently, Linux kernel developer submitted the patches for a new kernel namespace mechanism named *CPU namespace*, which isolates CPU information by creating a scrambled virtual CPU map [33]. CPU namespace enables that every virtual namespace CPU maps to a physical CPU. Therefore, both the control and display interfaces are CPU namespace context aware. In this way, a process can only get resource view via a virtual CPU map, which reduces potential leaks from `/proc`.

2) Sandboxed Container: Besides kernel mechanisms, container community also proposed sandboxed container solutions to isolate kernel vulnerabilities. The most well-known sandboxed solutions is gVisor developed by Google. Instead of running containers on the shared kernel directly, gVisor runs a container on a dedicated user-space kernel, called Sentry. For security reasons, Sentry intercepts and handles most syscalls from containers. Therefore, Sentry reduces the syscalls that can be invoked on host kernel from containers, and thus reducing the attack surface. Such defense-in-depth strategy improves container security. However, Sentry still needs to invoke 50+ system calls to serve its needs, which undermines the isolation between Sentry and the host kernel.

3) Virtualized Container: Beside the sandbox container. container community also leverages hardware virtualization techniques to improve the overall security of container systems. One example is the Kata container, which runs the container instance inside a virtual machine instance [34]. The virtual machine guarantees the strong isolation while the guest kernel is pruned to reduce the performance overhead.

4) Future Directions: Currently, containers rely on Linux mechanisms, such as namespaces and cgroups, to isolate and limit resource usages. However, the security and effectiveness of those mechanisms have never been studied systematically. The prior work has demonstrated that the father PID namespace mirrors any PIDs in the children namespaces. As a result, the attacker can easily break the isolation of PID namespace and exhaust all PIDs on the shared kernel, crashing other containers [5]. Therefore, one should not blindly trust the container related kernel mechanisms. More analysis on namespaces and cgroups is required to fully understand their effectiveness.

B. Hardening Container Layer

Researchers and developers provide plenty of best practices for protecting the container layer.

1) Enforcing Least-privileged: As mentioned before, native containers share the host permissions. Therefore, to improve security, one needs to reduce their privilege as much as possible. As such, people proposed to enforce least privilege for the container. More specifically, it is suggested not to use the `--privileged` parameter when running a Docker container. Moreover, containers should drop as many capabilities as possible and only keep the capabilities that are necessary to the container functionalities. Also, the container should set `--security-opt=no-new-privileges` to prevent the container from obtaining any new privileges. By dropping capabilities, the container can block a considerable amount of kernel vulnerabilities [15]. In addition, for the necessary capabilities, the container should further leverage the user namespace to constrain the capabilities to its namespace only, rather than the root namespace on the host.

2) Adopting More Security Measures: Linux kernel provides 8 namespaces for resource isolation and 13 cgroups for resource constraint. Therefore, it is suggested to adopt as many namespaces and cgroups as possible to isolate and limit the container resource usage. However, current container engines only have partial support on Linux namespaces and cgroups, leading to security weak spots.

Moreover, the container can adopt mandatory access control mechanisms, such as AppArmor, to enforce customized and fine-grained access control for each container instance. In addition, containers should also use seccomp to block sensitive system calls to further improve the security.

3) Checking Configuration: Container developers also build tools to automatically detect the insecure configurations in the container cloud. For example, Docker provides docker-bench-security tool to check dozens of insecure container

configurations. The tests are automated. However, docker-bench-security tool requires the root privilege, cannot be used to check the non-root containers.

4) Future Directions: For functionality support, current OCI Runtime Specification still allows 14 capabilities, including dangerous ones, such as `AP_SETUID` and `CAP_SETFCAP` [35]. These capabilities might allow privilege escalation as discussed by grsecurity [36]. Therefore, whether those allowed capabilities will cause security problem is still an open question. More research works are needed to understand the security impacts of these capabilities.

Moreover, researchers have developed an open-source tool named Metarget [37], which could deploy most of aforementioned vulnerabilities in this paper. Metarget is helpful for security researchers to study the vulnerabilities and further come up with more robust defense solutions.

C. Securing Orchestration Layer

Researchers and developers have proposed hardening guidance and techniques to secure orchestration layer.

1) Secure Configuration: Many Kubernetes security best practices have been released to guide people to configure and use Kubernetes securely. Among which, *CIS Benchmark for Kubernetes* [38], released by Center for Internet Security (CIS), serves as one of the most detailed configuration guidance for Kubernetes. Besides, the recently published *Kubernetes Hardening Guidance* by NSA and CISA [39] gives practical measures on security of pod and cluster network, access control, audit and upgrading.

2) Network Reinforcement: In Kubernetes multi-tenant environments, both inter-node communications and even inter-pod communications on the same node could be malicious. To mitigate such potential attacks, one of the best practices is to follow the principle of least privilege and drop the `CAP_NET_RAW` capability of pods. Another method is to adopt novel technique to validate packet routes in the cluster [40]. In this way, man-in-the-middle attacks (MITM) like ARP spoofing could be mitigated. Besides, many open-source container network plugins (CNI), like Calico and Cilium, support fine-grained network policies, which makes it easy to control traffic between pods and network endpoints. Especially, based on the extended Berkeley Packet Filter (eBPF), Cilium supports flexible security actions on network traffic. Nam et al. also proposed and implemented a security-enforced network stack named BASTION, which provides visibility and control over inter-container traffic utilizing eBPF and XDP [41].

3) Future Directions: The pod security policy (PSP) in Kubernetes is hard to use and has been deprecated in Kubernetes 1.21 [42]. At the same time, new external controllers have been proposed, such as K-Rail, Kyverno and OPA/Gatekeeper. However, it still remains unclear whether those successors will be effective in the security enforcement of pods in Kubernetes.

D. Securing Container Images

Researchers have proposed different techniques to secure container images.

1) Image Scan: Researchers proposed to use images scan to reduce software vulnerabilities in container images. Open-source tools like Trivy [43], Anchore [44], and Clair [45] can be used standalone or integrated into the continuous integration process to scan container images. They provide detailed reports about how many vulnerabilities one container image contains. However, because of differences of vulnerabilities data sets, different scanners might generate different scan reports for the same container image. Berkovich et al. proposed a benchmark for evaluating image scanners [46], with which users could select the most suitable scanner for their environments.

2) Image Signature: Digital signature is a traditional but effective way to defeat tampering and ensure the integrity of container images. In a container environment, Docker content trust (DCT) [47] mechanism provides publishers with the ability to sign their images and users with the ability to verify images. The functions are implemented in Notary [48], an open-source project. Besides, developers also implemented an admission controller to integrate Kubernetes with the image signature functionalities, based on Notary [49].

3) Future Directions: Image debloating not only reduces image sizes and runtime costs, but also shrinks the attack surfaces, and thus becomes a promising technique. Several research works have proved its practicality. CDE [50] gives the first try to find dependencies of an application. It uses `ptrace` to dynamically probe the application's necessary resources. Cimplifier [51] leverages syscall logs to identify resources needed by different applications in a container. It then partitions the container into multiple smaller ones based on the analysis. However, these methods analyze runtime behaviors and may miss some dependencies. How to debloat container images reliably and effectively is still an open question.

VI. Conclusion

This paper presents a comprehensive study on security challenges of the container system in the cloud. In this study, we first divide the container system into three layers according to their functionalities, including the kernel layer, the container layer, and the orchestration layer. We then summarize security related techniques and discuss the security challenges for each layer. Finally, we present the current protection status for the container cloud and point out several research directions for the future work.

Our study shows that to improve the container cloud security, we need to design and implement more robust kernel isolation mechanisms, conduct systematic security analysis on existing container techniques, and develop comprehensive configuration checking tools. More research works are needed on those topics.

References

[1] C. N. C. Foundation, "Cloud native survey 2020," https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf, 2020.

[2] "Container runtime and lifecycle," https://github.com/opencontainers/runtime-spec/blob/master/runtime.md, August 2021.

[3] "Docker run reference," https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities.

[4] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2017, pp. 237–248.

[5] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma, and K. Ren, "Demons in the shared kernel: Abstract resource attacks against os-level virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer Communications Security, CCS 2021, November 15-19, 2021*. ACM, 2021.

[6] Kubernetes, "Kubernetes components," https://kubernetes.io/docs/concepts/overview/components/.

[7] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for inter-service access control of microservices," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[8] Opencontainers, "Oci image format specification," https://github.com/opencontainers/image-spec#oci-image-format-specification.

[9] "namespaces - linux namespaces," https://man7.org/linux/man-pages/man7/namespaces.7.html.

[10] "cgroups - linux control groups," https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/cgroup-v2.rst.

[11] "seccomp - linux seccomp," https://man7.org/linux/man-pages/man2/seccomp.2.html.

[12] Y. Avrahami, "Escaping virtualized containers," https://i.blackhat.com/USA-20/Thursday/us-20-Avrahami-Escaping-Virtualized-Containers.pdf, 2020.

[13] Kubernetes, "Using rbac authorization," https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

[14] ——, "Kubernetes namespaces," https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[15] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, p. 418–429.

[16] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2019, pp. 1073–1086.

[17] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2010.

[18] H. Li, Y. Yang, Y. Dou, J.-M. J. Park, and K. Ren, "Pedss: Privacy enhanced and database-driven dynamic spectrum sharing," in *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2019.

[19] A. Yelam, S. Subbareddy, K. Ganesan, S. Savage, and A. Mirian, "Coresident evil: Covert communication in the cloud with lambdas," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1005–1016.

[20] Y. Avrahami and A. Zelivansky, "In-and-out - security of copying to and from live containers," https://osseu19.sched.com/event/TLC4/in-and-out-security-of-copying-to-and-from-live-containers-ariel-zelivansky-yuval-avrahami-twistlock, 2019.

[21] T. Zhou, W. Shen, N. Yang, J. Li, C. Qin, and W. Yu, "Analysis of dos attacks on docker inter-component stdio copy," *Chinese Journal of Netword and Information Security*, vol. 6, no. 6, 2020.

[22] J. Xiong, M. Wei, Z. Lu, and Y. Liu, "Warmonger: Inflicting denial-of-service via serverless functions in the cloud," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer Communications Security, CCS 2021, November 15-19, 2021*. ACM, 2021.

[23] M. Cherny and S. Dulce, "Well, that escalated quickly! how abusing docker api led to remote code execution, same origin bypass and persistence in the hypervisor via shadow containers," https://www.blackhat.com/docs/us-17/thursday/us-17-Cherny-Well-That-Escalated-Quickly-How-Abusing-The-Docker-API-Led-To-Remote-Code-Execution-Same-Origin-Bypass-And-Persistence_wp.pdf, 2017.

[24] M. Au and J. Šafránek, "Fixing the subpath volume vulnerability in kubernetes," https://kubernetes.io/blog/2018/04/04/fixing-subpath-volume-vulnerability/, 2018.

[25] N. Chako, "Attacking kubernetes clusters through your network plumbing: Part 2," https://www.cyberark.com/resources/threat-research-blog/attacking-kubernetes-clusters-through-your-network-plumbing-part-2, 2021.

[26] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," ser. CODASPY '17, New York, NY, USA, 2017, p. 269–280.

[27] E. Socchi, "A deep dive into docker hub's security landscape-a story of inheritance?" Master's thesis, 2019.

[28] "Attackers found building malicious container images directly on host," https://www.infoq.com/news/2020/09/Malicious-Container-Images/.

[29] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1205–1220.

[30] J. Sun, X. Zhou, W. Shen, Y. Zhou, and K. Ren, "Pesc: A per system-call stack canary design for linux kernel," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 365–375.

[31] Q. Liu, C. Zhang, L. Ma, M. Jiang, Y. Zhou, L. Wu, W. Shen, X. Luo, Y. Liu, and K. Ren, "Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution," in *2021 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[32] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1423–1439.

[33] P. R. Sampat, "Introduce cpu namespace," https://lwn.net/Articles/872507/.

[34] Kata, "Kata container," https://katacontainers.io/.

[35] Docker, "Default capabilities," https://github.com/moby/moby/blob/master/oci/caps/defaults.go.

[36] grsecurity, "False boundaries and arbitrary code execution," https://forums.grsecurity.net/viewtopic.php?f=7&t=2522, August 2021.

[37] Metarget, "Metarget," https://github.com/Metarget/metarget.

[38] C. for Internet Security, "Cis benchmark for kubernetes," https://www.cisecurity.org/cis-benchmarks/, 2020.

[39] NSA and CISA, "Kubernetes hardening guidance," https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/1/CTR_KUBERNETES%20HARDENING%20GUIDANCE.PDF, 2021.

[40] K. Bu, A. Laird, Y. Yang, L. Cheng, J. Luo, Y. Li, and K. Ren, "Unveiling the mystery of internet packet forwarding: A survey of network path validation," vol. 53, no. 5, 2020.

[41] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 81–95. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/nam

[42] T. Sable, "Podsecuritypolicy deprecation: Past, present, and future," https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/.

[43] Aqua, "Trivy," https://github.com/aquasecurity/trivy.

[44] Anchore, "Anchore engine," https://github.com/anchore/anchore-engine.

[45] Quay, "Clair," https://github.com/quay/clair.

[46] S. Berkovich, J. Kam, and G. Wurster, "UBCIS: Ultimate benchmark for container image scanning," in *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/cset20/presentation/berkovich

[47] Docker, "Content trust in docker," https://docs.docker.com/engine/security/trust/.

[48] N. Project, "Notary," https://github.com/notaryproject/notary.

[49] S. S. E. GmbH, "Connaisseur," https://github.com/sse-secure-systems/connaisseur.

[50] P. J. Guo and D. R. Engler, "Cde: Using system call interposition to automatically create portable software packages." in *USENIX Annual technical conference*, 2011, p. 21.

[51] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.