NSFOCUS

About NSFOCUS

NSFOCUS is an iconic internet and application security company with over 18 years of proven industry experience. Today, we are operating globally with 2000+ employees at two headquarters in Beijing, China and 40+ offices worldwide including the IBD HQ in Santa Clara, CA, USA. NSFOCUS protects four of the ten largest global telecommunications companies and four of the five largest global financial institutions.

With its multi-tenant and distributed cloud security platform, NSFOCUS effectively moves security into the internet backbone by: operating in data centers around the world, enabling organizations to fully leverage the promise of cloud computing, providing unparalleled and uncompromising protection and performance, and empowering our partners to provide better security as a service in a smart and simple way. NSFOCUS delivers holistic, carrier-grade, hybrid DDoS and web security powered by industry leading threat intelligence.

NeuVector
Container Network Security

About NeuVector

NeuVector is the first company to develop Docker/Kubernetes security products. It is the leader in container network security and delivers the first and only multivector container security platform. NeuVector has committed to guaranteeing the security of enterprise-level container platforms, with its products applicable to cloud, multi-cloud, and on-premises container production environments. NeuVector provides in-depth runtime visibility into the container network, monitors "east-west" container traffic, performs proactive isolation and protection, and ensures the security of hosts and within containers. Through seamless integration with container management platforms, it achieves automation of application-level container security. NeuVector customers include global leaders in financial services, healthcare, publishing, and emerging Internet enterprises, and NeuVector partners with AWS, Docker, Google, IBM, Rancher, Red Hat, Aliyun and others. Founded by industry veterans from Fortinet, VMware, Trend Micro, Symantec, and Juniper, NeuVector has developed patent-pending behavioral learning, network security, data security, and container security.

2018 NSFOCUS Technical Report on Container Security

NSFOCUS

CONTAINER SECURITY

NSFOCUS Star Cloud Laboratory

October 2018

Catalogue

Preface ····· 1

1. Overview ····· 2

1.1 Container and Virtualization ····· 3

1.2 Evolutionary History of Containerization ····· 4

1.3 Container Security ····· 5

2. Container Basics ····· 8

2.1 Container Image ····· 9

2.1.1 What Is a Container Image? ····· 9

2.1.2 Characteristics of a Container Image ····· 9

2.1.3 Image Building ····· 10

2.1.4 Image Repository ····· 11

2.1.5 Use of Images ····· 13

2.2 Container Storage ····· 14

2.2.1 Image Metadata ····· 14

2.2.2 Storage Driver ····· 14

2.2.3 Data Volume ····· 15

2.3 Container Networking ····· 16

2.3.1 Underlying Technologies of Container Networking ····· 16

2.3.2 Host Networking ····· 18

2.3.3 Cluster Networking ····· 19

2.4 Container Management and Application ····· 23

2.4.1 Container Management ····· 23

2.4.2 Container Usage Scenarios ····· 30

3. Vulnerability and Security Risk Analysis ····· 35

3.1 Vulnerability and Security Risk Analysis ····· 36

3.1.1 Software Risks ····· 36

3.1.2 API Security ····· 39

3.1.3 Insecure Images ····· 43

3.1.4 Container Isolation Losing Effect ····· 44

3.2 Security Threat Analysis 45
3.2.1 Container Escape Attack 45
3.2.2 Container Network Attack 45
3.2.3 Denial-of-Service (DoS) Attack 46
3.3 Container Application Security Threat 47
3.3.1 Microservice Security 47
3.3.2 DevOps Security 47

4. Container Security Protection 49
4.1 Linux Kernel Security Mechanism 50
4.1.1 Kernel Namespace 50
4.1.2 Control Groups 50
4.1.3 Linux Kernel Capabilities 51
4.1.4 Other Kernel Security Features 51
4.2 Container Service Security 53
4.3 Host Security 54
4.3.1 Hardening of Basic Host Security 54
4.3.2 Hardening of Container-related Security 55
4.4 Image Security 56
4.4.1 Image Build Security 56
4.4.2 Image Repository Security 57
4.4.3 Image Scanning 58
4.4.4 Image Distribution Security 58
4.5 Container Network Security 59
4.5.1 Network Security Mechanisms 59
4.5.2 Security Protection for the Container Network 68
4.6 Runtime Security 71
4.6.1 Security Configuration for Container Launch 71
4.6.2 Runtime Security Monitoring and Audit 76
4.7 Orchestration Security 81
4.7.1 Kubernetes Security Protection 81
4.8 Application Security 84
4.8.1 Microservice Security 84
4.8.2 DevOps Security 86

5. Security Tools ····· 90

5.1 Open-Source Security Tool Kubernetes ····· 91

5.1.1 Kubernetes's Native Security Policies ····· 91

5.1.2 Istio ····· 91

5.1.3 Grafeas ····· 94

5.1.4 Clair ····· 95

5.1.5 CIS Benchmarks ····· 96

5.1.6 TUF and Notary ····· 99

5.1.7 SPIFFE ····· 100

5.1.8 Open Policy Agent (OPA) ····· 100

5.2 NeuVector ····· 101

5.2.1 Cloud-Native Container Deployment ····· 101

5.2.2 Container Environment Visibility ····· 101

5.2.3 Security Audit ····· 102

5.2.4 Multidimensional Protection ····· 103

5.3 StackRox ····· 104

5.3.1 StackRox Prevent ····· 105

5.3.2 StackRox Detect and Respond ····· 106

6. Summary ····· 109

7. References ····· 110

Special Statement

All data for analysis is anonymized and no customer information appears in this report to avoid information disclosure by negligence on our part.

Preface

The container technology realizes lightweight virtualization and isolation of resources by sharing the kernel of the host operating system. In recent years, it has been widely used in DevOps and microservices. While the container technology is widely accepted and used, the security of containers and their running environment calls for urgent researches and solutions. To further understand security threats against containers and container environments as well as provide security protection suggestions for container users, NSFOCUS Information Technology Co., Ltd. (NSFOCUS) joins hands with NeuVector, a famous container security company headquartered in the Silicon Valley, to release 2018 NSFOCUS Technical Report on Container Security. This report contains five chapters. Chapter 1 briefly describes the comparison between the container technology and virtualization technology, development of the container technology, and container security. Chapter 2 describes technical principles of containers, including container images, container storage, container networking and cluster networking, container orchestration platforms, and container application. Chapter 3 dwells upon security threats existing in containers and their running environments from the perspectives of software implementation, API design, container images, and container application. Chapter 4 details corresponding threat detection and security protection methods from the perspectives of the Linux kernel security mechanism, host security, image security, network security, and runtime security. Chapter 5 introduces some container security tool and products, including open-source plug-in tools (such as Clair and Grafeas) and standard products of NeuVector and StackRox. In the process of writing this report, we consulted a lot of materials and got many valuable opinions and suggestions from a lot of people. We would like to express our deep gratitude to them here. Many thanks to the great support from related units during the writing and publishing of this report! Any advice and criticism about this report are appreciated.

2018 NSFOCUS
Technical Report
on Container Security

1. Overview

1.1 Container and Virtualization 3
1.2 Evolutionary History of Containerization 4
1.3 Container Security 5

In recent years, the cloud computing model has gradually been universally recognized and accepted in the industry. In China, sectors such as governments, finance, carriers, and energy as well as small and medium-size organizations, to varying degrees, have migrated their business to the cloud. However, just turning hosts, platforms, or applications into virtual form cannot solve their legacy issues such as slow upgrade, clumsy architecture, and no support for rapid iteration. Then the concept of cloud native comes into being.

Cloud native encourages the agility, reliability, high resilience, easy scalability, and continuous updating of applications. During the building of cloud-native applications and service platforms, the container technology springing up in recent years, has become an important supporting technology for cloud-native application scenarios, thanks to its agility and resilience features and strong support from active communities. This chapter mainly describes fundamental concepts of the container technology and container security.

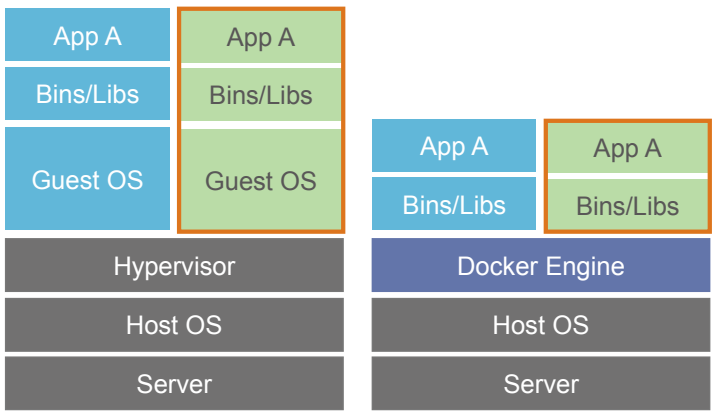
1.1 Container and Virtualization

One-to-many virtualization means to map one physical switch to multiple logical switches on a virtual tenant network for sharing among tenants.

Both virtual machine (VM) and containerization are system virtualization technologies to achieve system resource sharing through one-to-many virtualization. Compared with VM, containerization is lightweight. For example, VM is intended for virtualization of hardware resources at the hypervisor layer which serves as a platform to run VMs and manage VM operating systems as each VM has its own operating system, system library, and applications. Containerization, by contrast, is short of the hypervisor layer and each container shares hardware resources and the operating system with the host[1].

Containerization implements virtualization of computer system resources at the operating system layer. By separating, dividing, and controlling resources such as CPU, memory, and file system in the operating system, processes can utilize resources transparently. Figure 1.1 shows the difference in the implementation of VM and container technologies.

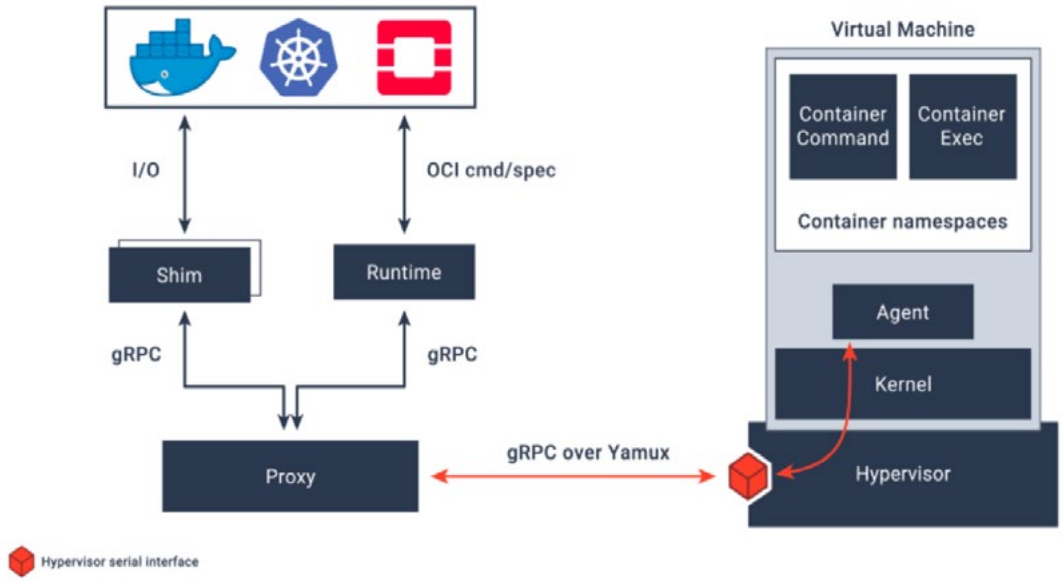
Figure 1.1 Architecture comparison between VM and containerization



1 Host machine, container host, and host mentioned in this report all indicate the host running the operating system kernel on which containers operate. Such a host can be either a physical server or a VM in the cloud environment.

As technologies advance and a balance is reached between business separation and performance indicators, containerization and VM tend to be fused. For example, the Kata Containers[1] project has containers embedded in lightweight VMs to achieve the container-level launch and running speeds as well as VM-level separation and security.

Figure 1.2 Architecture of the Kata Containers project



1.2 Evolutionary History of Containerization

The container concept can date back to chroot, a tool released for UNIX systems in 1979. Jails introduced in FreeBSD around 2000 can be deemed as one of earlier container technologies. Solaris proposed containers in 2004, bringing forward the concept of container resource management.

Arguably, Linux Containers (LXC) that appeared in 2008 is the first complete Linux container management solution implemented with Linux Control Groups (CGroups) and Linux Namespace. LXC, delivered in the liblxc library, provides API implementations of various programing languages and can run on the Linux kernel without any additional patches installed.

DotCloud (now Docker) open sourced its internal container project Docker[2] in 2013. Docker, based on LXC in early days, resorted to the self-developed libcontainer later. In addition to underlying services, Docker also introduces an ecosystem for container management which consists of a container image model, an image repository, REST APIs, and command lines.

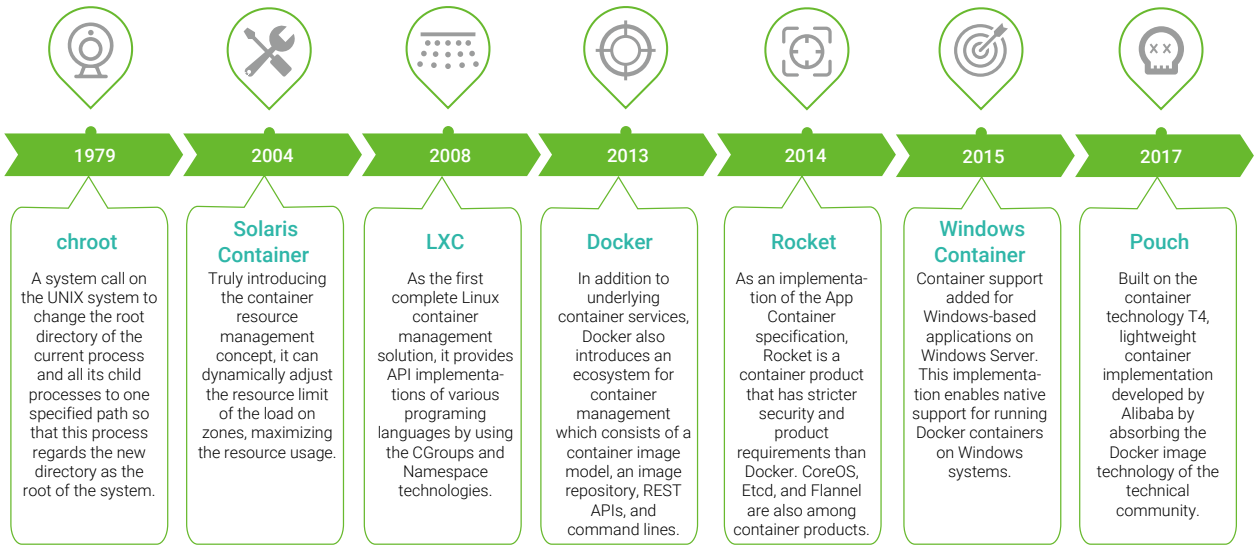
In 2014, CoreOS released a container engine, Rocket[3] (rkt for short), which is built on a more open standard container, App Container. Besides, CoreOS operating system, Etcd (a key-value storage component), and Flannel (a

networking component) are also among container products developed by CoreOS.

In 2015, Microsoft launched Windows Containers[4] to add container support for Windows-based applications on Windows Server. This implementation enables native support for running Docker containers on Windows systems.

In November 2017, Alibaba open sourced Pouch[5], a lightweight container technology licensed under the Apache 2.0 license agreement. Built on the container technology T4, Pouch gradually absorbs the Docker image technology and has been playing an important role in various scenarios in Alibaba, thanks to its high performance, high portability, and resource efficiency features.

Figure1.3 Evolution history of the container technology



1.3 Container Security

The container technology, though highly sought-after and having found a wide application, contains security issues that cannot be neglected.

> On June 14, 2018, a security vendor found 17 Docker container images infected with a cryptomining program. Worse still, these images had been downloaded more than 5 million times[6].
>
> According to a research report[7] released on June 18 by Lacework, a cloud security vendor, more than 21,000 container orchestration platforms were currently exposed on the Internet, including such widely used ones as Kubernetes, Docker Swarm, Mesos Marathon, and Redhat OpenShift. Of all those exposed platforms, Kubernetes account for 75% (for details, see NTI data described in section 3.1.2) and 305 exposed ones, even with no password at all, may lead to disastrous consequences once maliciously exploited.

First, let's take a look at Docker, the most popular container implementation technology within the industry. By July 31, 2018, a total of 38 vulnerabilities[8] had been found in Docker since it was released in 2014, including eight

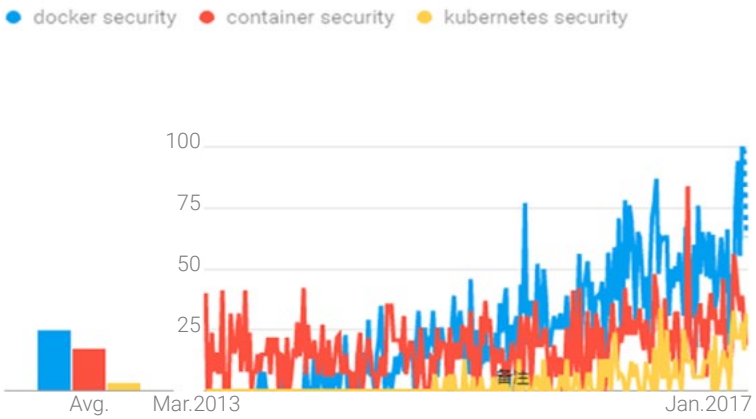
in 2018, five in 2017, and 11 in 2016. Of all those vulnerabilities, 33 are assigned a risk level, with four marked critical, eight identified high-risk, and only three labeled low-risk.

For example, the vulnerability which was found at the end of 2014 and assigned CVE-2014-9357 is deemed critical by Common Vulnerability Scoring System (CVSS) V2.0 which gives the composite score of 10. This vulnerability can bring such security risks as denial-of-service, access control privilege bypass, and arbitrary code execution.

In March 2017, Docker, Inc. announced the official release of Docker Enterprise Edition (Docker EE) which is 17.X (17.03.0-ee-1/17.03.1-ee-2 or 17.03.0-ce/17.03.1-ce) upgraded from 1.13.X. Docker, whether the community edition or enterprise edition, has security considerations added during the last updates.

For example, Docker EE 17.06.2-ee-6[9] released on November 27, 2017, fixes the execution state (moby/moby#35484[10]) of vulnerable versions, adds protection for health monitoring channels (moby/moby#35482[11]), as well as addresses the memory exhaustion issue (moby/moby#35424[12]) resulting from the daemon crash caused by abnormal images.

Figure 1.4 Keyword popularity trend on Google Trends



Global.Last 5 years.Web search

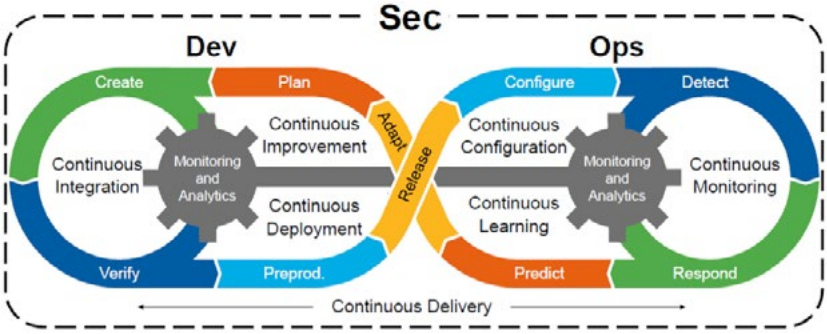
In addition, we have compared the search popularity of three keywords on Google Trends[1] i.e. "docker security", "container security", and "kubernetes security" in the past five years, as shown in Figure 1.4[2]. We can see that the search popularity of these keywords shows an upward trend, especially "docker security" for which the search popularity score was above 50 on average during the last year or two and even reached 100 in the period from February 4 to 10, 2018.

1 Google Trends is a kind of service developed by Google to analyze the popularity of search queries in Google. It can compare the search volume of different queries over time or allows users to see how popular a specific search term have been across regions and over time.

2 The y-axis indicates the search popularity of a keyword (relative to the top spot) at a given place and time. The popularity score is 100 for the most searched terms, 50 for the ones with half popularity of those searched most often, and 0 for such words without enough search volume.

In June 2017, Gartner released Gartner Identifies the Top Technologies for Security in 2017[13], which listed container security and DevSecOps (Development-Security-Operations) as two of the top security technologies. In its *2017 Cloud Security Hype Cycle*[14], Gartner pointed out that container security, though in the Innovation Trigger phase and not yet fully mature, developed at a rapid pace. Container security and security protection of the container ecology are both great security concerns of chief information officer (CIO) and chief information security officer (CISO) of enterprises which are set to deploy containers.

Figure 1.5 Closed loop of DevSecOps capabilities



As the container technology runs through the development, testing, and O&M phases of DevOps, security assurance is required throughout the lifecycle. For example, security assessment of open-source repositories and container images, orchestration security, and container runtime security are all on the to-do list of DevSecOps. Apart from that, container systems, applications, and networks in the actual operating environment also need security hardening, security detection, and security protection.

Containerization is essentially a virtualization technology at the operating system level. Once an attacker attacks a container by exploiting a kernel vulnerability in the operating system of a host, he or she will escape to the host to compromise other containers on it. In addition, the container system has a certain degree of insecurity and users tend to be short of aid from a professional security team when deploying and using a container. All these factors open up an opportunity of attack.

As containerization is a virtualization technology implemented between platforms and infrastructure, traditional cloud security solutions targeting infrastructure virtualization cannot solve all the above-mentioned security issues. For example, setting up a DevOps environment with containers as technical support requires a container security solution featuring a lifecycle starting from container image creation to production.

To sum up, security issues with containers and their operating environments need urgent survey and analysis. Having a thorough grasp of risks existing in container technologies and systems built on these technologies as well as appropriate coping policies is a primary premise of setting up a secure cloud-native environment.

2018 NSFOCUS
Technical Report
on Container Security

2. Container Basics

2.1 Container Image 9
2.2 Container Storage 14
2.3 Container Networking 16
2.4 Container Management and Application 23

2.1 Container Image

Images are the basis of containers. The container engine service can use different images to launch different containers. After a container becomes faulty, the service can be promptly restored by deleting the faulty container and launching a new one thanks to the underlying technique of container images[15].

2.1.1 What Is a Container Image?

A container image is a package comprising a file system encapsulated by layer and metadata that describes the image. It contains all systems, environments, and configurations necessary for the application, and the application per se. An image, after being created, is uploaded to an image repository. Users can obtain such image and use it to directly build their own applications.

The Linux Foundation sponsored the Open Container Initiative (OCI), which completed the first versions of the container runtime and image specifications in 2017[16]. Docker has made great contributions to the OCI by developing and donating a majority of the OCI code and has been instrumental in defining the OCI runtime and image specifications as a maintainer of the project.

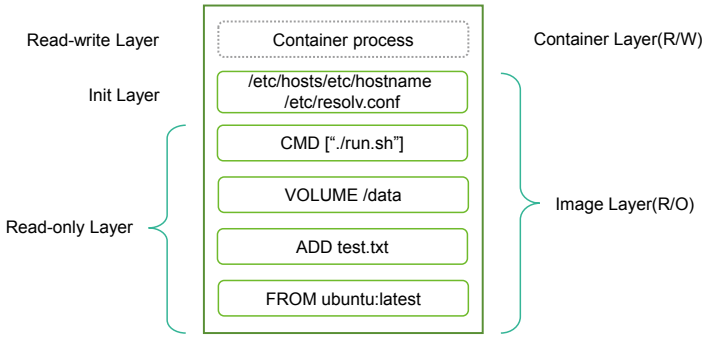
Compared with system images used by virtual machines (VMs), container images do not have the Linux kernel and they have a distinctive format. A VM image is a file encapsulated from an entire system, while a container image is not simply a file, but a file system featuring layered storage.

2.1.2 Characteristics of a Container Image

（1）Layered storage

Layered storage characterizes container images. As shown in Figure 2.1, each image is built up from a series of layers. When a file in an image needs to be revised, the operation is performed only at the uppermost read-write layer, without overwriting contents of the file system at lower layers. After being revised, the file needs to be submitted to generate a new image. In this case, only changes made at the read-write layer are saved, thus achieving the purpose of sharing the image layer between different container images. The following figure provides a rough illustration of a container image, where the uppermost layer is the read-write layer and other layers are read-only.

Figure 2.1 Container image structure



（2）Copy-on-write

Container images use the copy-on-write (CoW) strategy to share images between containers. With this strategy, a container, when launched, does not need a separately copied image file. Instead, all image layers are mounted to a mount point as read-only ones, which are overlaid with a read-write layer. When no change is made to the file, all containers have shared access to the exact same data. But when the file system is changed during container execution, the changes are written to the read-write layer and at the same time the earlier version of the file in the read-only layer is hidden. CoW, in combination with the layered mechanism, minimizes images' disk usage and containers' launch time.

（3）Content-addressable storage

Docker 1.10 introduces the mechanism of content-addressable storage (CAS) to allow retrieval of images and image layers based on file contents. Checksum calculation is performed for the content stored at the image layer to generate a hash value, which is taken as the unique ID of the image layer. This mechanism improves the security of images, and guarantees data integrity after *pull, push, load,* and *save* operations.

（4）Union mount

The union mount technique allows mounting of multiple file systems onto the same mount point. After the original directory at the mount point is integrated with the mounted ones, the final file system contains files and directories at all layers. A file system implementing the union mount technique is usually called a union file system (UFS).

Union mount is an approach to forming a union file system by mounting file systems of multiple image layers onto the same mount point. It can be seen as a method of amalgamating lower-layer storage drivers in a layered manner.

2.1.3 Image Building

Images, as a basis for container execution, can be obtained through various channels. One of them is to obtain existing images from image repositories, including public and private ones. Another method is to create a new image. Please read on to learn how to build and generate images in these two ways.

（1）docker build

The *docker build* command builds images automatically from a Dockerfile, which is readable and easy to understand. The mechanism goes as follows: Each line runs a corresponding modification command based on the upper-layer intermediary container before being submitted through the *docker build* command. These operations are repeated once and again until the desired image is produced. Following is an example of image building from a Dockerfile:

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

In a Dockerfile, the command (*COPY, RUN,* or *CMD*) at each line generates a new layer, which will overlay the file system previously generated with the last command. Finally, all image layers are combined to form a file system of the new image.

（2）docker commit

With this method, first, an existing image is used to launch the container; then, all required operations are performed in this container; finally, the *docker commit* command is executed on the host to package the container into a new image. This method is good in that it is convenient to make changes and easy to troubleshoot image faults. However, it also has disadvantages as the image building process is not transparent enough and images created this way are not easy to maintain.

2.1.4 Image Repository

An image repository is a place to store images. It is also an important channel to obtain images. Image repositories can be divided into public ones and private ones, depending on the usage.

（1）Public repository

A public repository is available for all Internet users. A typical example of public repositories is Docker Hub[17] , which currently houses more than 15,000 images. Most common applications have images stored here and can be directly downloaded for use.

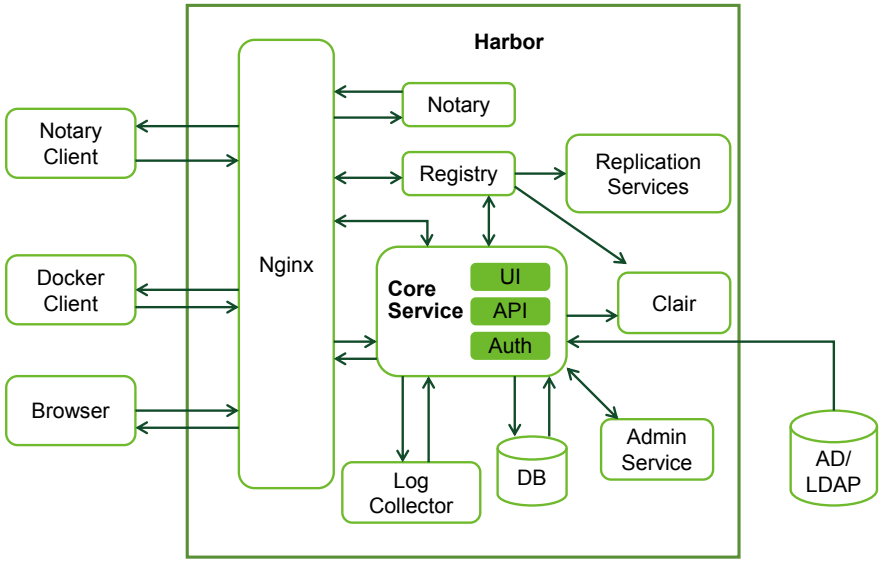
In addition, users can upload self-built images to a public repository for other people's use. Software vendors can also release their software in the form of images.

（2）Private repository

Not all images can be released and shared over the Internet. Sometimes, images can only be shared within a team because of containing sensitive information.

The sensitivity of images as well as the availability and stability of the Internet necessitates the use of private or local repositories. A private repository is one that can be accessed by a specified scope of users. A typical example of private repositories is Harbor.[18]

Figure 2.2 Harbor architecture



Harbor, developed by the R&D team of VMware China, aims to help users quickly set up an enterprise-grade registry service. Based on Docker Registry, Harbor provides such functions as management UIs, role-based access control, AD/LDAP integration, audit log, and image vulnerability scanning.

Table 2.1 Components of the Harbor service

Registry	Responsible for storing Docker images and processing Docker push/pull commands
UI	A graphical user interface to help users manage images on the Registry and to grant permissions to users
Job Service	Task management service of Harbor
DB	Database service that is responsible for storing data of user permissions, audit logs, image groups, and so on
AD/LDAP	Providing unified user identity authentication and permissions control
Log Collector	Responsible for collecting logs of other modules for future analysis
Notary	Auditing image contents to ensure the authenticity of images; optional for integration
Clair	Responsible for vulnerability scanning of images; optional for integration

2.1.5 Use of Images

Besides *docker pull/push*, there are other common commands in Docker for image-related operations, as listed below. For details, see the related official document of Docker [19].

(1) docker export

Exports a container to a local disk drive.

```
# docker export redis-server -o redis.tar
# ls
redis.tar
```

(2) docker import

Imports a local container as an image.

```
# docker import -m import-test-redis redis.tar
sha256:9cc818fc6b5d62f6d05fa0873885bb4b2cefa20aac5de0fb3cc7907086d166b7
# docker images | grep 9cc
<none>          <none>          9cc818fc6b5d          30 seconds ago          244MB
```

(3) docker save

Saves one or more images to a local disk drive.

```
# docker images
REPOSITORY      TAG             IMAGE ID                CREATED         SIZE
nginx           latest          3c5a05123222            4 days ago      109MB
redis           latest          71a81cb279e3            2 weeks ago     83.4MB

# docker save nginx redis -o test.tar
# ls
test.tar
```

2.2 Container Storage

2.2.1 Image Metadata

By default, in the Linux system, Docker data is stored in /var/lib/docker by default. However, different systems have different Docker storage drivers and directory structures.. This document uses Docker images in the OCI standard format as an example to describes how Docker images are stored.

Figure 2.3 Image storage directory

```
.
├── blobs
│   └── sha256
│       ├── 014246127c672bac4a8790acc1acba4fa356fd15b575d660c975acdd46e753de
│       ├── 01729050d692e70a9c961d7caeaa471aebb90185b574a9998d8b7292fce0de1d
│       ├── 3d77ce4481b119f00e53bee9b4a443469c42c224db954ddaa2e6b74cd73cd5d0
│       ├── 73674f4d94033a4285dde32367bb36d6287a12de1c1d272103c7d88f8feaa0e4
│       ├── 7cd2e04cf570947b4db6b63492d7a25772272107153e1cfe328abf5699d17d6b
│       ├── a4903a9d1f01852ddf36bb867aabdcc2ecdc8b28ca2db6f683c823e2088bef10
│       ├── ce7b0dda0c9f3a31a1965c920075d7dc128e6ed444f5a07e1d9bec2a2080625c
│       ├── d266646f40bd8883ddddb289a6379a9f38129afa5d9a205dafe464cbf3005847
│       └── e6cde63f2a108dbe06822548128f78d4240f98d26648cfb2172d18d0c35e287e
├── index.json
├── manifest.json
└── oci-layout

2 directories, 12 files
```

The hash value of the file content is taken as the unique ID of each image layer. After being obtained, the image is indexed as follows: Docker server reads the mainfests file of the image and locate the config file based on the sha256 code of config in mainfests. Then the server traverses all layers in mainifests to search for the image content locally based on the sha256 code and finally reassemble a complete image.

2.2.2 Storage Driver

Ideally, mount volumes are used to store high read/write directories, and data is seldom directly written into the writable layer of the container. However, in particular circumstances, data has to be written into the writable layer. In this case, a storage driver is required to work as a media between the container and the host. Docker uses the driving technique to manage images, and storage and interaction of container instances.

Currently, Docker supports five storage drivers: AUFS, BtrFS, Device Mapper, OverlayFS, and ZFS[20] . Any of those storage drivers cannot fit all application scenarios. Therefore, you need to select appropriate storage drivers according to the specific scenarios to maximize the performance of Docker.

Table 2.2 Comparisons of common storage drivers

	Characteristic	Advantage	Disadvantage	Application Scenario
AUFS	UnionFS Not merged into the mainline kernel File-level storage	AUFS, the first storage driver of Docker, features relative stability, wide application in production environments, and powerful community support.	As AUFS is a multi-layered system, copy-on-write (CoW) operations on big files from lower layers may be performed at a slow rate.	AUFS applies to scenarios characterized by high concurrency and low I/O counts.
OverlayFS	UnionFS Merged into the mainline kernel File-level storage	OverlayFS has only two layers.	OverlayFS always duplicates the entire file regardless of the size of the modified content. It takes longer to modify big files than to modify small ones.	OverlayFS applies to high concurrency and low I/O scenarios.
Device Mapper	Merged into the mainline kernel Block-level storage	Regardless of the file size, Device Mapper duplicates the block to be modified rather than the entire file.	Device Mapper does not support shared storage. If multiple containers try to read the same time simultaneously, multiple copies need to be generated. Enabling and disabling many containers can possibly cause hard disk overflow.	Device Mapper does not apply to I/O-intensive scenarios.
BtrFS	Merged into the mainline kernel File-level storage	BtrFS directly operates on underlying devices and allows to add devices dynamically.	BtrFS does not support shared storage. When multiple containers are reading one and the same file simultaneously, multiple copies need to be generated.	BtrFS does not apply to PaaS platforms with too many containers.
ZFS	All devices are aggregated into a storage pool for centralized management.	ZFS allows multiple containers to share a cache block and is applicable to environments with large memory.	CoW worsens the fragmentation. This leads to the discontinuity of files' physical addresses on the hard disk, making it difficult for Docker to read these files in sequence.	ZFS applies to PaaS-intensive scenarios.

2.2.3 Data Volume

Generally, stateful containers have a requirement of storing data persistently. As mentioned above, the modification of file systems occur in the uppermost read-write layer. In the lifecycle of a container, data is continuous, even after the container is disabled. However, after the container is deleted, the data layer is also deleted.

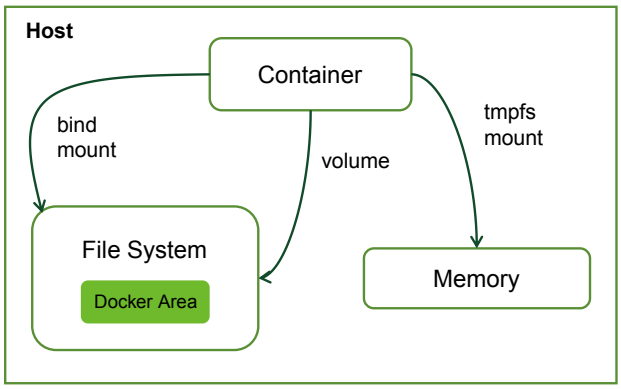
Therefore, Docker uses volumes for persistent storage of container data. Data volume is the preferred mechanism for persistent storage of data in Docker containers. Bind Mounts depend on the directory structure of the host, while data volumes are managed by Docker. Compared with bind mounts, data volumes have the following advantages:

- They can be easily backed up or migrated.
- They can be managed with Docker CLI commands or Docker APIs.
- They can be used on both Linux and Windows systems.
- They can be securely shared among multiple containers.

- Data volume drivers allow to store contents of storage volumes and encryption volumes onto a remote host or cloud, support the introduction of other functions.

In addition, volumes are a better choice for data storage than the read-write layer of a container. This is because volumes can be used for persistent data storage without increasing the container size, and therefore the storage is independent of the container lifecycle.

Figure 2.4 Mounting volumes on the Docker host



2.3 Container Networking

From the evolutionary history of cloud computing systems, the industry has reached a consensus that, while constant breakthroughs have been made to drive the maturation of computing virtualization and storage virtualization, network virtualization has lagged behind, becoming a major bottleneck that encumbers the fast growth of cloud computing. Such features as network virtualization, multitenancy, and hybrid clouds are posing brand new challenges of varying degrees to the security development of cloud networks.

The container technology provides lightweight virtualization capabilities, significantly reducing resource usage of instances and enhancing the performance of distributed computing systems. However, networking of distributed container systems remains a complicated link.

This part dwells upon container networking, which is divided into host networking and cluster networking.

2.3.1 Underlying Technologies of Container Networking

Container networks take various forms, but most use a fixed set of underlying technologies, including network namespaces, Linux bridges, and virtual Ethernet interface pairs (veth pairs).

2.3.1.1 Network Namespace

The network namespace technology is a technology for network isolation. A network namespace, after being created, has an independent network environment with such network resources as network interfaces, routes, and access control rules (iptables). The network of this namespace is isolated from other networks.

2.3.1.2 Linux Bridge

A Linux bridge is a virtual bridge in the Linux system. It connects network interfaces of different hosts to enable communication between hosts.

After Docker is started, a Linux bridge with the name of *docker0* is created by default. If no container has been built, the *docker0* bridge has no interface connection.

```
# brctl show
bridge name     bridge id               STP enabled         interfaces
docker0         8000.0242d1837f60       no
```

During creation of a container *(d458f9bd528)*, Docker creates a virtual network interface for it and connects it to the *docker0* bridge.

```
# brctl show
bridge name     bridge id               STP enabled         interfaces
docker0         8000.0242f22b2de4       no                  veth4d91464
                                                            veth6ed0a8c
```

2.3.1.3 Veth Pair

For communication with the host network and with the external network, the container needs to connect to a Linux bridge through a veth pair.

When launching a container *(d458f9bd528)*, Docker creates a veth pair, namely two interconnected virtual Ethernet interfaces. In this veth pair, one interface connects to the container and thus becomes its network interface card (NIC) *eth0;* the other connects to the *docker0* bridge. As such, packets within the container will first go through the veth pair of *eth0* and *veth6ed0a8c* successively before reaching the *docker0* bridge. In this manner, different containers on the same subnet can communicate with one another.

Run the following commands to obtain the index (peer-ifindex) of NIC *eth0* of the container:

```
# docker exec d458f9bd528 ethtool -S eth0
NIC statistics:
peer_ifindex: 37
```

Continue to run the following commands to find the name of the interface, whose peer_ifindex is 37, on the host:

```
# ip link | grep 37
37: vethfaa2a17@if36: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
docker0 state UP mode DEFAULT
```

Run *ethtool* to find the index of the peer interface:

```
# ethtool  -S vethfaa2a17
NIC statistics:
     peer_ifindex: 36
```

2.3.2 Host Networking

Take Docker for example. Currently, the Docker container host network comes in one of the following modes.

2.3.2.1 None Mode

In None networking mode, each container has its own network namespace, but has no network configuration. A container on such a network has only a *loopback* interface. The user has to add NICs and configure an IP address for the container.

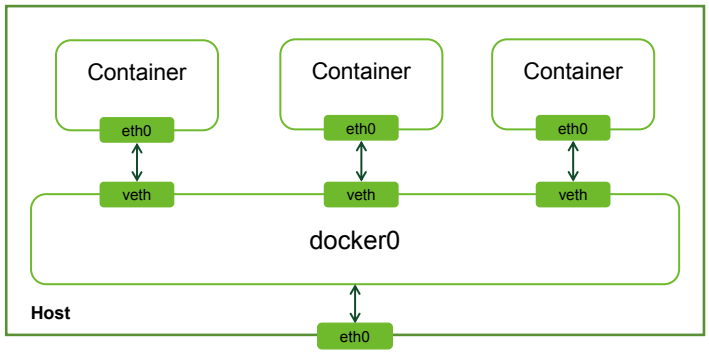
Rkt also supports the None networking mode, which is mainly used for container testing and then allocation of networks for containers as well as in scenarios where a high level of security is required and network connections are unnecessary.

2.3.2.2 Bridge Mode

The bridge mode features a single-host network achieved by using iptables for NAT and port mapping. Similar to NAT networking of virtual machines, this mode allows containers on the same host to communicate with one another, but does not allow external access to the IP address assigned to each container.

Bridge networking is the default type used by Docker. After being installed, Docker creates the *docker0* bridge by default and uses veth pairs to connect to containers and this bridge respectively. In this manner, all containers on the host are located on a layer 2 network.

Figure 2.5 Bridge networking



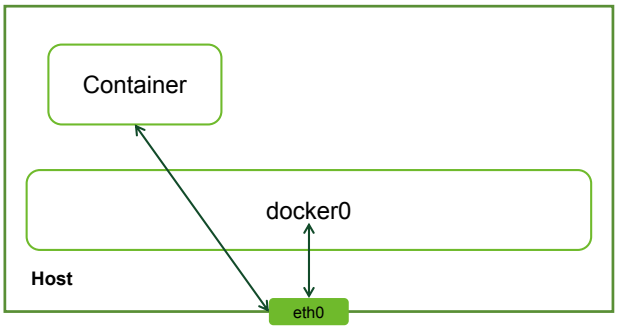
2.3.2.3 Host Mode

In host mode, the Docker service, when launching a container, does not create an isolated network environment for it, but adds it to the network of the host to share the host's network namespace (*/var/run/docker/netns/default*). The container shares the same network configurations (network address, routing table, iptables, and so on) as the host, and communicates with the external network via the host's NIC and IP address.

The container does not have an independent network namespace, and the port of the container service is directly exposed on the host, omitting the step of port mapping. Therefore, the port number of the container service cannot conflict with those already in use on the host.

A container created in this networking mode has access to all network interfaces of the host, but may not reconfigure the host's network stack unless deployed in privilege mode. Host networking is the default type used within Apache Mesos. In other words, if no network type is specified, a new network namespace will not be associated with the container, but with the host network.

Figure 2.6 Host networking

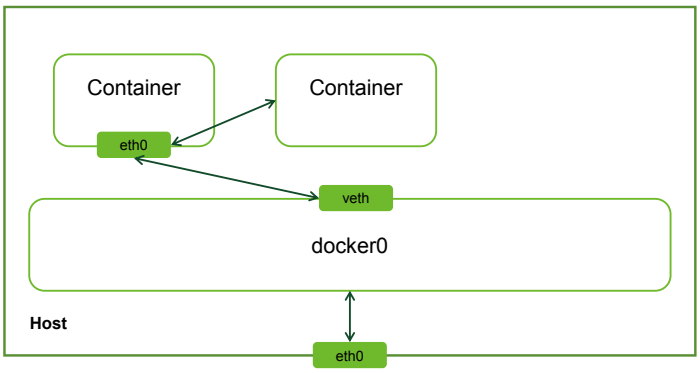


2.3.2.4 Container Mode

The container mode is a special networking type which features a namespace shared by a newly created container and an existing container. This new container does not configure its own NIC or IP address, but shares the IP address and port range with a specified container.

The two containers share data in network configurations only, with isolated file systems and process lists, among others. Their processes communicate with each other through the *loopback* NIC.

Figure 2.7 Container networking



2.3.3 Cluster Networking

This section takes Docker Swarm and Kubernetes as examples to illustrate how container cluster networking is implemented.

2.3.3.1 Docker Swarm

The docker_gwbridge network is a bridge network created when the Docker Swarm cluster is initialized. It is available on each host node in a cluster to enable communication between containers and hosts on which containers are running. To check the docker_gwbridge network, run the following commands:

（1）Docker_gwbridge network

The docker_gwbridge network is a bridge network created when the Docker Swarm cluster is initialized. It is available on each host node in a cluster to enable communication between containers and hosts on which containers are running. To check the docker_gwbridge network, run the following commands:

```
# docker network inspect docker_gwbridge
......
"Options": {
    "com.docker.network.bridge.enable_icc": "false",
    "com.docker.network.bridge.enable_ip_masquerade": "true",
    "com.docker.network.bridge.name": "docker_gwbridge"
},
.....
```

As the *_icc* configuration item is *false*, containers connecting to the *docker_gwbridge* bridge cannot communicate with one another when passing through interfaces on docker_gwbridge.

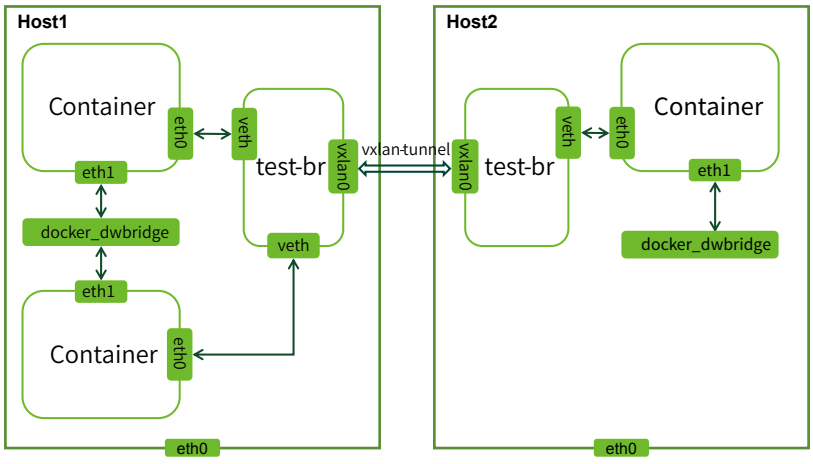
（2）Ingress network

The ingress network is created when the Docker Swarm cluster is initialized. It is available on each host node in a cluster to expose services to the external network and provide the routing mesh. For how an ingress network exposes services to the external network, see section 4.5.1.

（3）Custom overlay network

The custom overlay network is created after the Docker Swarm cluster is initialized and before services are created. It is mainly used for communication between containers on the same overlay network.

Figure 2.8 Container networking model for a service



In Figure 2.8, the three containers provide the same service. In Docker Swarm mode, these containers share the overlay network defined by the user. *test* is the custom overlay network. Containers on different hosts all connect to *docker_gwbridge*, whose interfaces, however, cannot be used for communication between containers on the same host. Communication between containers of the same service is achieved through the *test overlay* network.

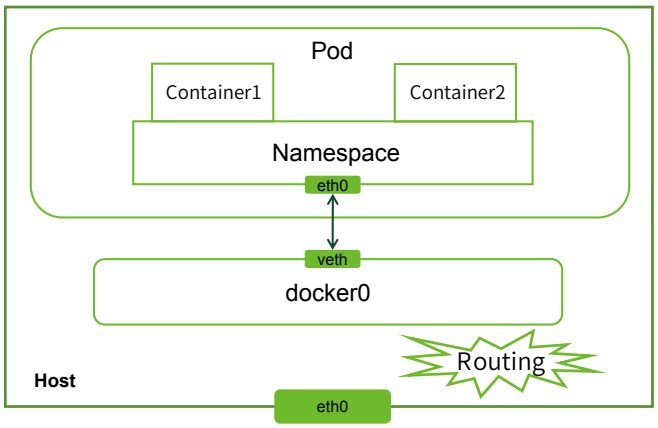
2.3.3.2 Kubernetes

Kubernetes features the design of pod objects, which correspond to logical hosts in a particular application. Each service is split by task and related processes are packaged into corresponding pods. A pod consists of one or more containers, which run on the same host and share the same network namespace and Linux protocol stack.

A Kubernetes cluster usually involves the following types of communication:

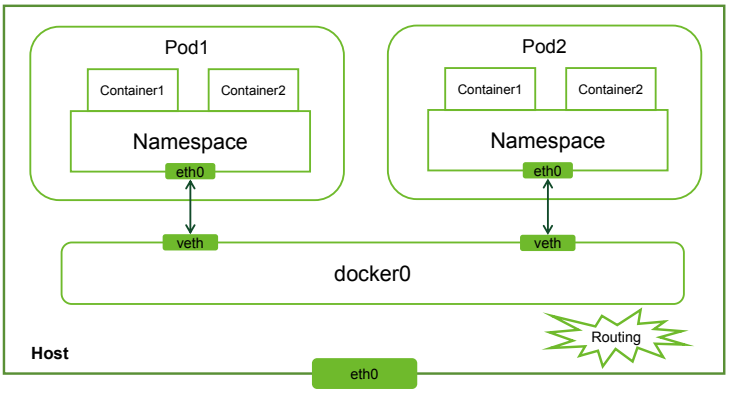
- Communication between containers in the same pod
- Communication between pods on the same host
- Communication between pods across hosts

Figure 2.9 Container networking model within a pod



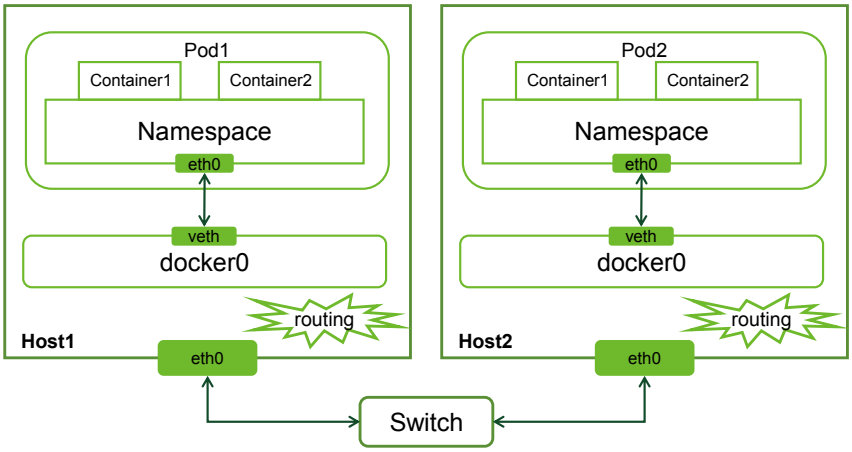
Communication between containers in the same pod is the most simple as these containers share the same network namespace and Linux protocol stack. As if running on the same machine, these containers directly use the local inter-process communication (IPC) mechanism of Linux for communication and their access to one another requires only localhost + port number.

Figure 2.10 Container networking model for different pods on the same host



Different pods on the same host connect to the same bridge (*docker0*) through *veth* and each pod dynamically obtains from *docker0* an IP address, which is on the same segment as the IP address of *docker0*. The default routes of these pods lead to the IP address of *docker0*. All non-local network data is transmitted to *docker0* by default for forwarding. This is equivalent to setup of a local layer 2 network.

Figure 2.11 Container networking model for different pods across hosts

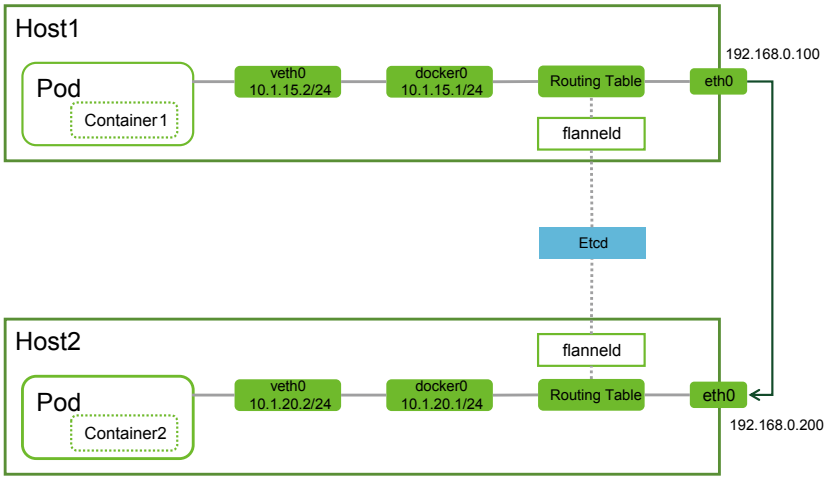


Communication between pods across hosts is relatively complicated. The IP address of each pod is on the same segment as that of *docker0* on the host that the pod is running. However, *docker0* and the physical network of the host are on different segments. Then how to ensure network data within a pod passes through the physical network to find the physical host address of the peer pod and reach the peer pod becomes key to successful communication.

For this reason, on a Kubernetes network, a module for global network address planning is required so that Pod1 can obtain the host IP address of Pod2 when sending data to the latter. The data sent by Pod1 goes through the *docker0* bridge of Host1 to the physical NIC *eth0* of Host1, and then, via a physical network, to the physical NICs *eth0* and *docker0* of Host2 before reaching Pod2.

In most private container cloud environments, using a third-party open-source network plug-in, such as Flannel[21] or Calico [22], to implement a cluster network is a common practice. Flannel is an overlay networking tool designed by the CoreOS team for Kubernetes to help each host that uses Kubernetes to own an integrated subnet.

Figure 2.12 Kubernetes Flannel network architecture



2.4 Container Management and Application

The container technology has been widely applied in microservices and cloud computing not only because it is lightweight but also owing to such important factors as flexible and agile management as well as orchestration system support.

2.4.1 Container Management

Clustering, flexibility, and agility characterize container application. How to effectively manage container clusters constitutes an important aspect of container technology implementation and application. Cluster management tools (orchestration tools) enable users to launch containers in clusters and help implement network interconnection besides providing load balancing capabilities and such assurances as extensibility, error tolerance, and high availability (HA). Examples of container cluster management tools that have currently drawn wide attention and seen extensive application include Kubernetes, Apache Mesos, Docker Swarm, and Docker Compose.

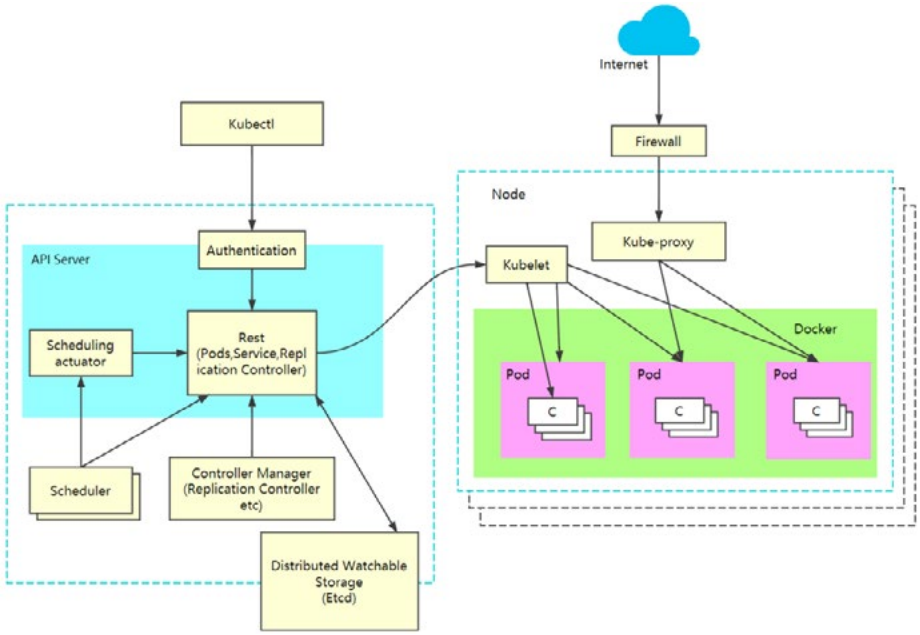
2.4.1.1 Kubernetes

Kubernetes[23] is an open-source distributed container management platform launched by Google based on its internal large-scale cluster management system Borg. It is also known as K8S, an abbreviation derived by replacing the eight letters "ubernete" with "8". Kubernetes delivers cluster management capabilities, multitenancy application support, transparent service registration and discovery mechanism, and load balancing, fault discovery, and self-healing capabilities.

The following figure shows the Kubernetes architecture, which is composed of one or more masters and one or

more nodes.

Figure 2.13 Kubernetes architecture



A master consists of multiple important components, for example, Etcd for storage, Scheduler for scheduling, HA component Controller Manager for cluster control, and API server for cluster coordination for external communication.

A node consists of two important components: kubelet that manages the lifecycle of pods in clusters and kube-proxy that is responsible for network configuration on Kubernetes.

Kubernetes contains the following **critical operation units:**

- **Pod:** the smallest unit that runs applications or services on Kubernetes. It is designed to provide support for multiple containers in a pod to share a network address and file system.
- **Service:** a proxy abstraction service for access to pods, mainly used for service discovery and load balancing within the cluster.
- **Replication Set:** scales the number of pod replicas.
- **Scheduler:** scheduling controller for resource objects in the cluster.
- **Controller Manager:** manages and synchronizes resource objects in the cluster.
- **Etcd:** distributed key/value pair (k, v) storage service to store status information of the entire cluster.
- **Kubelet:** pod lifecycle event generator.
- **Labels:** key/value pairs that are attached to objects to specify identifying attributes of objects.

- **Deployment:** pod object manager that integrates go-live deployment, rolling update, replica creation, go-live task suspension/restoration, and rollback functions.
- **Volumes:** directories that store data. At the time of container launch, the specified parameters are automatically mounted within the container. Volumes can be static and dynamic, with different lifecycles.
- **Stateful Set:** Generally, a newly created pod is stateless, resulting in the failure to find the previously mounted volume after the pod hangs and is restarted. To resolve this issue, the Stateful Set is used to retain the pod status.

The workflow of a Kubernetes pod is described as follows:

（1）Submit a request.

A user submits a YAML file to the API server, requesting creation of a pod. The YAML file contains details about this pod, including the number of replicas, images, labels, names, and exposure of ports. Upon receipt of such a request, the API server saves spec data in the YAML file to Etcd.

（2）Allocate resources.

The scheduler regularly listens for resource changes (that is, for pods created in the first step, which are waiting for allocation) in the Etcd database through the watch interface of the API server. When detecting a new pod, the scheduler selects an eligible node according to the scheduling policy, binds the pod to this node, and at the same time updates pod allocation information in the Etcd database.

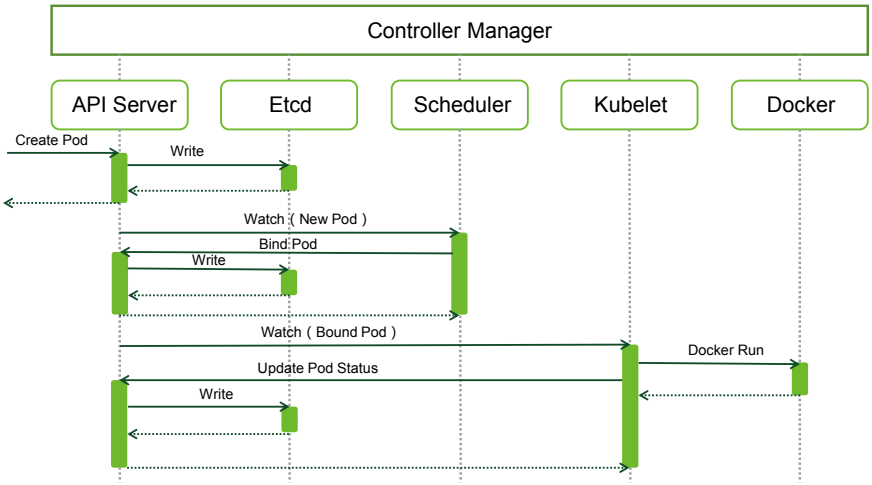
（3）Create containers.

Kubelet on the target node detects new pod allocation information in the Etcd database through the watch interface of the API server. Then it passes data of this pod to the container runtime for handling of the lifecycle of the pod. Later, Kubelet obtains pod status information from the container runtime and sends the updated status information to Etcd via the API server.

（4）Synchronize the resource status.

To ensure that this pod runs properly on the node (a pod may be killed for particular reasons), the Replication Set component in the Controller Manager regularly listens for the latest status of the pod in Etcd through the API server before synchronizing the number of replicas in this pod. In this manner, the number of running replicas is always the same as that of replicas specified by the user.

Figure 2.14 Kubernetes's pod creation flowchart

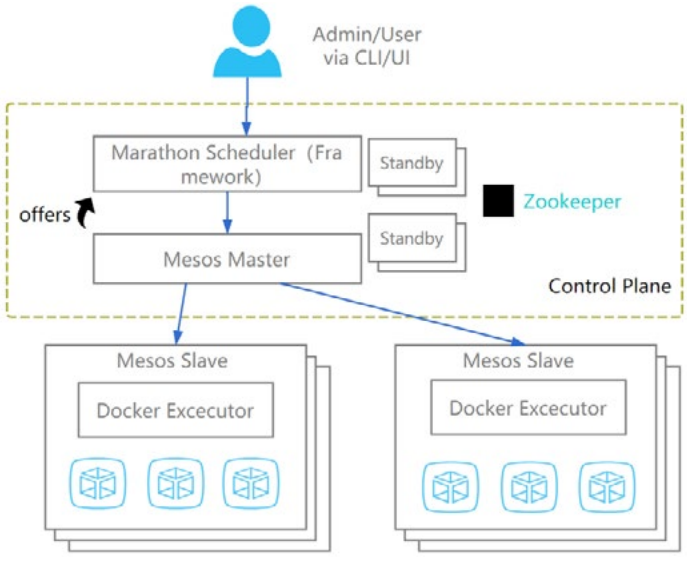


2.4.1.2 Apache Mesos

Apache Mesos[24] originated from an open-source project of UC Berkeley for abstracting and managing cluster resources and later has been widely used at Twitter. Mesos abstracts, manages, and schedules resources across the data center. With functionality similar to a host operating system, it enables multiple applications to run in a cluster and share resources.

The following figure shows the Mesos architecture, which consists of one or more master nodes, one or more slave nodes, and a number of framework applications.

Figure 2.15 Architecture of Apache Mesos



The master node is responsible for managing frameworks and slave nodes and allocating resources on slave nodes to associated frameworks. The slave node manages local task assignments. The framework is an external

computing application that is hot-swappable. Common frameworks include Marathon, Hadoop, and MPI. After registration, frameworks can connect to Mesos for unified management and resource allocation.

A framework mainly has two components: scheduler and executor. The scheduler interacts with the master node and schedules tasks to slave nodes according to the availability of resources. The executor obtains variables from the framework and executes tasks on slave nodes.

The workflow of Mesos is described as follows:

（1） Aggregate resource information.

The slave node uploads information about available resources on itself to the master node, which, as a resource pool, maintains collected resource information.

（2）Assign tasks.

The master node allocates available resources to the framework scheduler of an application. The scheduler checks whether such resources are eligible to support task execution and, if not, notifies the master node that it rejects the allocated resources (in this case, the master node allocates the resources to other frameworks); if yes, accepts such resources.

（3）Dispatch tasks.

The framework scheduler dispatches the task to the master node, which accepts it and uses the executor to run the application on the slave node.

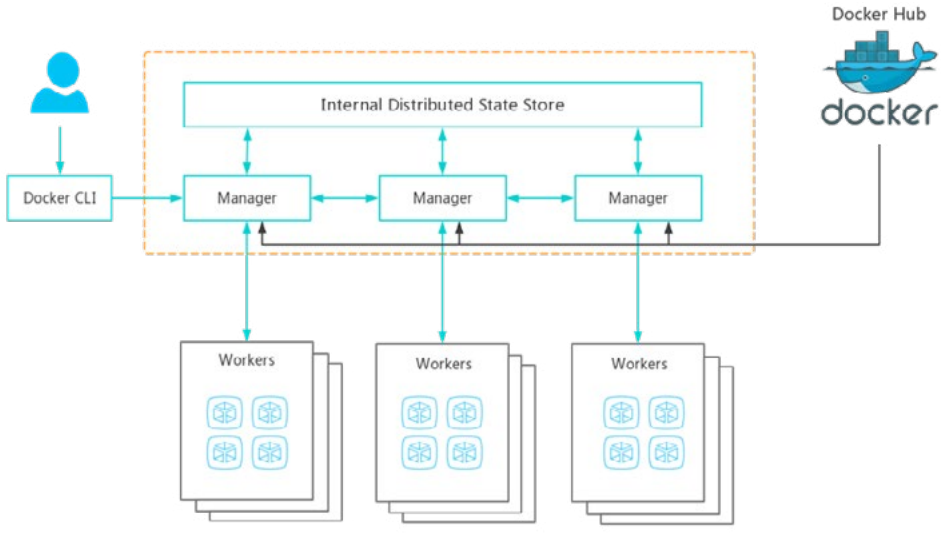
2.4.1.3 Docker Swarm

Docker Swarm[25] is a container management tool launched by Docker in December 2014. With quite simple functions, it is mainly used to manage Docker clusters, exposing the latter to users as a virtual whole.

Docker Swarm, as a component of Docker, shows its biggest advantage in the integration of its own API into the Docker API, making it easy for developers to integrate Docker applications with Docker Swarm. Currently, Rackspace and other platforms have adopted Docker Swarm. Users also find it easy to use Docker Swarm on public cloud platforms such as Amazon Web Services (AWS).

Docker Swarm consists of manager and worker nodes, as shown in Figure 2.15. The manager node receives user requests and assigns tasks to worker nodes. In addition, it can also perform duties of worker nodes. Worker nodes receive and execute tasks scheduled by the manager node.

Figure 2.16 Architecture of Docker Swarm



All nodes have basic Docker execution components, namely Docker daemon and load balancer, which provide an environment for the running of containers. Some important components for master and slave nodes are listed as follows:

- **Master node:** Manager CLI (Swarm command-line tool), Scheduler, and discovery service
- **Slave node:** Swarm Agent
- **Service discovery component:** The physical and communication environment in the cluster is complicated, resulting in inaccurate prediction of service interruption. This component provides service assurance for the cluster. When a slave node is down, the service discovery component can immediately detect it and instructs the scheduler to restore the failed service on another node.

The workflow of Docker Swarm is described as follows:

（1）Cluster operation

A user uses the Manager CLI of the master node to plan application deployment.

（2）Application distribution

The scheduler distributes the application within the cluster. The Swarm Agent is responsible for communication between nodes in the cluster.

（3）Service HA

Docker Swarm uses the service discovery component to maintain information about applications running on slave nodes, ensuring that the number of stably running applications in the cluster meets the related requirement to achieve HA of services.

2.4.1.4 Docker Compose

Compose[26] is another orchestration tool launched by Docker for users to define and deploy application services. Note that Compose only supports deployment of applications on a single node. If orchestration is required for a cluster, the Docker Swarm should be used to deploy applications in a cluster.

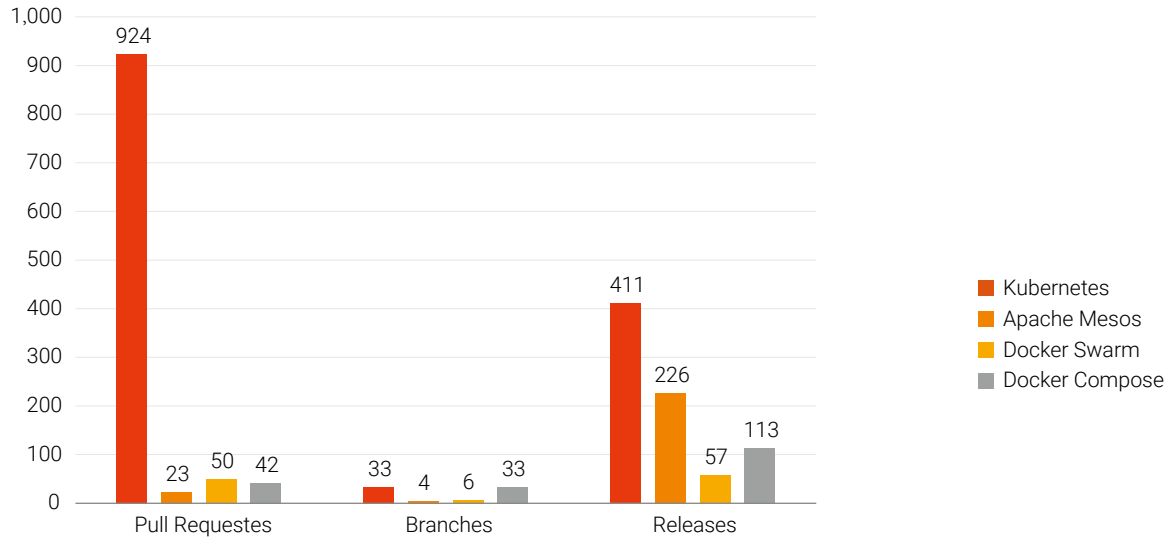
2.4.1.5 Comparative Analysis of Tools

This section provides a comparative analysis of Kubernetes, Apache Mesos, Docker Swarm, and Docker Compose projects on GitHub in terms of their activity. The analysis result, to some extent, reflects the open-source community's interest in and support for the four orchestration tools.

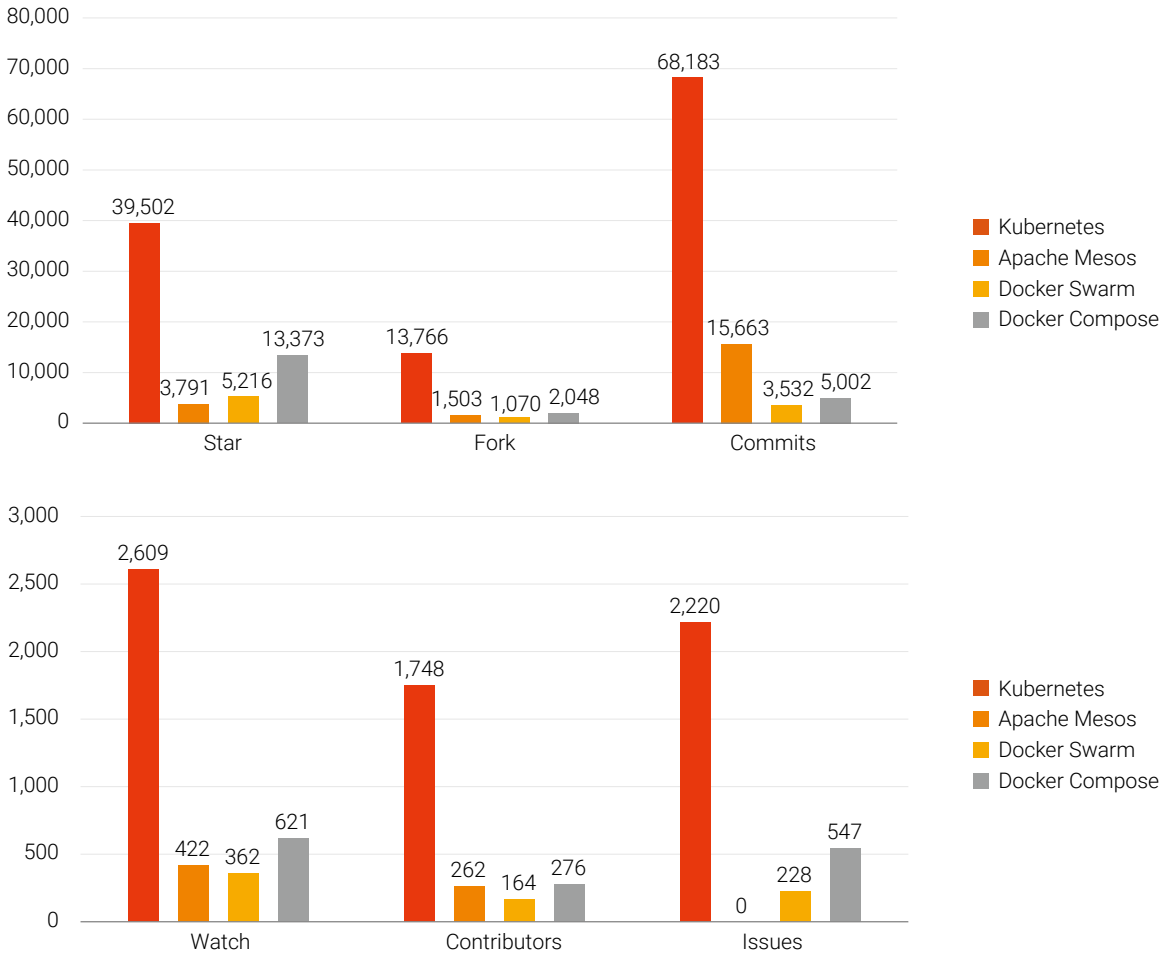
From such indicators as pull requests, issues, commits, branches, releases, contributors, watch, star, and fork, developers are obviously most interested in Kubernetes, followed by Apache Mesos, Docker Swarm, and Docker Compose.

In terms of releases, Kubernetes and Apache Mesos both have a large number of iterations in the aggregate, followed by Docker Compose and Docker Swarm. From a combined view of the interest in projects and the aggregate number of iterations, it can be seen that Kubernetes and Docker Mesos are at the first tier of orchestration tools. By contrast, Docker Swarm and Docker Compose are at the second tier due to less attention paid to them.

Figure 2.17 Comparison of container orchestration tools on GitHub in activity[1]



1 This is based on statistics as of July 31, 2018.



Besides the preceding tools, Rancher[27] and Docker Machine[28] are also common container cluster management tools. Of course, different tools have different functions to address different needs. Thanks to the rapid development of these cluster management tools, the container technology becomes increasingly easy to be accepted and adopted by users at the stages of production and implementation.

2.4.2 Container Usage Scenarios

Like virtual machines, containers provide the isolation function, but with much less resource usage. In the past few years, it has seen more and more widespread application in a variety of scenarios.

According to *A Survey on the Development of Open-Source Cloud Computing Technologies in China (2018)* [29] issued by the China Academy of Information and Communications Technology (CAICT), in 2017, 30.1% of enterprises adopted the container technology in their production environments, an increase of 6.8 percentage points compared with 2016, 36.3% of enterprises used containers in their test environments, and 24.5% of enterprises were evaluating the container technology. Insufficient technology penetration (43.7%), lack of success stories (41.4%), and security concerns (33.8%) are three major contributors to the low adoption rate of the container technology.

The survey also collected data about container usage scenarios and found that 57.9% of enterprises listed operation and maintenance (O&M) automation as the most common scenario involving the use of containers. Other usage scenarios include fast delivery at development and test stages (34%), multi-environment consistency management (31.25), continuous integration/continuous deployment (CI/CD, 29.4%), and microservices (22.7%).

This document identifies cloud computing, DevOps, microservices, and O&M automation as common usage scenarios of the container technology, which are briefly described and analyzed in the following sections.

2.4.2.1 Cloud Computing

Platform as a Service (PaaS) and Container as a Service (CaaS) are both service models of cloud computing, aimed at offering software R&D platforms or service infrastructure to users as a service. By virtue of its inherent advantages to DevOps and CI/CD, CaaS has gained momentum for rapid development.

Many cloud service providers, from Google, Microsoft, and Amazon outside of China to Huawei, Alibaba, and Tencent in China, include container services as part of their public clouds. Examples of such public clouds are Google Kubernetes Engine (GKE)[30], Microsoft's Azure Kubernetes Service (AKS)[31] and Azure Container Instances (ACI[32]), and Amazon's Elastic Container Service (ECS[33]) and Elastic Container Service for Kubernetes (EKS[34]).

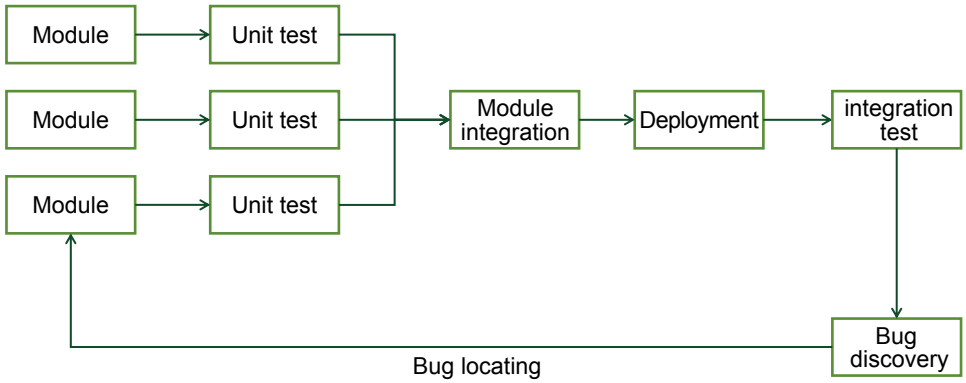
A 2017 RightScale report[35] found that, among all services offered by cloud service providers, Docker stole the limelight with presence in AWS ECS (35%), Azure container services (11%), and GKE (8%).

Enterprise-class PaaS cloud service providers in China include DaoCloud, Shurenyun, TenxCloud, and Alauda, which cooperate with governments and finance and telecom sectors, helping organizations build state-of-the-art cloud-native application platforms. Among these service providers, an overwhelming majority has chosen Docker as their container runtime engine.

2.4.2.2 DevOps

Currently, the development process of software systems usually consists of coding, unit testing, integration testing, and bug handling, as shown in Figure 2.18. Systems are becoming increasingly complicated, and so are dependencies between modules. In this context, many bugs fail to be discovered until project integration. Worse still, the longer away from the development stage, the higher cost you will pay for bug fixing.

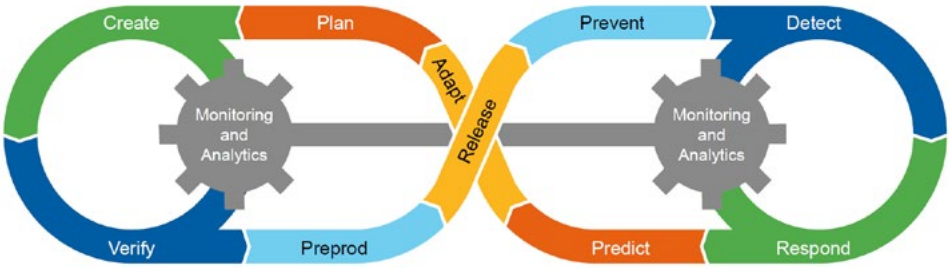
Figure 2.18 Traditional software system development process



DevOps is an IT O&M-oriented end-to-end workflow based on IT automation and CI/CD to optimize development, testing, and O&M, among others.

DevOps is a new model that combines development with operations, with emphasis placed on communication and cooperation between development and O&M personnel. It features an automated process that makes software building, testing, and release more convenient, frequent, and reliable. In addition, DevOps shortens the wait time at each stage and reduces redundant and manual labor, thus significantly cutting down the problem resolving cost and becoming an effective solution to the aforementioned issues.

Figure 2.19 Closed loop of DevSecOps capabilities



One of the key metrics for measuring the success of DevOps is its role in improving the influence of development on operations. The container technology can play a critical role throughout the DevOps process. In practice, the development team should not think their work is complete by just delivering code to the operations team, but should also learn how code is executed in the production environment. For this purpose, they can leverage the container technology to release applications in the form of container images, which cover not only application code but also the entire stack of the basic operating system and dependent libraries. Furthermore, they may even configure a runtime environment. Such consistent deployment of applications reduces the operating expense (OPEX).

According to a 2017 RightScale report[35], over 84% of enterprises had an 80% adoption rate of DevOps; of all

DevOps tools, Docker had an adoption rate of 35% and Kubernetes's adoption rate also increased from 7% in 2016 to 14% in 2017.

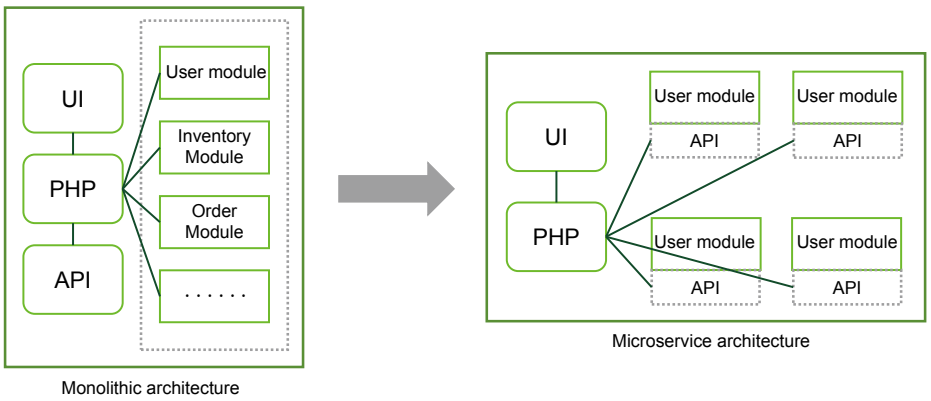
2.4.2.3 Microservices

In a microservice architecture, an application is structured as a collection of loosely coupled modules that provide different services, each of which can be independently deployed, maintained, and expanded. Services communicate with one another through RESTful APIs.

In essence, microservices are designed to resolve more serious and practical problems with services of specific functionality and abstract capabilities. The biggest advantage of the microservice architecture lies in the lowered degree of coupling between modules. This means that each service can be developed and maintained by a separate team. Different services may even use different programming languages and architectures. As a result, each service can be implemented with the most suitable technique. All these contribute to an enhanced application development efficiency, a more clear structure, and reduced learning costs.

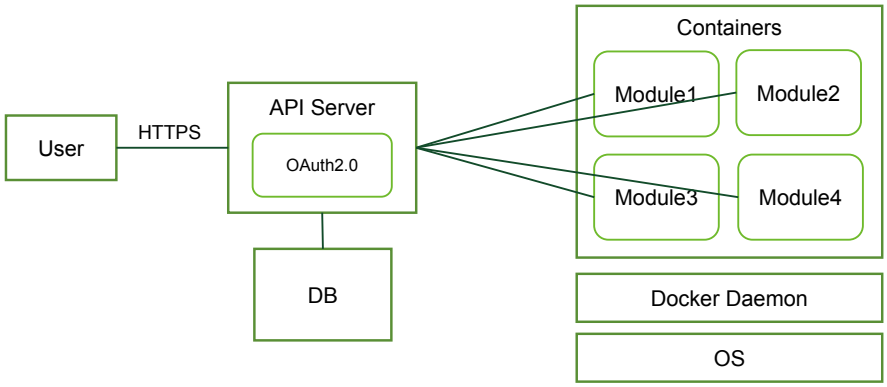
The following figure is an example of conversion from the monolithic architecture to the microservice architecture. The former couples the PHP service with modular functions, while the latter splits the application into different modules, providing an API for each isolatable module, to achieve the purpose of providing services via logical management of APIs.

Figure 2.20 Comparison of the monolithic architecture and microservice architecture



The core architecture of microservices consists of two components: API server and microservice applications. The API server, as the core of the entire microservice architecture, incorporates functions such as authentication, authorization, request logging, request log monitoring, and load balancing. Microservice applications can be packaged into containers to provide services through RESTful APIs within the system. Following is an example of the container-based microservice architecture.

Figure 2.21 Container-based microservice architecture



With the emergence of microservices come a variety of microservice architectures. Spring Cloud[1] and Dubbo[2] are typical examples of the first-generation architecture, which has been widely adopted and used in production environments owing to its stability. The second-generation architecture is a service Mesh[3] , which comes in various forms and typically represented by Linkerd[36] from Buoyant and Istio[37] from the collaborative work of Google, IBM, and Lyft.

2.4.2.4 Fast Deployment and Delivery

The preceding sections dwell upon the advantages of containers in the implementation model and design. In fact, containers are good also in their support for fast deployment and delivery of complicated systems to cut down a lot of work on installation and configuration. Typically, infrastructure platforms such as OpenStack can be deployed using the container technology to deliver automated and agile O&M management.

Kolla[38] is an OpenStack project aimed at containerized deployment of all components. As we all know, it usually takes a lot of time and energy to install and deploy OpenStack. In reality, people always want to put more energy in service logic implementation rather than setup and deployment of the platform environment.

If Kolla is used to deploy OpenStack, after the operating system is installed, it takes only 10 minutes, give or take, to complete installation of an OpenStack system with various community functions. This containerized deployment is similar to block building. It makes all operations, from installation to upgrade, more graceful.

1 Spring Cloud builds on Spring Boot to provide tools for developers to quickly build some of the common patterns in JVM-based cloud application development, including configuration management, service discovery, intelligent routing, and micro-proxy.

2 Dubbo is a high-performance service framework open-sourced by Alibaba. Able to be seamlessly integrated with the Spring framework, Dubbo makes it possible for applications to output/input services through high-performance RPCs.

3 A service mesh is an infrastructure layer for handling inter-service communication. In practice, the service mesh generally consists of a series of lightweight network proxies, which are deployed with applications and are transparent to the latter.

2018 NSFOCUS
Technical Report
on Container Security

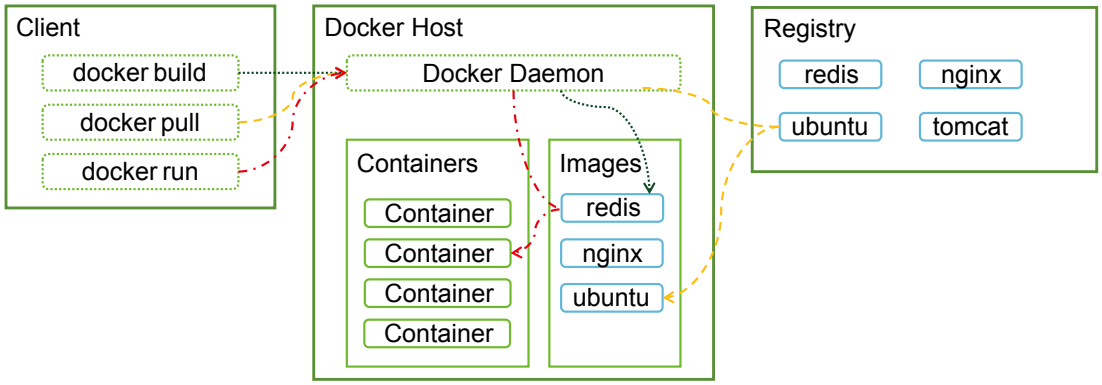
3. Vulnerability and Security Risk Analysis

3.1 Vulnerability and Security Risk Analysis ····· 36
3.2 Security Threat Analysis ····· 45
3.3 Container Application Security Threat ····· 47

3.1 Vulnerability and Security Risk Analysis

As a specific implementation of the container technology, Docker is getting more and more popular in recent years. To some extent, Docker has become a typical representation of the container technology. Docker is based on the common client/server (C/S) architecture design as shown in Figure 3.1. In this architecture, the Docker daemon (server) runs on the Docker host and the Docker client sends requests to the daemon. This chapter uses Docker as an example to describe containers' vulnerabilities and security risks.

Figure 3.1 Docker's C/S architecture



3.1.1 Software Risks

Docker surely contains security risks in its software design and code implementation. This section analyzes security risks in Docker from two aspects: software design and code vulnerabilities.

3.1.1.1 Software Design

Docker, though implementing excellent operating system-level isolation in design, contains security risks in multiple aspects such as its default networking mode, sharing of the operating system kernel with the host, sharing of host resources, use of the Linux capabilities mechanism, and insufficient isolation. The following describes attacks that could result from security risks in Docker.

（1）Local area network (LAN) attack

As for the network implementation, Docker supports multiple networking modes. In default bridge mode, containers created on the same host will all connect to the *docker0* bridge. This means that those containers constitute an LAN. In this case, LAN-targeting attacks like ARP spoofing, sniffing, and broadcast storm will pose security threats to the containers. Therefore, if more than one container is deployed on the same host, you need to perform proper network configuration and set access control rules to isolate networks with borders.

（2）Denial-of-service (DoS) attack

Container instances share resources of the host, including the underlying CPU, memory, and disk resources that are scheduled and allocated by the host's operating system in a centralized way. If resources used by containers

are not properly restricted and managed, a disparity may exist in the use of resources among containers, causing exhaustion of resources of hosts and the cluster in worst-case scenarios and finally leading to a denial of service.

（3）Vulnerable system call

The major difference between Docker and virtual machines (VMs) is that Docker shares the operating system kernel with the host. If the host kernel contains a horizontal unauthorized access vulnerability or privilege escalation vulnerability, attackers compromising containers can still be able to exploit such vulnerability to escape to the host for malicious operations, though containers run in non-privileged mode.

（4）Root privilege sharing in privileged mode

Now that the containers share the same operating system kernel with the host, when containers run in privileged mode, root users of the containers have root access to the host and can nearly perform operations without limit.

（5）Non-isolated file system

Docker has isolated the file system, but fails to isolate important system files such as */sys, /proc/sys*, and */proc/bus*.

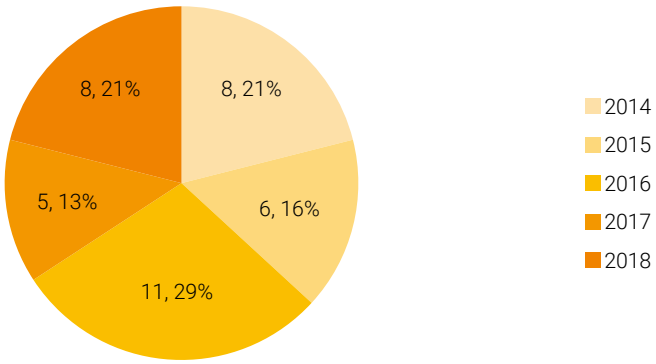
Docker has protection solutions in place to deal with the preceding attacks. Therefore, such attacks can be prevented as long as proper protection configuration is performed. For details, see the related official documentation[39].

3.1.1.2 Software Vulnerabilities

According to statistics of China National Vulnerability Database of Information Security (CNNVD[40]), by July 31, 2018, a total of 38 vulnerabilities had been discovered in Docker which is integrated in open-source projects or products from vendors such as Cisco, Boot2Docker, and Jenkins.

Figure 3.2 shows the number and percentage of Docker vulnerabilities discovered in recent years. We can see that each year witnessed a certain quantity of Docker vulnerabilities announced and there is no sign of decrease in vulnerabilities, indicating that this software still contains security risks after years of development.

Figure 3.2 Percentage of Docker vulnerabilities in recent years[1]



1 Data as of July 31, 2018

above and 36% high-risk and critical. Four critical vulnerabilities have a CVSS score of 10, including two respectively assigned CVE-2014-9357 and CVE-2016-9223. The CVE-2014-9357 vulnerability exists in Docker 1.3.2 and allows a remote attacker to execute arbitrary code with root privileges via a crafted image or "build" in a Dockerfile in an LZMA (.xz) archive. The CVE-2016-9223 vulnerability exists in the Docker Engine configuration of Cisco CloudCenter Orchestrator due to a misconfiguration. A remote attacker could exploit it to deploy Docker containers with high privileges on the affected system. Figure 3.3 shows the distribution of all Docker vulnerabilities in terms of risk level.

Figure 3.3 Distribution of Docker vulnerabilities [1] by risk level

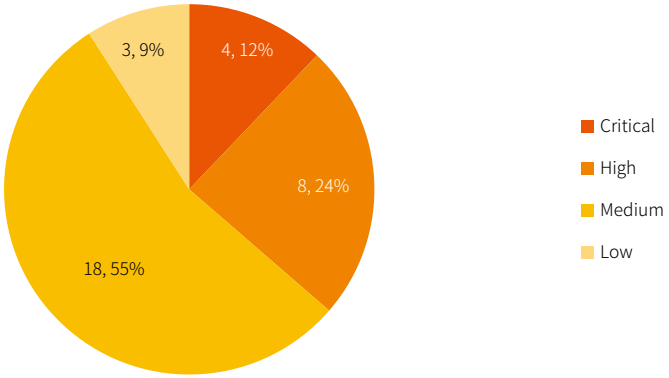
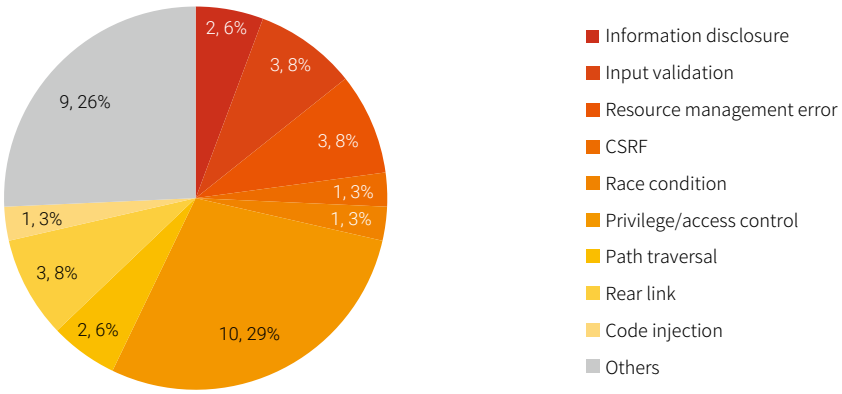


Figure 3.4 shows the distribution of security risks that could be incurred by all vulnerabilities, including privilege and access control, rear links, resource management errors, and input validation. For example, the vulnerability (CVE-2016-8867) in Docker Engine 1.12.2 could pose privilege and access control threats. This vulnerability exists because Docker Engine enables ambient capabilities with misconfigured capability policies. An attacker, via a malicious image, could bypass user permissions to access files within the container file system or mounted volumes.

Figure 3.4 Impacts of Docker vulnerabilities[2]



1 Data as of July 31, 2018
2 Data as of July 31, 2018

3.1.2 API Security

According to the architecture, Docker listens on a UNIX socket by default, for example, *unix:///var/run/docker.sock*. For cluster management, Docker also provides a REST API for remote management, allowing remote access through TCP.

For example, when Docker Swarm is used, the Docker node will open TCP port 2375 which is bound to the IP address 0.0.0.0. Running as a daemon in the background, Docker, by default, exposes the non-encrypted listening port 2375. Besides, a user, when executing the startup parameter, *dockerd -H=0.0.0.0:2375 -H unix:///var/run/docker.sock*, can enable Docker to listen on port 2375 on all local addresses. In this way, the Docker daemon can execute Docker command requests it receives, for example, executing the *docker –H tcp://$HOST:2375 ps* command to retrieve all Docker instances on the $HOST host. Now that the user can execute the *docker ps* command, he or she can also run commands such as *docker run* and *docker rm*.

However, enabling the Docker Remote API without any encryption and access control is very risky. In particular, once spotting the default port 2375 exposed on the Internet, an attacker can gain access to container data without authentication, resulting in sensitive information disclosure. Also, the attacker can delete data on containers or directly access sensitive information on the host by taking advantage of characteristics of containers. Doing so, the attacker may gain the server's root privileges to manipulate sensitive files and finally take full control of the server.

In May to July, 2018, we used NTI[41] to search for all port 2375 on the network and found that 337 IP addresses bound to this port were exposed on the Internet. Figure 3.5 shows the global distribution of hosts with port 2375 exposed. This reflects the popularity of Docker, but also reveals that this program is not used in a proper way and relevant operations need to be regulated to prevent security risks.

Figure 3.5 Global distribution of hosts with port 2375 exposed

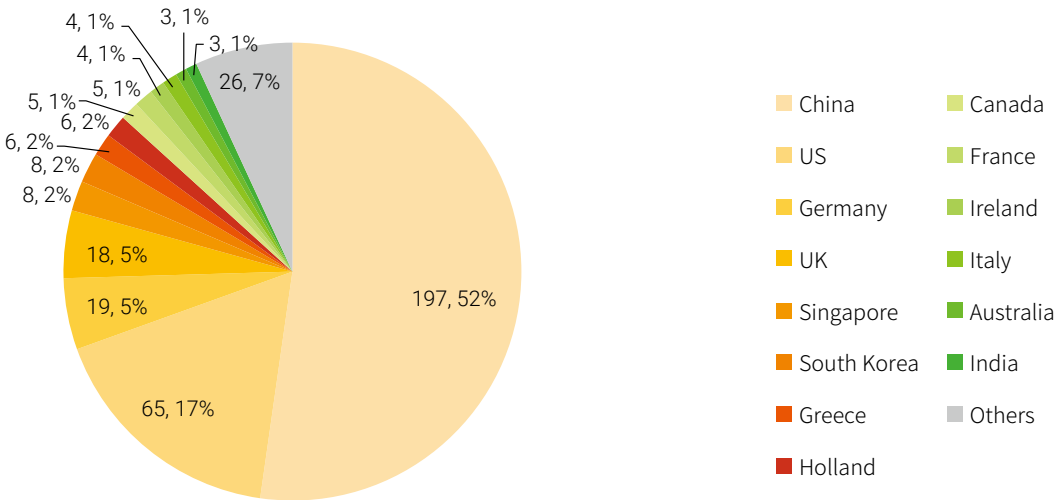


Figure 3.5 shows geographical statistics of 337 exposed IP addresses. Globally, most of the exposed Docker services were found in three countries: China ranked first with 197 (52%) exposed IP addresses, followed by USA

(65 or 17%) and Germany (19 or 7%).

Figure 3.6 shows the distribution of Docker hosts exposed in China. Zhejing took the largest proportion (29%) with 57 exposed IP addresses and Beijing and Guangdong came in second and third respectively with 43 (22%) and 35 (18%) exposed addresses.

Figure 3.6 Distribution of port 2375 exposed in China

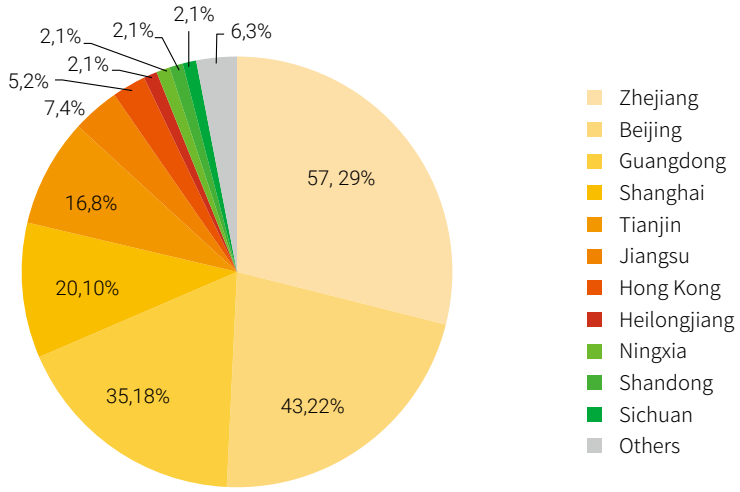
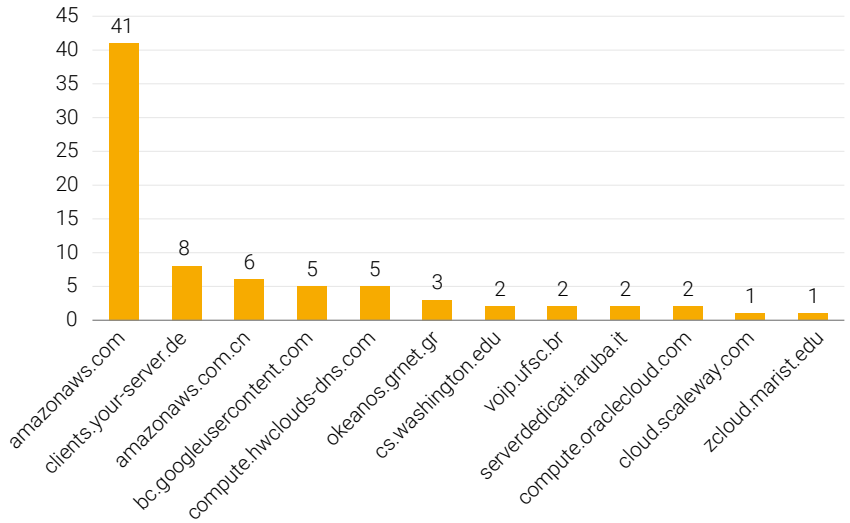


Figure 3.7 shows domain name statistics of Docker's 337 IP addresses, including IP addresses of certain known public cloud vendors.

Figure 3.7 Distribution of domain names of exposed Docker IP addresses



It should be noted that port 2375 exposure on the Internet is a kind of misconfiguration rather than a Docker vulnerability. For this point, an official Docker document gives a warning that binding Docker to a TCP port

accessible from the Internet allows a malicious user to obtain root privileges of the Docker host. The warning is as follows:

> Warning: Changing the default docker daemon binding to a TCP port or Unix docker user group will increase your security risks by allowing non-root users to gain root access on the host. Make sure you control access to docker. If you are binding to a TCP port, anyone with access to that port has full Docker access; so it is not advisable on an open network.

Apart from port 2375 for remote access to the Docker daemon, we also analyzed Kubernetes exposure in July 2018. Scanning the entire Internet for port 6443 (default SSL port of the Kubernetes API server) by using NTI, we found that a total of 12,803 Kubernetes services were exposed on the Internet. Figure 3.8 shows the global exposure of Kubernetes services.。

Figure 3.8 Global exposure of the Kubernetes service

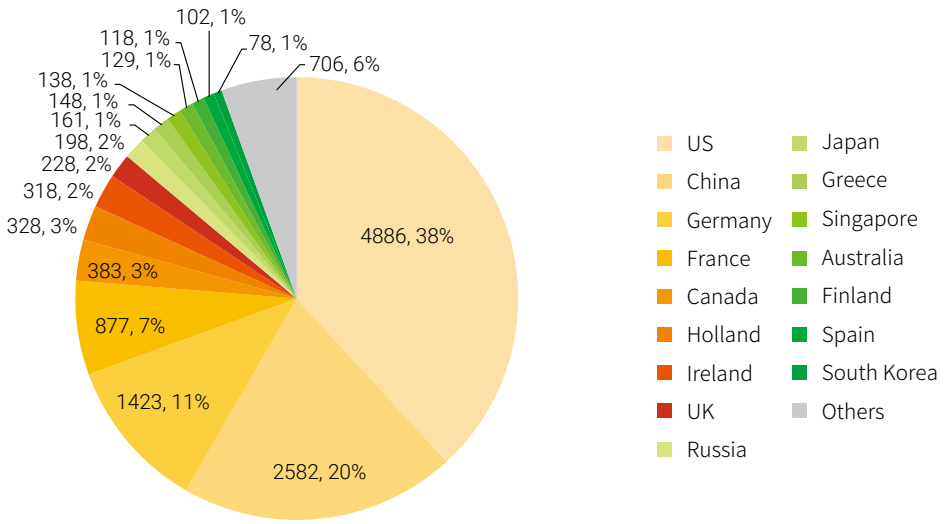
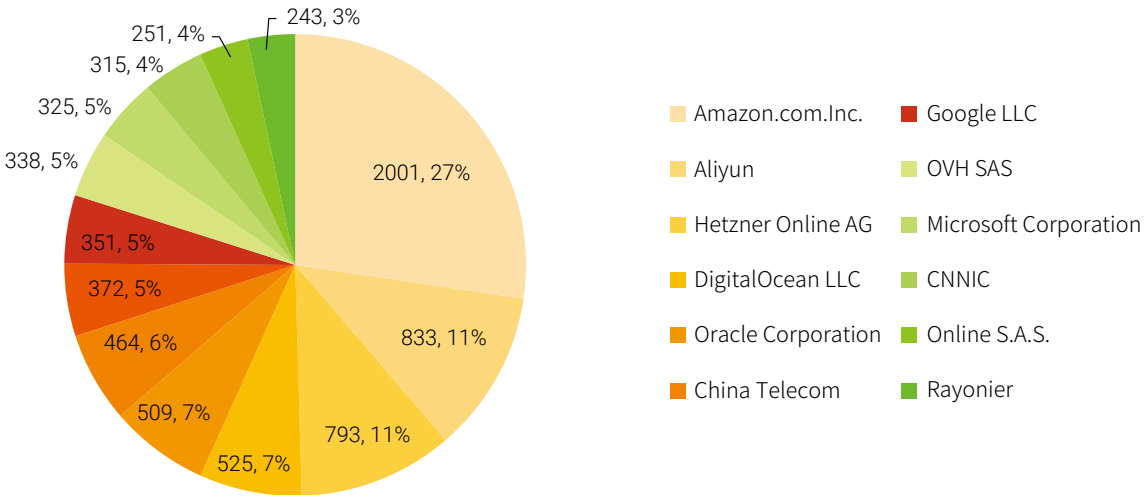


Figure 3.9 Global distribution of managed service providers with Kubernetes exposed



Like port 2375 for remote access to the Docker daemon, the Kubernetes service was mainly exposed in the USA, China, and Germany. The USA saw the most Kubernetes services (4886 or 38%) exposed and China (2582 or 20%) and Germany (1423 or 11%) respectively came in second and third. Most of those exposed Kubernetes services are deployed on public clouds such as Amazon Web Services (AWS) and Aliyun.

Figure 3.10 Distribution of exposed Kubernetes services in China

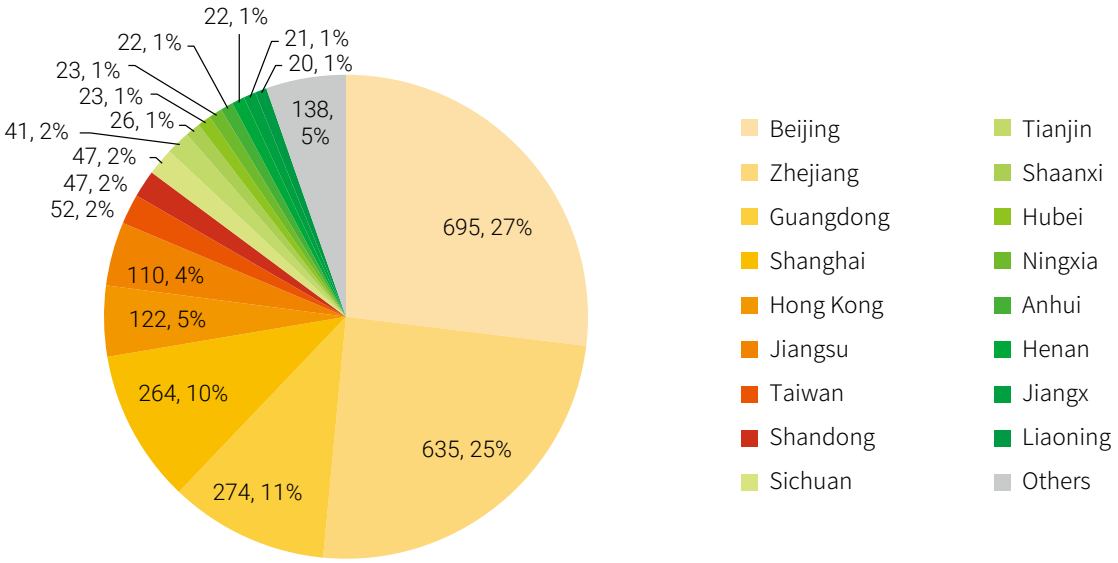
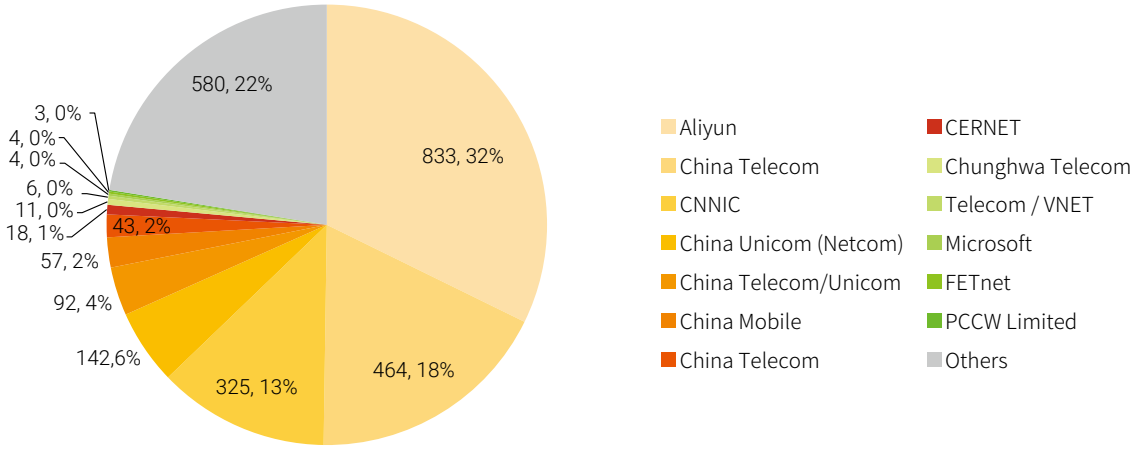


Figure 3.11 Distribution of managed service providers that have Kubernetes exposed in China



As shown in Figure 3.10, Kubernetes hosts in China were mostly exposed in Beijing, Zhejiang, and Guangdong. Beijing topped the ranking list with 695 (27%) exposed hosts, followed by Zhejiang (635 or 25%) and Guangdong (274 or 11%).

From the above, we can see that container orchestration systems have developed rapidly at home and abroad, and due to container deployment characteristics, the exposure distribution of containers deployed on public

clouds, data centers, and orchestration system services is in line with the development trend of container services and cloud services. As containers and container orchestration systems are deployed in large quantity, numerous services are exposed on the Internet, broadening the attack surface. To make matters worse, a large portion of such services lack basic security configurations and hardening measures and can therefore easily be exploited by attackers to compromise the availability and integrity of systems or by third parties to launch a variety of attacks.

3.1.3 Insecure Images

Developers often download images from the official Docker Hub repository, of which some are from official organizations developing certain software in images and a lot are contributed by third-party organizations and even individuals. Throughout the application lifecycle, developers, testing personnel, and O&M personnel will download and run images as required. Therefore, it is really important to check images before the container runs.

Apart from Docker Hub, there are a great number of third-party image repositories such as China's NetEase Cloud[42] (registration is required), USTC Mirror[43] (public and open), DaoCloud[44] (registration is required), and Aliyun[45]. These third-party image repositories make it convenient to obtain images, but also bring potential security risks. For example, downloading software included in images is insecure because possibilities exist that the downloaded images contain malicious backdoors or are tampered with in transit.

（1）Some images contain vulnerable software.

A relevant research report[46] shows that more than 30% of official images in Docker Hub contain high-risk vulnerabilities and almost 70% of the total have high or medium-risk vulnerabilities. We selected 10 images with highest downloads and evaluations from Docker Hub and scanned their latest versions with Clair[47]. The following table presents our analysis results.

Table 3.1 Vulnerability statistics of some images in Docker Hub

Image	STARS	PULLS	HIGH	MEDIUM	LOW
nginx	8.1K	10M+	3	14	8
ubuntu	7.3K	10M+	0	6	14
mysql	5.8K	10M+	4	7	5
node	5.2K	10M+	68	193	71
redis	4.9K	10M+	6	11	9
postgres	4.7K	10M+	15	32	12
mongo	4.2K	10M+	6	11	9
centos	4.1K	10M+	0	0	0
jenkins	3.4K	10M+	11	25	21
alpine	3.3K	10M+	0	0	0

As shown in the preceding table, a great majority of images contain high-risk vulnerabilities and some even have dozens. Vulnerable images mainly belong to application software, while mainstream operating systems are

relatively secure because of proper vulnerability management. For example, no CVE vulnerabilities are found in CentOS and Alpine.

(2) Some malicious images are uploaded by attackers.

If hackers install malware such as trojans and backdoors in their created images before uploading such images to public repositories like Docker Hub, users may have an insecure container environment from the day they use such images, let alone days afterward.

For example, Docker does not handle images in a secure way as it supports three compression algorithms, Gzip, Bzip2, and XZ. Gzip and Bzip2 are relatively secure, owing to the use of the Go standard library. XZ is built upon the open-source XZ Utils written in C, instead of the native Go, and therefore has a possibility of being maliciously written by hackers. Once the malicious writing occurs, XZ will be potentially vulnerable to arbitrary code execution. In fact, as long as there is a vulnerability in XZ, a user is likely to crash the system when executing the *docker pull* command to pull images.

For specific files, you can typically use antivirus (AV) software to check whether they are secure. However, AV software currently lacks sufficient support for image scanning. If users would like to check the security of images to be downloaded, all they can do is determine if backdoors exist (for example, run images and install AV software to scan them) and make sure that their requests are directed to official images.

(3) Some images are tampered with via man-in-the-middle attacks.

If users download images from the image repository in a non-encrypted way, these images in transit may be tampered with through man-in-the-middle attacks. To address this issue, Docker 1.8 provides the content trust mechanism.

3.1.4 Container Isolation Losing Effect

As containers share the operating system kernel with the host, security risks may be incurred once container isolation loses effect.

1. Once the attacker successfully compromises the operating system kernel used by containers, he or she is able to access the file system on the host or break into other containers, rendering container isolation void.
2. The host's file system can be mounted to the directory of more than one container so that they can access this file system. An attacker, once gaining access to one container and learning the file system directory, can get into file systems of other containers so as to finally cause information disclosure, content defacement, or other security issues.

3.2 Security Threat Analysis

When we talk about security risks to containers, we mean security threats to hosts, to containers, and to the carried applications.

3.2.1 Container Escape Attack

Similar to virtual machine (VM) escape attacks, in a container escape attack, attackers exploit a vulnerability existing in virtualized software to gain access to a host through a container, in a bid to launch an attack against the host.

Specifically, some PoC tools, such as Shocker[48], can demonstrate how to escape from a Docker container to a host and read the file content of a directory on the host. The key of Shocker attacks lies in the call of the *open_by_handle_at* function. As specially mentioned in the Linux manual, the caller must have the *CAP_DAC_READ_SEARCH* capability to invoke *open_by_handle_at*. However, Docker 1.0 adopts the blacklist mechanism to manage capabilities and sets no limit for the *CAP_DAC_READ_SEARCH* capability, leading to container escape risks.

3.2.2 Container Network Attack

Currently, Docker provides three types of network drivers, namely, bridge network driver, MacVLAN driver, and overlay network driver. All of them are prone to security risks.

For the bridge network driver built in Docker, a bridge called *docker0*, which acts as a router and NAT, connects to all containers. This means that all traffic between containers should first go through the container host. If there is no firewall between containers, attackers could exploit private network internal to the host to have containers attack one another.

The MacVLAN driver is a lightweight network virtualization technique. The MacVLAN driver connects container interfaces directly to host interfaces. Compared with the bridge network driver, the MacVLAN driver ensures better isolation from the physical network. However, this type of network driver does not exercise permission control between containers on the same virtual network, so that attackers could easily gain access to containers and launch network attacks.

To address this issue, Docker provides fine-grained container permission control by introducing some extension plug-ins. For example, a network plug-in provides the network connection model between containers, a volume plug-in provides multi-host volume management for Docker, and a privilege-based plug-in provides access control.

The overlay network driver uses the VXLAN technique to form a new virtual network on basis of the underlay network between different hosts. Such network architecture is often used in distributed orchestration frameworks. Though the overlay network driver can build a distributed container cluster rapidly, it has its own disadvantages. The biggest disadvantage is that traffic on the VXLAN network is not encrypted, which allows attackers to steal or tamper with the transmitted content.

However, Docker users can choose to encrypt data when setting IPSec tunnel parameters to guarantee the

security of the container network. In addition, like the bridge network driver and MacVLAN driver, connections between containers in the overlay network mode are not effectively controlled, which may cause ARP spoofing, sniffing, and broadcast storm attacks. Such risks are prevalent in container networks at present.

3.2.3 Denial-of-Service (DoS) Attack

Since containers are technically implemented based on the host kernel and share host resources, they are more prone to container-targeting DoS attacks. For example, by default, each container is allowed to use all memory on the host. If a container accesses the host exclusively or consumes plenty of host resources, other containers on the host are likely to work improperly due to lack of resources.

DoS attacks can be launched against any resources, such as computing resources, storage resources, and network resources.

（1）Computing resources

The fork bomb[1], in the computing field, is a form of DoS attack against computing resources. Normally, the host kernel only supports a specified number of processes. If too many processes are created in a container and they consume all process resources on the host, other containers will have no resources for creating new processes or the host may fail to work properly.

The fork bomb has always been a topic discussed in the Docker community ever since 2015. So far, the best way to defend against it is to restrict memory use (*--kernel-memory=#M*). However, errors may occur when this method is used for encrypted files.

（2）Storage resources

In the implementation of the container technology, file systems are isolated by applying *mount* namespaces. However, file system isolation is only a very basic requirement. It is recommended that AUFS not be used as the storage driver. This is because though it can create isolated container file systems, it has no limitations on storage space. In other words, if a container keeps writing files, it will finally fill up the storage media, leaving no space for other containers to perform write operations, and thereby causing a denial of service.

（3）Network resources

DoS attacks emerge in an endless stream, and network bandwidth exhaustion in a container is one type of such attacks. Specifically, an attacker directs a large number of controlled hosts to send loads of network packets to the attack target (a container), in a bid to take up its network bandwidth and consume the container host's capability of processing network packets, finally causing a denial of service.

1 The fork bomb is a DoS attack that uses the fork system call (or other equivalent methods). Unlike viruses and worms, the fork bomb is not infectious. It disables systems that have restrictions on the number of concurrent processes and programs from executing new programs and makes systems that do not have such restrictions stop responding.

3.3 Container Application Security Threat

3.3.1 Microservice Security

From traditional monolithic applications to modern microservice applications, security has always been a hot issue. A monolithic application usually exposes fewer services and ports, narrowing the attack surface. In addition, security professionals know common points from which attacks are often launched. Therefore, security is not that big of a problem for such applications as long as they are properly protected.

By contrast, the microservice model splits traditional monolithic application modules into separate services, resulting in a greater number of exposed ports and a broader attack surface. As for the traditional monolithic architecture, it is enough to protect only one entrance for access permissions, authorization, and isolation audits.

The microservice architecture contains a number of services, access to each of which should be properly monitored, controlled, and protected. Just imagine that an authentication token is leaked or a forged access credential is accepted for a service. This will put the entire system in jeopardy. Such threats undoubtedly add to the difficulty of microservice protection.

A microservice architecture consists of a collection of small, autonomous services. Although this model lays particular emphasis on isolation, lightweight, independent development and deployment, and loose coupling of services, sometimes services need to be closely connected for a particular purpose, for example, sharing data. Connections between these services in the microservice architecture are usually point-to-point. With the increase of connections, once a service is compromised due to a vulnerability, other services connecting to it will also be affected, finally leading to the attacker taking control of the entire system.

Moreover, microservices usually use containers as carriers. This means that applications are packaged into images and run in the microservice architecture as containers in a distributed manner. Containers themselves are vulnerable to threats, such as container escape and container network attacks.

3.3.2 DevOps Security

In the big data era that sees rapid growth of networks, many enterprises maintain an "agile" mindset and take "agile" actions. For example, DevOps, as a new development and operations model, shortens the wait time at various stages of the software lifecycle and reduces redundant and manual labor, thus significantly cutting down the problem resolving cost and enhancing the efficiency of agile development. But the preference of agility to security can be a dangerous signal, which has been repeatedly verified in practice[49].

> On November 22, 2017, Uber released a statement, acknowledging that the company suffered a breach in 2016 that exposed massive personal information. According to this statement, two hackers attacked Uber through a third-party cloud service, stealing data of 57 million users, including names and license numbers of drivers and names, email addresses, and mobile numbers of customers. The subsequent investigation found that the data breach had been caused by Uber engineers' storage of security keys for unlocking the database on a GitHub page that was publicly accessible.

This is not the only case where data is disclosed because of misoperations. What deserves particular attention is that the rapid development of cloud environments and DevOps nowadays has brought forth much more security risks. Frans Rosén, a security advisor from the security company Detectify, said in a report released on July 13, 2017 that network administrators too often glossed over rules for configuring AWS's access control lists (ACLs) and the misconfiguration of servers had caused a large number of data breaches.

More and more consumers, supervisory bodies, and markets have found that the cost of data breaches arising therefrom can be unacceptably exorbitant. Because of the leak of data, hundreds of millions of dollars may be lost in market capital overnight and consumers' confidence in organizations degraded. In certain circumstances, an organization's executives may even meet their Waterloo in their careers. Some enterprises relying heavily on data may literally close down due to an unintentional negligence in the storage of keys.

4. Container Security Protection

4.1 Linux Kernel Security Mechanism 50
4.2 Container Service Security 53
4.3 Host Security 54
4.4 Image Security 56
4.5 Container Network Security 59
4.6 Runtime Security 71
4.7 Orchestration Security 81
4.8 Application Security 84

As a lightweight virtualized implementation, the container technology took into account security factors at the time of design, which constitute an important basis for container security protection. This chapter describes security risks and threats facing containers and common protection ideas and methods.

4.1 Linux Kernel Security Mechanism

Basic security mainly refers to the implementation of functional modules related to the Linux kernel, including the isolation and management of container resources. At the time of design, the container technology (both Docker and LXC) has similar security considerations. For example, the kernel namespace is used to build a relatively isolated running environment, guaranteeing that each container runs independently. cgroups is used to isolate, restrict, and audit shared resources (such as CPU, memory, and disk I/O), so as to avoid the competition of containers for system resource. These Linux kernel modular functions provide the basic security assurance for containers.

4.1.1 Kernel Namespace

Namespaces are a powerful feature of the Linux kernel. When you start a container, behind the scenes Docker creates a set of namespaces and control groups for the container. Namespaces provide the first and most straightforward form of isolation for containers: Processes running within a container cannot see, and even less affect, processes running in another container, or in the host system. There are process ID (PID), network, interprocess communication (IPC), mount (mnt), UTS, and user ID namespaces, among others.

For instance, PID namespaces isolate the processes of different users, ensuring that processes in different PID namespaces can have the same PID and facilitating container nesting. For process interaction in a container, Docker adopts the Inter-Process Communication (IPC) mechanisms, including semaphores, message queues, and shared memory. Namespace information should be added at the time of applying for IPC resources, ensuring that each IPC resource has a unique 32-bit ID.

For container network isolation, Docker uses the network namespace mechanism. Each network namespace has its own network device, IP address, and routing table. Each container also gets its own network stack, meaning that a container does not get privileged access to the sockets or interfaces of another container.

Kernel namespaces were introduced in the Linux kernel version 2.6.26. This means that since July 2008 (date of the 2.6.26 release), namespace code has been exercised and scrutinized on a large number of production systems. So both the design and the implementation are pretty mature.

4.1.2 Control Groups

Control Groups (cgroups) are another key component of Linux Containers. They implement resource accounting and limiting. The code of control groups was first started by Google in 2006, and initially merged in kernel 2.6.24.

cgroups are important in the implementation of container technology. They provide many useful metrics, but they also help ensure that each container gets its fair share of memory, CPU, and disk I/O; and, more importantly, that

a single container cannot bring the system down by exhausting one of those resources.

So while they do not play a role in preventing one container from accessing or affecting the data and processes of another container, they are essential to fend off some denial-of-service attacks. They are particularly important on multi-tenant platforms, like public and private PaaS, to guarantee a consistent uptime (and performance) even when some applications start to misbehave.

4.1.3 Linux Kernel Capabilities

Your average server needs to run a bunch of processes as root, including the SSH, cron, syslog daemon, hardware management tools, network configuration tools, and more. Once the service is compromised, attackers will have the highest privileges. Linux kernel capabilities provide finer-grained access control over file system mounting and access and kernel module loading.

In most cases, container services can exploit Linux kernel capabilities to avoid using real host root privileges. Therefore, containers can run with a reduced capability set. For instance, it is possible to deny all "mount" operations, deny access to raw sockets, and deny access to some file system operations like creating new device nodes and changing the owner/attributes of files. This means that even if an intruder manages to compromise a container, it will be much harder to do serious damage, to the host from within the container.

By default, Docker adopts a whitelist approach to drop all capabilities except those needed. You can see a full list of available capabilities[50] in Linux manpages.

One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may cause incomplete isolation between containers. Furthermore, attackers could exploit host system vulnerabilities to launch an attack against containers. Docker supports the addition and removal of capabilities, allowing use of a non-default profile. This may make Docker more secure. The best practice for users would be to remove all capabilities except those explicitly required for their processes.

4.1.4 Other Kernel Security Features

Capabilities are just one of the many security features provided by modern Linux kernels. It is also possible to leverage existing, well-known systems like TOMOYO[51], AppArmor[52], SELinux[53], and GRSEC[54], with Docker to harden the security of Docker containers.

You can run a kernel with GRSEC and PAX. This adds many safety checks, both at compile-time and run-time; it also defeats many exploits, thanks to techniques like address randomization. It does not require Docker-specific configuration, since those security features apply system-wide, independent of containers.

If your distribution comes with security model templates for Docker containers, you can use them out of the box. For instance, Docker has officially released a template that works with AppArmor and Red Hat comes with SELinux policies for Docker. These templates provide an extra safety net. You can define your own policies using your favorite access control mechanism.

4.1.4.1 Security-Enhanced Linux

Mandatory access control (MAC) is a security strategy that restricts the ability individual resource owners have to grant or deny access to resource objects in a file system. That is to say, the system forcibly performs access control, regardless of user behaviors.

Introduced by the National Security Agency (NSA), Security-Enhanced Linux (SELinux) is an implementation of MAC and is the most outstanding security subsystem on Linux. Under the restrictions of SELinux, processes can access only files needed for a specific task. Among currently available Linux security modules, SELinux is the most powerful and thoroughly tested. It is built on the basis of 20 years of research on MAC.

Method: Add the SELinux option when launching the Docker service.

```
# docker daemon --selinux-enabled = true
```

When shocker[1] is running,

```
# docker run --rm -ti --cap-add=all shocker bash
```

The loop traversal of Shocker code in the process of brute-force guessing locates */etc/shadow*, but is denied to read the information by SELinux. The returned information is as follows:

```
#open: Permission denied
```

This means that SELinux is really effective.

4.1.4.2 AppArmor

AppArmor is also a MAC system to confine programs to a limited set of resources. It can deny programs to read/write a certain directory and open/read/write network ports.

During running, the Docker service determines whether the current kernel supports AppArmor. If yes, it creates the default AppArmor configuration file */etc/apparmor.d/docker* and then applies this file. When a container is launched, Docker applies the corresponding AppArmor configuration during initiation. The *–security-opt* option can also be set to specify the AppArmor configuration file for the container.

To create an AppArmor rule and apply it to the container, you need to first copy a template.

```
# cp /etc/apparmor.d/docker /etc/apparmor.d/container
```

Method: Modify the */etc/apparmor.d/container* configuration file by adding a line of *deny /etc/hosts rwklx*, and then apply this rule to run Shocker.

```
# apparmor_parser -r /etc/apparmor.d/container
# docker run --rm -ti --cap-add=all --security-opt apparmor:container-default shocker bash
```

The returned result is also "*open: Permission denied*", suggesting that the attack fails. The "*deny /etc/hosts rwklx*" rule denies the request of other programs in the container for obtaining */etc/hosts*.

1 Here, it refers to Shocker of Github, which describes how to escape from Docker containers and read the content in /etc/host on the host. For complete code, visit the following link: http://stealth.openwall.net/xSports/shocker.c

Compared with the virtualization technology, the container technology does not have Hypervisor. That is to say, the container's resource management does not have Virtual Machine Manager (VMM) and therefore entirely relies on the host's operating system. Thus, how to make sure that containers are properly used according to their specification becomes an important prerequisite for container security.

Together with the Center for Internet Security (CIS), Docker released a detailed benchmark configuration file[55], which provides configuration suggestions from the perspectives of container host, Docker dameon, image configuration, and running configuration.

4.2 Container Service Security

The security of the container management and orchestration service has a direct bearing on that of the container control plane. Take Docker for example. Whether the Docker daemon is properly configured determines the security of Docker to some extent. It is recommended that the following settings be configured when starting the Docker daemon:

（1）Restrict inter-container network traffic.

By default, all traffic between containers on the same host is permitted. This is the so-called blacklist mechanism, which allows users to add access control lists as required. To restrict communication between containers on the same host, the whitelist mechanism can be employed (by passing a command-line parameter *--icc = false* to the Docker daemon) to prohibit inter-container traffic by default. Containers can communicate with one another only after they are added to the whitelist. In addition, we can add containers to a custom network to restrict their communication with containers on other networks.

（2）Set remote, centralized log management.

Containers have a short lifecycle that is full of rapid service changes. To address this issue, we can put logs on a remote, centralized platform for security analysis and forensics by using *docker --log-driver=syslog --log-opt syslog-address=tcp://192.xxx.xxx.xxx.*

（3）Allow Docker to modify iptables rules.

The Docker daemon can automatically modify iptables as required according to users' network configuration. It is recommended that the Docker daemon be allowed to automatically update the iptables configuration by setting *docker --iptables to true*.

（4）Do not use AUFS as the storage driver.

The advanced multi-layered unification filesystem (AUFS), as a rather old storage driver of Linux, may trigger some major issues such as kernel crash. Many new Linux kernel releases stop providing support for AUFS. Users can use the *docker --storage-driver devicemapper* command to specify the device mapper as the storage driver. By default, Docker on most platforms adopts the device mapper as the storage driver. This may vary with operating systems. The preferable choice goes for the best storage driver supported by the operating system.

（5）Use default cgroups.

Docker controls container resources with cgroups, which can be set with the *docker --cgroup-parent = /foobar* command line.

For configuration of the daemon, besides the preceding command line, another common operation is to modify the *docker.service* file, whose path is generally */lib/systemd/system/docker.service*. The path may vary with Linux distributions. After the file is modified, the Docker daemon needs to be restarted to control resources with cgroups.

（6）Set ulimit at the Docker daemon level.

ulimit is a built-in function of Linux. It contains a set of parameters for controlling resources available for the shell or processes started by it. Resource types supported by ulimit include the size of kernel files created, size of process data chunks, size of files created by shell processes, size of locked memory, size of the resident memory set, number of opened file descriptors, maximum value of the allocated stack, CPU time, maximum number of stacks for a single user, and maximum virtual memory available for shell processes. ulimit has two types of settings: hard and soft.

Setting ulimits can be useful for avoiding resource exhaustion-caused problems, such as a fork bomb. Sometimes, legitimate users and process can also overuse system resources, leading to resource exhaustion. Therefore, it is necessary to control resource usage by setting default ulimits for the Docker service.

（7）Use trusted repositories.

By default, the Docker process allows *push* and *pull* of images only from repositories configured with trusted certificates. It is not advisable to enable this configuration item unless the untrusted repository to be used is built internally.

4.3 Host Security

4.3.1 Hardening of Basic Host Security

Containers share the operating system kernel with the host. Therefore, host configuration determines whether containers can be executed in a secure manner. For example, vulnerable software puts the host at risk of arbitrary code execution; opening ports at will exposes the host to the arbitrary access risk; misconfiguration of a firewall downgrades the host's security; sudo login without key-based authentication may lead to brute-force cracking against the host.

To enhance the security of the container host, users should adhere to the following principles:

- Follow the minimum installation principle and do not install extra services and software to introduce more security risks.
- Configure login timeout for interactive users.
- Disable unnecessary packet forwarding functions.
- Disable ICMP redirect.

- Configure ranges of remotely accessible IP addresses.
- Delete or lock accounts unnecessary for device running, maintenance, and other related work.
- Set permissions for important files and directories.
- Disable unnecessary processes and services.

4.3.2 Hardening of Container-related Security

（1）Allocate a separate partition as the storage of containers.

By default, all Docker-related files are stored in the */var/lib/docker* directory. Where possible, a separate partition should be allocated for containers to ensure their security. Docker, after being installed, should be audited by running the *grep /var/lib/docker /etc/fstab* command.

（2）Harden the security of hosts.

Ensure that hosts comply with related security specifications by conducting effective vulnerability and configuration management.

（3）Upgrade Docker to the latest version.

Docker frequently releases updates to fix security vulnerabilities in earlier versions. Therefore, it is important to make sure that the currently used Docker version is free from known vulnerabilities and regularly checked for security risks.

（4）Control privileges for the Docker daemon.

The Docker daemon requires root privileges and grants users within the "docker" user group full root access. Therefore, on the container host, it is important to strictly restrict users in the "docker" user group and delete all untrusted users.

（5）Audit the Docker daemon.

For containers, it is necessary to audit not only the regular Linux file system and system calls but also the activity and use of the Docker daemon. By default, the Docker daemon is not audited. Auditing the Docker daemon requires addition of an audit rule by using the *auditctl -w /usr/bin/docker -k docker* command or a rule update by modifying the */etc/audit/audit.rules* file.

Auditing the Docker daemon will generate a large number of log files, which should be archived regularly. It is advisable to use a separate audit partition for log storage to prevent normal business from being affected by the root system filled with log files.

（6） Audit Docker-related files and directories.

Besides auditing the Docker daemon, it is also necessary to audit Docker-related files and directories, such as */var/lib/docker* (containing all container-related information), */etc/docker* (containing all keys and certificates for the TLS communication between the Docker daemon and Docker clients), *docker.service* (parameter configuration file of the Docker daemon), *docker.socket* (running sockets of the Docker daemon), */etc/default/*

docker (supporting various parameters of the Docker daemon), */etc/default/daemon.json* (supporting various parameters of the Docker daemon), and */usr/bin/docker-containerd* and */usr/bin/docker-runc* (the two are used by Docker to generate containers).

The method of auditing these files and directories is the same as that for auditing the Docker Daemon, namely, by modifying the configuration file or adding an audit rule with a command line.

4.4 Image Security

Images are the basis of containers. Therefore, their security speaks a lot for that of the entire container ecosystem. Container images are a series of images stacked layer by layer. They are distributed and updated through image repositories. The following sections describe how to secure images from the aspects of image build security, repository security, and image distribution security.

4.4.1 Image Build Security

Generally, Docker builds images either based on containers or with Dockerfiles. The recommended practice is to create all image files with Dockerfiles because images built this way are completely transparent and all operation instructions are controllable and traceable.

Image build is exposed to the following risks:

1. A base image pulled is not released by a trusted organization or person so as to possibly contain a backdoor or other types of risk.
2. A Dockerfile contains sensitive information, such as fixed passwords or credentials in plaintext used in service configuration.
3. Unnecessary software is installed, thus expanding the attack surface.

To avoid the preceding risks, image build security should be hardened from the following aspects:

（1）Verify the source of images.

To ensure that image contents are credible, users are advised to enable Docker's content trust mechanism. This mechanism provides digital signatures for data transmitted and received by remote image repositories, allowing clients to verify the integrity and publisher of image tags. By default, content trust is disabled. Users can use the following instruction or edit the Docker configuration file to enable this mechanism:

```
# export DOCKER_CONTENT_TRUST=1
```

（2）Make images lightweight.

Installing only necessary software packages is not only of great help for enhancing container performance but, more importantly, is useful in reducing the attack surface.

（3）Properly use image instructions.

Appropriate instructions should be used to build an image. If an external file is required, *COPY* is preferred to *ADD*

in a Dockerfile because the former only copies the files from the local host to the container file system, while the latter may download the file from a remote URL and perform such operations as decompression. This could bring in the risk of malicious files.

（4）Properly handle sensitive information.

Although Docker assigns read-only permissions to users, users should still exercise caution when handling data stored in containers. For example, a Dockerfile cannot store passwords, tokens, keys, and users' privacy information, which are at risk of leakage even if being deleted immediately after the container is built. This is because such data can be retrieved from historical records of images.

A recommended practice is to use the encryption management function provided by Kubernetes and Docker Swarm to encrypt data for transmission and storage, and allow only authorized users to decrypt it at the time of search.

4.4.2 Image Repository Security

4.4.2.1 Public Repository

Docker Hub is currently the largest platform for container image repositories. As mentioned in a previous section, over 30% of images on Docker Hub contain high-risk vulnerabilities. Therefore, while enjoying the convenience brought by Docker Hub, users should do as follows to ensure the security of images:

1. Use the latest images officially released and update them regularly.
2. Scan and assess images that are downloaded for vulnerabilities.
3. Perform OS-level and application-level scanning for images that provide services.
4. When public Dockerfiles are available, preferably create images with such a Dockerfile to avoid image backdoors and ensure the controllability of the image building process.

4.4.2.2 Private Repository

（1）Docker Registry

Docker Registry[56] is an open-source tool officially provided by Docker to help developers build private image repositories rapidly. The security of Docker Registry should be considered from the following aspects:

Security of Docker Registry itself. For example, when using the Registry, users should configure it to use a security certificate.

Security of the interaction between Docker clients and Docker Registry. This requires exercise of user access control. Private repositories exposed on the Internet usually grant the access permission only to specific organizations. In this case, it is insufficient to only verify the certificate of the Registry. Configuring passwords or bidirectional SSL is also required to verify the identity of Docker clients that interact with repositories.

（2）VMware Harbor

Users should do as follows when deploying Harbor in their production environments:

- Enable HTTPS and do not use the default password provided in *harbor.cfg*.
- Modify source code by adding a brute-force cracking protection mechanism to prevent passwords from being cracked as Harbor does not include such a mechanism to secure user login.
- Strictly control mounted volume permissions, which can be rw (default) or ro (optional).

4.4.3 Image Scanning

Containers are started from locally stored images for fast execution. Therefore, the security of images is directly linked with that of containers. While images downloaded from public repositories are exposed to security risks, as mentioned before, locally built images are not impervious due to the possible use of third-party libraries. Therefore, it is especially important to scan both downloaded and locally created images for security issues.

Image scanning engines that are currently popular include Docker Security Scanning (non-open-source), Clair (open-source), and Anchore (open-source).

Nowadays, image checking still revolves around known CVE vulnerabilities in systems. After capturing an image, the scanner separates it into layers and obtains a software package from each layer. Then it compares these packages with those in the CVE databases in terms of the name and version to determine whether they are vulnerable.

There are other schemes that identify malicious images by scanning environment variables, operation commands, and open ports in images. However, they do not directly tell whether an image is malicious, but require users to make a judgment based on the scanning result.

Overall, current so-called mature detection schemes are still limited and have much room for further development and improvement.

4.4.4 Image Distribution Security

When downloading and uploading container images, we should ensure their integrity and confidentiality. For this purpose, these tips should be followed to help withstand security threats such as man-in-the-middle attacks:

（1）Digital signature

The uploader adds a signature in the image to be uploaded. The downloader, after obtaining the image, should first verify its signature before use for fear that the image may be tampered with by malevolent actors.

（2）User access control

Sensitive systems and deployment tools (registration center, orchestration tool, ⋯) should have an effective mechanism to restrict and monitor users' access.

（3）Wherever possible, use image repositories that support HTTPS.

To avoid bringing in suspicious images, users should try to avoid using the *--insecure-registry* option to connect to any HTTP image repositories from untrusted sources.

4.5 Container Network Security

4.5.1 Network Security Mechanisms

Isolation and access control are two major protection means for computer networks. This section uses host networks in bridge mode and cluster networks in overlay mode as examples to discuss how these security mechanisms are implemented. In general, container networks are isolated by using the network namespace technology and iptables, while access control mainly depends on iptables.

4.5.1.1 Network Isolation and Access Control in Bridge Mode

According to Docker's design principle, in a bridge network, as long as two containers are connected to the same network bridge, they can access[1] each other without any access control or isolation mechanisms. To isolate the two containers from each other, users need to create different bridge networks to house the containers. The detailed process is as follows:

First, create bridge network *test*:

```
# docker network create --subnet 102.102.0.0/24 test
c11c01a07ed0ca3f4cdddee55e3e058e79c334d516b3a49dd3e56b86a4ff9302

# ifconfig | grep 102.102.0. -B 1
br-bff064219957 Link encap:Ethernet HWaddr 02:42:69:46:0b:21
inet addr:102.102.0.1 Bcast:102.102.0.255 Mask:255.255.255.0
```

From the preceding command output, you can see that *test*'s network bridge is *br-bff064219957*. After this network is set up, Docker will add *DROP* rules for the *DOCKER-ISOLATION* chain in iptables to block bidirectional traffic between *test* and other networks for the purpose of network isolation.

```
# iptables -t filter -L -v
Chain DOCKER-ISOLATION (1 references)
pkts bytes target prot opt in         out       source   destination
0 0 DROP all -- docker_gwbridge br-bff064219957 anywhere anywhere
0 0 DROP all -- br-bff064219957 docker_gwbridge anywhere anywhere
0 0 DROP all -- docker0         br-bff064219957 anywhere anywhere
......
```

A zero trust model requires that different containers should not communicate with each other by default. For this purpose, users can set the Docker daemon's startup parameter, *--icc=false*, so that packets from the *FORWARD* chain of the iptables will be dropped by default after the daemon is started. When the Docker daemon runs, users need to add appropriate access control policies that fit in with actual business requirements. For Cloud Workload Protection (CWP), such whitelist-based policies are quite important for deployment of mission-critical business environments as they can minimize the exposure surface and restrict connections of unknown business so as to

1 The default startup parameter setting of the Docker daemon, --icc=true, indicates that the inter-container communication is allowed by default.

reduce the likelihood of breach.

```
# iptables -nL
Chain FORWARD (policy DROP)
DROP all -- 0.0.0.0/0 0.0.0.0/0
......
```

The preceding inter-container access control policies can be created manually by using the *--link* parameter or the *docker-compose* command when containers are started.

External access to containers on the host is based on such a principle: Container-provided services are accessible via ports that are configured to be exposed to the Internet by using the *-p* or *-P* parameter. when and only when containers are started. Arguably, this access control mechanism is specific to hosts, with ports as the granularity of protection.

4.5.1.2 Network Isolation and Access Control in Cluster Mode

This section uses Docker Swarm as an example to describe the security mechanism for container networks in cluster mode. Put simply, Docker Swarm's overlay network complies with IETF VXLAN standards for isolation of different subnets. The following illustrates how the isolation and access control mechanism of Docker Swarm's overlay network works.

（1）Set up a cluster environment.

Prepare a cluster that contains two nodes.

```
# docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER      STATUS       ENGINE             VERSION
otsycjte53ve20byk79aukwl4 *     node1        Ready        Active Leader      18.03.1-ce
icd266rwlmpa47mghng1jvd7b       node2        Ready        Active             17.12.1-ce
```

（2）Create two overlay networks.

Create two overlay networks, i.e., *test0* and *test1*.

```
# docker network create -d overlay test0
vn86l7s11gfj9ttnzexhgl0yh
# docker network create -d overlay test1
w9p3tfmr09c6gw1to0guglfc0
```

```
# ls -l /run/docker/netns
total 0
-r--r--r-- 1 root root 0 May 28 00:56 1-3cv9ry2rsk
-r--r--r-- 1 root root 0 Jun 1 14:15 1-vn86l7s11g
-r--r--r-- 1 root root 0 May 9 19:08 default
-r--r--r-- 1 root root 0 Jun 1 14:17 1-w9p3tfmr09
-r--r--r-- 1 root root 0 May 28 00:56 ingress_sbox
……
```

Clearly, Docker creates a namespace for each overlay network. Such namespace names begin with "1-" followed by a string which is the prefix of the overlay network ID. For example, the namespace is *1-vn86l7s11g* for *test0*, *1-w9p3tfmr09* for *test1*, and *1-3cv9ry2rsk* for the ingress network. Note that those namespaces are named differently on different posts.

（3）Create services.

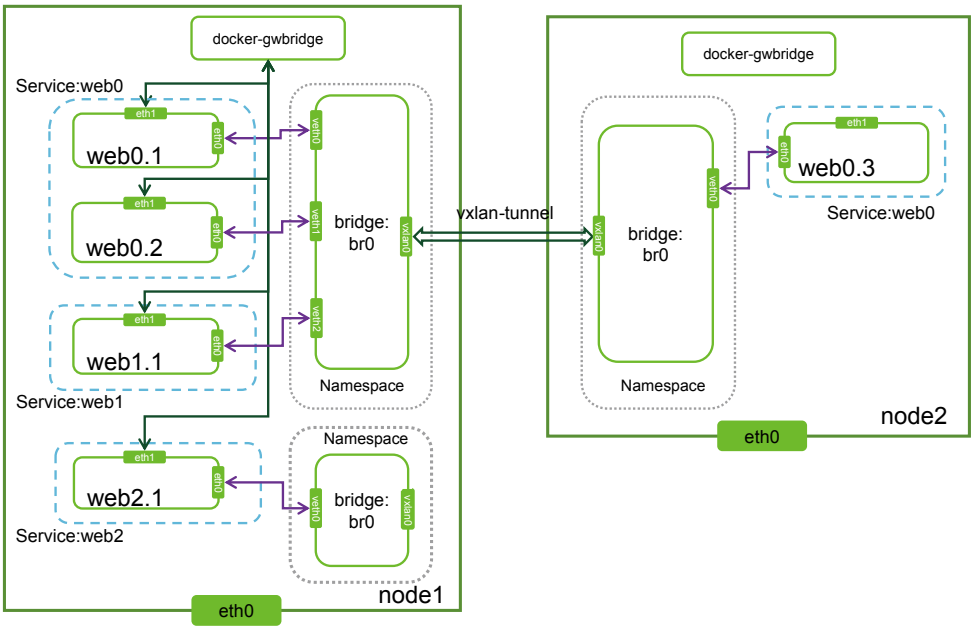
Create services, web0, web1, and web2, of which the former two share *test0* and the latter uses *test1*[1] .

```
# docker service create --name web0 --network test0 nginx:alpine
ugeorthtdyrf6axaxdigseyru
# docker service create --name web1 --network test0 nginx:alpine
i4rg9tdkuhv2maf9wzwdkclog
# docker service create --name web2 --network test1 nginx:alpine
wk34tw25bvdu9zxxdkbsfxv1w
```

1 In addition to the loopback interface, containers for the new services provide another two network interfaces, eth0 and eth1. The eth0 interface connects to the network bridge interface of the overlay network test0 or test 1 for inter-container communication across different hosts on the same network. The eth1 interface connects to the docker-gwbridge bridge network for the communication between containers and the host. If *--internal* is used during the creation of the overlay network, containers for services created with the *docker service create –network* command, besides the loopback interface, have only the eth0 interface which connects to the network bridge interface of the created overlay network.

Set the number of replicas to 3 for web0 and 1 for web 1 and web2 respectively. The cluster topology is shown in the following figure:

Figure 4.1 Swarm cluster topology



As shown in the preceding figure, web0.1, web0.2, and web0.3 are container replicas of web0; web1.1 and web2.1 are container replicas created respectively for web1 and web2. web0.1 is the short form of web0.1.u800qsqym guo9avo72w2o3ci0. For the sake of simplicity, suffixes of container instance names are completely removed in subsequent sections.

After the environment is set up, you can, based on services, find the network to which containers connect. Take web0 as an example.

```
# docker service inspect web0
…
"VirtualIPs": [
    {
        "NetworkID": " vn86l7s11gfj9ttnzexhgl0yh
",
        "Addr": "10.0.0.10/24"
    }
]
…
```

From the above, we can see that this service connects to the *test0* network. At least, container replicas of this service also connect to this network. If this service is provided externally, it will connect to the ingress network. For example, querying the *test0* network, we can find the server to which the service is connected.

```
# docker network inspect test0
[
    {
        "Name": "test0",
         "Id": "vn86l7s11gfj9ttnzexhgl0yh",
        "IPAM": {
             "Driver": "default",
             "Options": null,
            "Config": [
             {
                 "Subnet": "10.0.0.0/24",
                 "Gateway": "10.0.0.1"
             }
        ]
    },
         "Containers": {
            "759bbdf216d801615ff90578dd5a828e53449990a9016b82fdee1689cb7d448f": {
                 "Name": " web0.1.u800qsqymguo9avo72w2o3ci0",
                "EndpointID": "b03c9a96d81d91301594951fcc08ec0765102b37810c77ec8cb4943bd
                a4e9868",
                "MacAddress": "02:42:0a:00:00:06",
                "IPv4Address": "10.0.0.6/24",
             },
             "791de588518714ae1b3140956918c5d5adcae151f10ef3c9b5808ff0638ab357": {
                 "Name": "web0.1.n9pjx00w99fiwqch01gaiew90",
                "EndpointID": "8bfbb44db8e81734676d4756d7152a5ea3f57d7aae987600e238b5a70
                c15d670",
                "MacAddress": "02:42:0a:00:00:04",
                 "IPv4Address": "10.0.0.4/24",
            },
             "e6c02f73a9c72d23fcd5c17aa265adf1de56d1c933f396e6995b066c0b333f11": {
                 "Name": "web0.3.rmmfijc7jmr8grld6pwkm5e0q",
                 "EndpointID": "d223943dc20da5c57bcd19d9f2b743cd59cad00b6a84eae44048937
                ef3780324",
                 "MacAddress": "02:42:0a:00:00:05",
                 "IPv4Address": "10.0.0.5/24",
            },
            e9fc84b0d736107cec501436d08d4911dd4a516bfcce3e5b95016913ff2b8521": {
                 "Name": "web1.1.6f9chzfd711nmjenc6bgg109x",
                "EndpointID": "703622e4d0cd0ceca52c6b10a2c5451e6767845147625a21d18a7cf99
                b7cef56",
                "MacAddress": "02:42:0a:00:00:07",
                "IPv4Address": "10.0.0.12/24",
            },
        },
            "Peers":[
        {
            "Name": "17a9fb8e9cd2",
            }
        ]
    }
 ]
```

It is clear that a VXLAN tunnel is set up between Docker Swarm and the host. The above *test0* network information shows that on top of the underlay network comprising two hosts, 192.168.19.11 and 192.168.19.12, an overlay subnet, 10.0.0.0/24, is built, with the network gateway address of 10.0.0.1 (network address of *br0* of the namespace *1-vn86l7s11g*).

In addition, when both web0 and web1 connect to *test0*, their container replicas also connect to this network. As network interfaces in the namespace of *test0* have no VLAN IDs, no VLAN is set for these interfaces on the network bridge and containers on one host in the same network can access on another without any isolation measures.

```
# s1-vn86l7s11g bridge vlan
port    vlan ids
br0     None
vxlan0  None
veth0   None
veth1   None
# nsenter --net=/var/run/docker/netns/1-vn86l7s11gbridge fdb show dev vxlan0
22:71:ed:59:a9:59 master br0 permanent
02:42:0a:00:01:09 dst 192.168.19.12 link-netnsid 0 self permanent
02:42:0a:00:01:08 dst 192.168.19.12 link-netnsid 0 self permanent
02:42:0a:00:01:06 dst 192.168.19.12 link-netnsid 0 self permanent
```

On computing nodes in OpenStack, different VLANs are set to isolate subnets of different tenants on the network bridge connected to the VXLAN tunnel. In contrast, in Docker Swarm as shown above, networks are isolated by using network namespaces, instead of VLANs.

For example, container replicas of web0 connect to the *test0* network. This means that packets, which arrive from containers to the network bridge *br0* of the namespace *1-vn86l7s11g*, can only be sent to other container replicas (containers connected to *br0* in the *1-vn86l7s11g* namespace of this host or other hosts) that are connected to the *test0* network, instead of containers connected to the *test1* network. This can be verified as follows:

```
# docker exec web0.1 ping 10.0.0.12  -c 1
PING 10.0.0.12 (10.0.0.12): 56 data bytes
64 bytes from 10.0.0.12: seq=0 ttl=64 time=0.250 ms
--- 10.0.0.12 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.194/0.194/0.194ms

docker exec web0.1 ping 10.0.1.5  -c 4
PING 10.0.1.5 (10.0.1.5): 56 data bytes
--- 10.0.1.5 ping statistics ---
4 packets transmitted, 0 packets received, 100% packet loss
```

(4) Control access to containers in a cluster.

As previously mentioned, in Docker Swarm, containers in the same overlay network can access one another by default, while those in different overlay networks cannot. There are no customizable access control policies for containers. If access control policies need to be applied explicitly, users need to create them manually.

Now, we create a service exposed to the external network:

```
# docker service create --name web5 --network test1 --replicas 3 -p 3000:80 nginx:alpine
c3ql2e1e0g7xbxksfz3pjlrgb
```

As the *-p* parameter is selected during service creation, four network interfaces will be created for containers of this service.

The eth0 interface connects to the ingress network which acts as an intermediary between external networks and containers.

The eth1 interface connects to the docker_gwbridge network. This network provides the communication from containers to an external network. For example, when the *ping www.baidu.com* command is executed in a container, the traffic will go through the default_gwbridge network.

The eth2 interface connects to the *test1* network for communications between container replicas across host services.

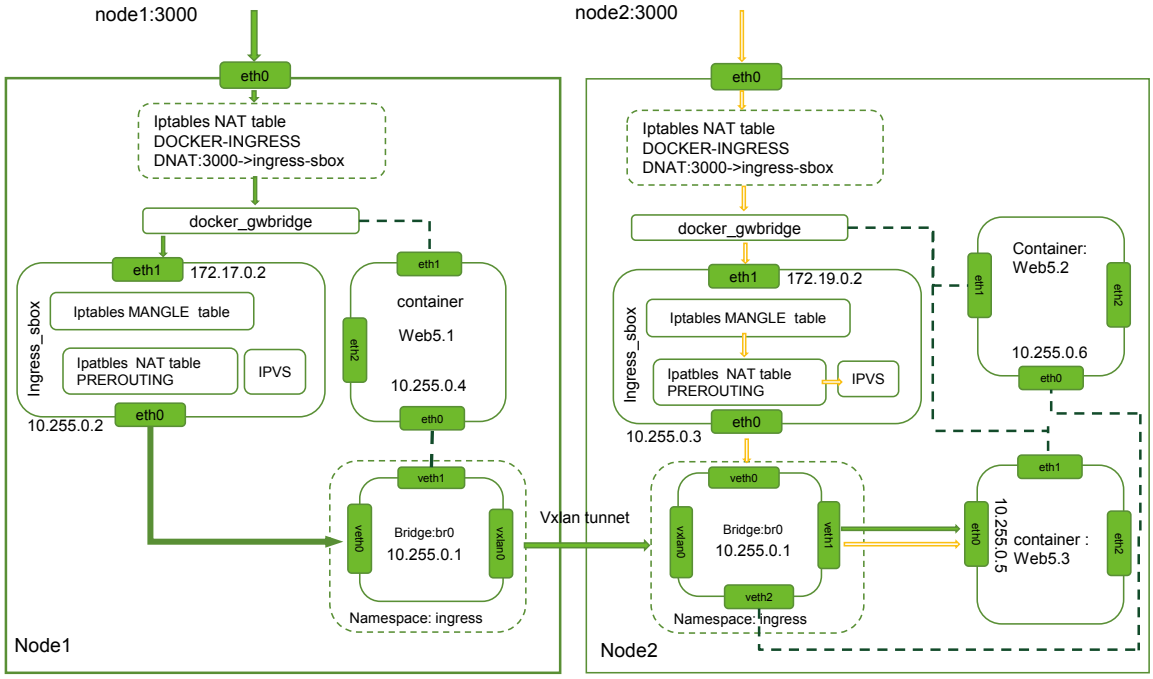
This service is available on port 3000 exposed externally, while the container listens on port 80. External access to port 3000 is achieved through port mapping.

```
# iptables -t nat -n -L | grep -A 4 'DOCKER-INGRESS'
--
Chain DOCKER-INGRESS (2 references)
target prot opt source destination
DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 tcp dpt:3000 to:172.17.0.2:3000
RETURN all -- 0.0.0.0/0 0.0.0.0/0
```

Passing through the iptables of the host, the packet arrives at the namespace *ingress_sbox* after going through the network bridge *docker_gwbridge*. After IPVS load balancing, the destination IP address will be translated into the IP address of the container for which the packet is destined when being transmitted in the ingress network.

It can be seen that to control external access to the service, access control rules can be deployed for data channels between the *eth0* interface on the host and the destination container. For example, create access control lists (ACLs) for the *DOCKER-INGRESS* chain in the *iptables* table or deploy a firewall outside the host. Of course, whichever method is used, access control rules should be adjusted in light of changes to containers. This requires full integration between the access control management mechanism and the container orchestration system. The following figure shows the external access to port 3000 on node1 and node2. Arrows in green and orange indicate the entire process of packets passing through different host ports to the Web5.3 container.

Figure 4.2 Cluster data transmission process of Docker Swarm



4.5.1.3 Micro-Segmentation

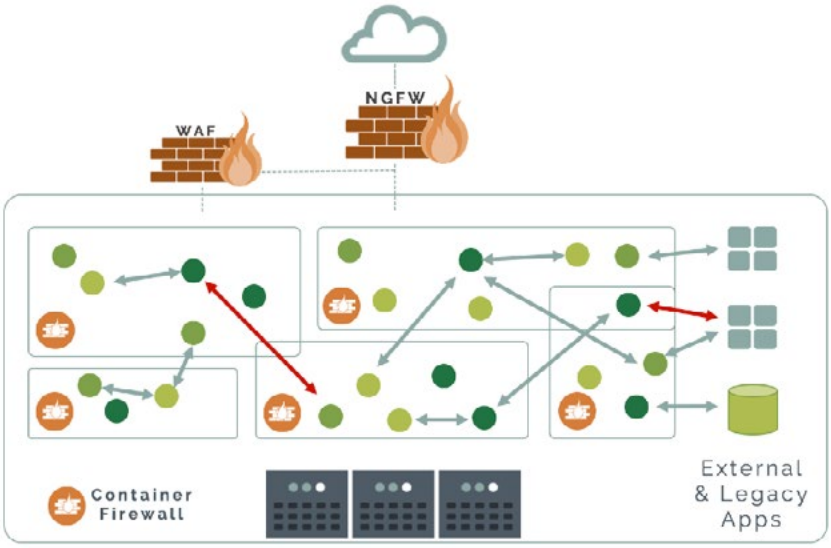
Service software's transition from individual applications to a lot of container-based microservices brings many benefits and also changes the internal communication mode of software. As for network and security, two noticeable changes are observed: 1. The communication traffic surges horizontally. 2. Borders become much more blurred. Even if each running container can be hardened and a limit can be set on the number of interfaces for external communication, as the total number of interfaces increases dramatically, more opportunities are opened up for network attackers to detect vulnerabilities.

Besides, containers can be rapidly deployed within seconds. The container orchestration system can automatically start new containers on one or more hosts according to the actual resource situation. Each container has its own network mapping interface which can be reallocated and unbound in the course of running, especially because containers have a very short lifecycle, with 17% lasting less than one minute and 78% disappearing within an hour[57]. Therefore, container protection should be agile and elastic.

In a dynamically changing container environment, traditional network firewalls could hardly detect any inter-container network traffic and also cannot adapt to the constantly changing situation as containers start and disappear rapidly. As one cybersecurity architect said, "In a containerized world, you can never manually configure iptables or update firewall rules."

Given all above analysis, in a cloud-native environment, cloud-native container firewalls are required to isolate and protect application containers and services. Even if containers dynamically expand or shrink, such firewalls can still detect, track, and protect them. Like traditional gateway firewalls, container firewalls can also protect network communication that is from external networks and traditional applications to the container environment.

Figure 4.3 Container firewall in micro-segmentation



Micro-segmentation is a kind of segmentation technology which is more fine-grained than traditional segmentation based on network addresses. For example, this technology can implement isolation and segmentation for individual containers, container collections or container applications in the same network segment. With micro-segmentation as an essential feature, container firewalls can perceive layer 7 or the application layer and provide dynamic control of connections, depending on upper-layer applications. We can see that this kind of firewalls implements dynamic micro-segmentation for services and therefore has become the first line of defense to protect containers in horizontal traffic scenarios from malicious attacks.

Container firewalls mainly protect inter-container network sessions in horizontal scenarios, and thus will not replace protective systems, like NGFW, IDS/IPS, and WAF, deployed at the entrance of the data center. On the contrary, container firewalls, through collaboration with those traditional firewalls, can effectively block attacks that are initiated from internal applications.

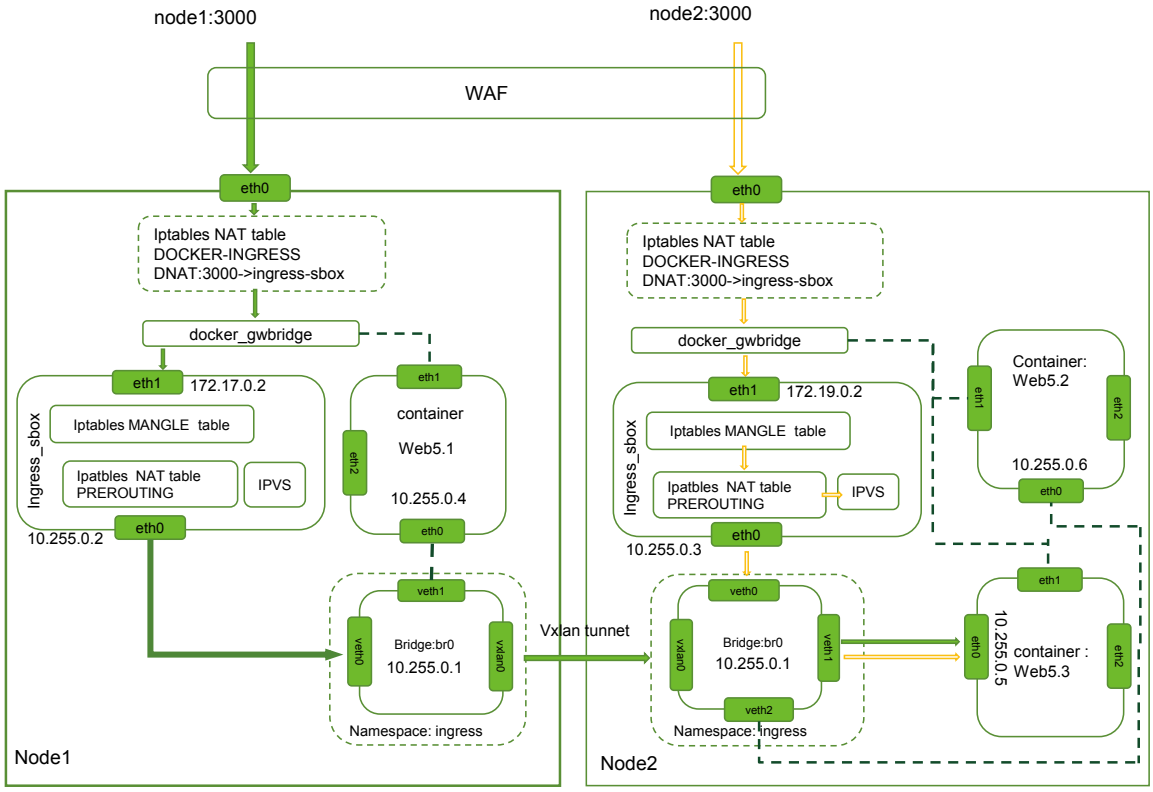
4.5.2 Security Protection for the Container Network

4.5.2.1 Web Application Security

Typically, external microservices take the form of web or HTTP applications and services, for example, websites or applications with support for REST APIs. For the sake of security, you can deploy web applications to detect and block malicious HTTP requests. This section describes how to perform cybersecurity protection for web applications. For the security of other applications, see the threat detection section.

Microservices can receive traffic from vertical and horizontal directions. Take node1 as an example. As shown in the following figure, the traffic from the external network to the Web5 service is vertical. To protect the traffic, you can deploy a web application firewall on the external network.

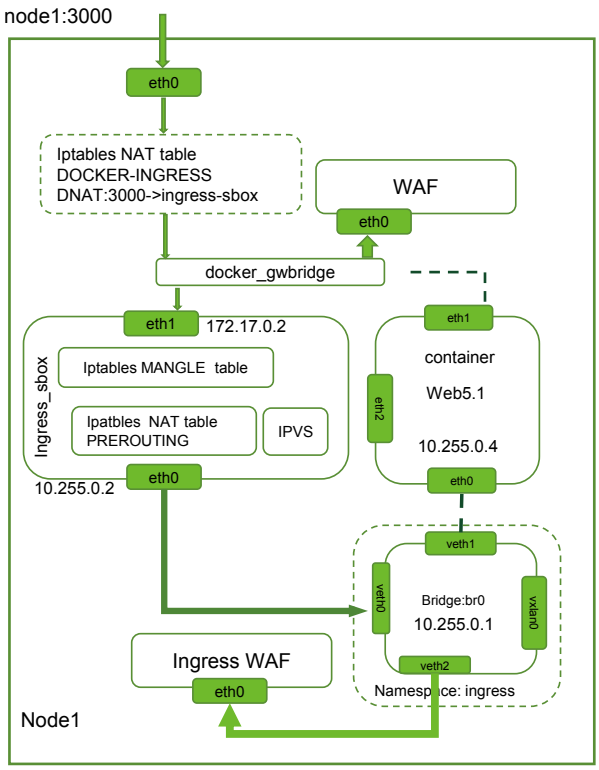
Figure 4.4 Example for vertical traffic protection of web applications



The container environment also contains a great number of API calls, or even you might say that the traffic of a data center mainly comprises invocation requests and responses of such APIs. Such traffic is called horizontal traffic. To protect the traffic, you need to deploy virtual web application firewalls (vWAF or cWAF) in the internal network of a host. As the container management and orchestration system will create or delete container instances dynamically according to actual business requirements, it needs to connect to the container management and control plane and has vWAF deployed to track dynamic network changes.

Take node1 as an example. Web5 connects to the 10.255.0.0/24 and 172.17.0.0/24 networks. Therefore, for protection of Web5's container replica (Web5.1 in the following figure) on the node1 host, you can deploy a vWAF respectively at the network bridge docker_gwbridge and the network bridge br0 in the namespace *ingress*. In this case, vWAF, in out-of-path mode, can detect malicious requests by mirroring traffic, and in in-path mode, can filter traffic for protection.

Figure 4.5 Example for horizontal traffic protection of web applications



The software-defined networking (SDN) technology can be introduced to implement flexible traffic scheduling so that fewer WAFs need to be deployed for intended security protection. For details, see the *2015 NSFOCUS SDS White Paper* [58].

4.5.2.2 Network Security Protection

Each host and server have a container firewall deployed to gain full access to the local Docker daemon. As the container firewall resides within the container network, it can easily detect abnormal behaviors on the container network, especially horizontal inter-container movements of malicious attackers. Also, a machine learning module can be integrated in the container firewall to automatically learn security policies for effective container security protection.

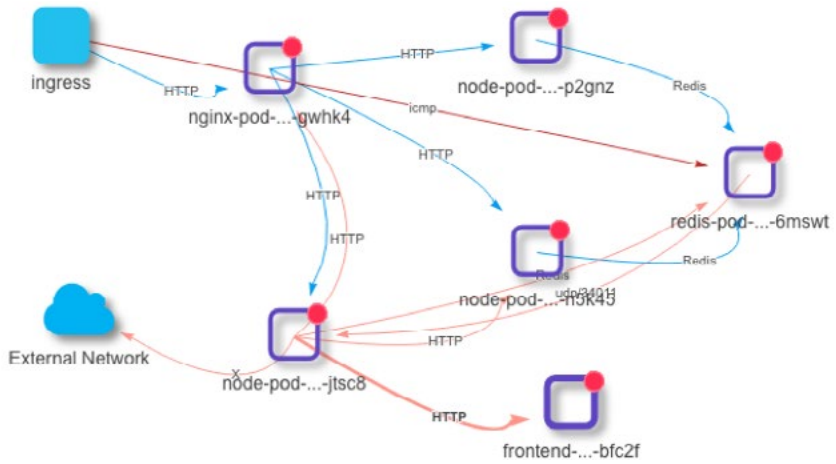
Deployed in a distributed manner, a container firewall can provide effective local monitoring and protection. Seamlessly integrating with the Docker engine and container orchestration management tools, the container firewall, besides access control, can also provide security audit, security testing, resource monitoring for

hosts and containers, and the like. From this point of view, the container firewall is essentially a security policy strengthening point in the container network, rather than the traditional firewall in the narrow sense. This security policy strengthening point provides the following functions:

- Deep inspection of and visibility into container network communications to gain accurate application knowledge for container protection.
- Host process monitoring, process privilege escalation monitoring, and suspicious process detection.
- Static scanning or dynamic real-time vulnerability scanning, image repository scanning, host scanning, and vulnerability scanning for running containers.
- Container audit based on CIS's security baselines.
- Capture of raw packets of containers for forensics and debugging.

To ensure that containers can be deployed and operate in a secure way, container firewalls should be able to support virtual workloads, automatically learn behaviors of applications, generate smart security policies, and seamlessly integrate with the container orchestration platform.

Figure 4.6 Detection and visibility into container network communication



When containers operate, the network security detection engine needs to detect the following items:

- Unauthorized network connections
- Trusted IPs/ports used in an unauthorized manner
- Known cyberattacks against applications
- Data theft, reverse shell channels, and hidden network pipelines
- Container process privilege escalation and malicious processes within containers and the container host
- Unauthorized network ingress/egress control

4.6 Runtime Security

4.6.1 Security Configuration for Container Launch

A container runs on the host as a process. Running container processes are isolated from one another. Each has its own file system, networking, and isolated process tree separate from the host. The following sections detail how to use the *docker run*[1] command to define a container's resources at runtime.

4.6.1.1 Kernel Option Configuration

（1）Enable AppArmor.

AppArmor protects the Linux system and applications from various threats by executing security policies, which are reflected in AppArmor profiles (for details, see section 4.1.4.2). Users can create their own AppArmor profiles for containers or use Docker's default AppArmor profile.

```
# docker run --security-opt=”apparmor=$PROFILE”
```

（2）Set SELinux.

SELinux provides an effective Linux access control mechanism. When starting the Docker service, users are advised to configure SELinux. When running a container, users are also advised to use this security option. Creating or importing an SELinux policy template can effectively secure the Docker container.

```
# docker run --security-opt lable=level:TopSecret
```

（3）Restrict the use of kernel capabilities.

The Linux kernel capabilities mechanism supports grant of only necessary kernel capabilities within a Docker container. With unnecessary capabilities removed, the attack surface is narrowed, which, in turn, enhances the security of containers. For example, *NET_ADMIN*, *SYS_ADMIN*, and *SYS_MODULE* capabilities can be removed as they are usually not required for containers.

Users can run the following commands to add or delete capabilities as required:

```
# docker run --cap-add={“NET_ADMIN”,“SYS_ADMIN”}
# docker run --cap-drop={“NET_ADMIN”,“SYS_ADMIN”}
```

A recommended practice is to delete all capabilities first by using the following command and then add those that are required:

```
# docker run --cap-drop=all --cap-add={“NET_ADMIN”,“SYS_ADMIN”}
```

Besides, particular attention should be paid to the *--privileged* option, which provides all Linux kernel capabilities for a container. At the same time, it lifts all bans on cgroups devices' execution permissions, which means that the container is allowed almost all the same access to the host as processes running outside containers on the host. Therefore, unless absolutely necessary, users are advised not to set a container to privileged mode. A

1 https://docs.docker.com/engine/reference/run/

process can set the *no_new_privs* bit in the Linux kernel, which ensures that the process or its child processes do not gain any additional privileges through *suid* or *sgid* bits.

```
# docker run --security-opt="no-new-privileges:true”
```

（4）Do not disable the default seccomp profile.

Secure computing mode (seccomp) is a security feature of the Linux kernel since version 2.6.23. On the Linux system, most system calls are directly exposed to user mode programs. But, not all system calls are required. Besides, insecure code's abuse of system calls would impose a security threat to the system. seccomp restricts a program's use of certain system calls, thereby reducing the system's exposure to threats and at the same time putting the program into "secure" mode.

By default, Docker enables the *seccomp* profile. Since Docker 1.10, the default profile prevents some of the system calls even if related capabilities have been added for a container by using the *--cap-add* command. In this case, users can use a custom seccomp profile to gain privileges for kernel system calls based on the whitelist mechanism.

```
# docker run --security-opt="seccomp=profile.json"
```

（5）Set ulimits when necessary.

Generally, the Docker daemon uses default configuration for container resource allocation. If a container requires custom configuration in terms of resource usage, the *--ulimit* option can be set to overwrite the default configuration when the container is launched.

```
# docker run --ulimit nofile=1024:1024
```

（6） Do not share kernel namespaces.

Linux implements isolation through kernel namespaces. For example, it uses the process ID (PID) namespace to isolate processes. That is to say, processes in different namespaces can have the same PID. If a container has the same PID namespace as the host, it is possible to view or even kill all processes on the host from within the container.

This is true for other types of namespaces. If containers share the namespace with the host, the isolation of containers falls void. Therefore, when launching a container, users should be cautious about the use of namespace-related parameters, such as *docker run --pid=host* (uses the PID namespace of the host), *docker run --ipc=host* (uses the IPC namespace of the host), and *docker run --uts=host* (uses the UTS namespace of the host).

4.6.1.2 Resource Management and Control

To prevent DoS attacks caused by system resource exhaustion, Docker can use specific parameters to limit containers' usage of resources, such as the CPU, memory, or disk usage, thereby achieving monopoly or control of resources.

（1） Limit containers' memory usage.

By default, each container on the Docker host can use the host's available memory resources in an unlimited manner. Users can limit a container's memory usage to prevent the container from exhausting host resources, thus leading to a denial of service, which may further affect normal running of other containers on the same host.

Specifically, the *-m* or *--memory* parameter can be used with the docker run command:

```
# docker run --memory 256
```

（2）Limit containers' CPU usage.

By default, Docker uses time to allocate the CPU resource evenly between all containers on the same host. Users can use the *--cpu-shares* option to set the priority. The CPU share mechanism allows a container to take precedence of another for CPU usage and forbids containers with a lower priority level to frequently use the CPU resource. This guarantees the proper running of higher-priority containers and effectively prevents CPU resource exhaustion.

For example, the *-c* or *--cpu-shares* parameter can be used with the *docker run* command:

```
# docker run --cpu-shares 512
```

By default, each newly created container has a CPU share of 1024, that is, 100% of the CPU resource. In the preceding example, the CPU share for the container is set to *512*, indicating that this new container will be executed with 50% of the CPU share allocated to other containers.

（3）Limit containers' storage usage.

If data isolation is not achieved between containers or between the host and containers on the host, attackers can easily obtain important data, causing security risks. A better solution is to adopt the copy-on-write (CoW) feature, allowing all running containers to share a base file system. File systems in containers interact with the base file system, thus avoiding risks arising from data shared between containers and at the same time achieving data isolation.

Moreover, strict control should be exercised over containers' mounted directories. It should be prohibited to use a sensitive directory on the host, such as */dev*, */boot*, */etc*, or */sys*, as the container volume for mounting. In addition, users should be especially cautious when sharing host devices to containers through *--device*. This is because, be default, containers have read/write access to these devices, which makes it possible to delete devices from the host. Therefore, it is not advisable to directly share host devices to containers. If this has to be done, permissions should be properly restricted in this regard.

4.6.1.3 Network Configuration

（1）Do not use the host networking mode on Docker hosts in production environments.

The default bridge networking mode requires setup of its own network protocol stack by using the virtualization technology, while the host networking mode uses the host's local network stack, thus delivering better network

performance. Therefore, when high network performance is required, containers may run in host mode in some scenarios.

But what deserves our attention is that containers, when using the host's network stack, can "see" all interfaces of the host, thus providing containers with full access to local system services, such as D-Bus. For this reason, this mode is considered insecure and not recommended for use in production environments.

（2）Do not use the default bridge docker0 of Docker.

In the default bridge mode, all containers on the Docker host are connected to one another via the *docker0* bridge. This mode brings in the risks of ARP spoofing, sniffing, and broadcast storm attacks on the local area network (LAN), thus threatening the security of containers. Therefore, users are advised not to use the default *docker0* bridge when running containers.

It is recommended that a custom network be used to interconnect containers. For example, users can run the following commands to set up their own network on the Docker host:

```
# docker network create --subnet 102.102.0.0/24 test
# docker run --network test  ...
```

Alternatively, a container cluster management platform, such as Docker Swarm or Kubernetes, can be leveraged to create an overlay network of clusters.

```
# docker network create -d overlay overlaynetwork-test
# docker run --network overlaynetwork-test ...
```

（3）Map only necessary ports.

There are some things to be noted regarding port usage. On the one hand, by default, common users should not use privileged ports with numbers below 1024 on Linux. Privileged ports are usually used to receive and send various types of sensitive and privileged data. If container ports are mapped onto privileged ones on the host, a severe consequence may ensue.

Of course, it is necessary to bind the HTTP service to port 80 and the HTTPS service to port 443, which is not within the scope of our discussion here. Such privileged ports include ports 22 (SSH), 21 (FTP), 25 (SMTP), and 110 (POP3 email service).

On the other hand, Dockerfiles of container images must be strictly audited to ensure that only necessary ports are exposed.

（4）Do not run unnecessary services, such as SSH, in containers.

It is not unusual that people, when using containers, tend to treat them like virtual machines. For example, they often consider how to enter a container for debugging and how to configure sshd. To properly use containers, we must have a mindset of containerization and look at containers as they really are. In particular, in the microservice system, containers, by nature, represent all kinds of dependencies necessary for one or more processes and the execution of processes, that is, the minimum set of runtime environments.

Execution of such services as SSH and Telnet in a container does not add any functions to the microservice. Instead, it brings in a series of security threats and at the same time complicates security O&M, which will involve, for example, the access policy and security compliance management of SSH, key and password management of containers, and security upgrade of the SSH service. Therefore, in production environments, SSH and other unnecessary services should be excluded from containers.

4.6.1.4 Other Configurations

（1）Configure a container restart policy.

If no container restart policy is configured, the restart service will continuously attempt to restart the container until it succeeds, which, in extreme situations, may crash the host.

When a container exits due to an error, the right thing to do is limit the number of container restart attempts and find the root cause instead of attempting to restart the container infinitely. For Docker, this number can be limited by setting the *on-failure* value. For the sake of security, it is recommended that this value be set to *5*.

```
# docker run --restart=on-failure:5 redis
```

（2）Do not run the container service as root.

The cgroups mechanism allows containers' root users root access to the host when containers run as root. To prevent attacks launched by exploiting this flaw, a viable solution is to avoid allocating too many resources to a single container. Therefore, the container service should be launched by a common user rather than root.

（3）Check the health status of running containers.

If the *HEALTHCHECK* command is not specified in container images, users are advised to add the *health-cmd* parameter to check the health status of running containers. For example, the parameter can be set as follows:

```
# docker run --health-cmd = ‘stat /etc/passwd || exit 1’
```

（4）Restrict Linux kernel capabilities within containers.

Default Linux kernel capabilities on Docker include chown, dac_override, fowner, kill, setgid, setuid, setpcap, net_bind_service, net_raw, sys_chroot, mknod, setfcaph, and audit_write. Unnecessary ones can be removed by using the *--cap-drop* command and, when necessary, be added by using the *--cap-add* command.

4.6.2 Runtime Security Monitoring and Audit

The monitoring and audit mechanism is a common approach to ensuring service security. Please read on to learn how to perform security monitoring and audits in the container environment.

4.6.2.1 Runtime Monitoring

Unlike conventional environments, the container environment requires monitoring and auditing at the levels of hosts, container instances, and images to ensure the stable running of containers.

（1）Host monitoring

Host monitoring in the container environment is the same as that in conventional computing environments. It generally covers the host's operating system, CPU, memory, processes, file system, and network status. In O&M, host monitoring is nothing new and there are quite a few open-source tools available to implement it, such as Performance Co-Pilot[1] , Icinga[2], and Munin[3] .

（2）Container instance monitoring

Monitoring of container instances covers basic information of containers on the host (container ID, image name, time of creation, status, port information, and container name), and usage of various resources by containers, including the CPU, memory, block I/O, and network resources. The *docker stats* command can be used to view the preceding information:

```
# docker stats 9d83d3116584
CONTAINER ID   NAME          CPU%   MEM USAGE/LIMIT    MEM%   NET I/O       BLOCK I/O       PIDS
9d83d3116584   dev_proxy_1   0.00%  14.86MiB/125.9GiB  0.01%  286MB/286MB   14.3MB / 8.19kB   25
```

4.6.2.2 Malicious Behavior Monitoring

Figure 4.7 Multidimensional monitoring of malicious behaviors

1 https://pcp.io/
2 https://www.icinga.com/
3 http://munin-monitoring.org/

（1）Container monitoring

Attackers can exploit vulnerabilities in and incorrect configurations of applications to break into a container and then use it as a springboard to infiltrate the internal network, in hopes of finding weaknesses in processes and file systems of other containers or the host for further exploitation and attacks. To address this issue, it is necessary to monitor containers for suspicious processes, such as the port scanning and reverse shell processes. This type of monitoring covers the following objects:

Processes running in containers

Containers, after being compromised, will start malicious processes, such as the cryptocurrency mining software or network port scanning process. Ongoing monitoring finds that it has grown into a trend to perpetrate such malicious activities in the container environment.

In a lightweight container environment, usually each container runs a limited number of normal processes that have been expressly defined and are relatively stable and consistent. Therefore, the whitelist security policy at the process level can effectively detect and block abnormal or malicious processes.

Specifically, process monitoring can be conducted as follows:

1. Scan the */proc* directory.

/proc is a virtual file system in the Linux environment. For each running process, a directory named with the process ID is generated in the /proc directory. Files in this directory provide all information about the running process. For example, the directory of a process usually contains the following files or directories:

- **/proc/pid/cmdline:** contains the command line used to execute the application.
- **/proc/pid/cwd:** contains a link to the current working directory of the process.
- **/proc/pid/environ:** contains a list of environment variables for the process.
- **/proc/pid/fd:** contains descriptors of the files that the process has opened.
- **/proc/pid/maps:** contains the mapped executables, libraries, and memory used by the process.

In the */proc* directory, there are other files that provide more information about a process. By scanning this directory, we can have a clear picture of which processes are running in the system and containers.

2.Monitor process activities through Netlink sockets.

By scanning the */proc* directory, we can obtain detailed information of processes. But if a large number of processes are running in the system, repeatedly reading the file system will generate unnecessary overhead. A desired solution is to have process status changes notified in real time, which will reduce system resource usage and improve the response speed.

Luckily, the Linux system provides such an asynchronous notification mechanism, which is implemented via Netlink messages. Netlink is used to transfer information between the kernel and user-space processes. Compared with other communication mechanisms like system call (syscall) and input/output control (ioctl),

Netlink has a smaller impact on the kernel and is easy to develop and maintain. In the programming interface of the user layer, it uses the socket process. Besides, it provides broadcast and multicast functions so that multiple processes can obtain kernel status changes at the same time. All these advantages make Netlink superior to other communication mechanisms.

The Netlink socket uses the address family AF_NETLINK. To obtain process information, users need to specify the NETLINK_CONNECTOR protocol when opening a socket and then send a Netlink message to subscribe to process activity reports. Specifying different protocol IDs allows the user-layer program to monitor behaviors of key components in the kernel, including the routing table, ARP, and iptables.

3. Use the system performance analysis tool Perf to learn about process activities.

The Perf event subsystem was intended to debug and diagnose the performance of the Linux kernel and applications. It provides an exhaustive report and statistics of system behavior parameters, including the analysis of system hardware events such as the CPU clock, pipeline instruction execution, and interruption events, and the report of software behaviors such as process switching and operations regarding the memory, files, and sockets.

For process-related events, the Perf event subsystem generates reports on PERF_RECORD_FORK and PERF_RECORD_EXIT events. From such reports, users can not only obtain statistics about event counts but also read event details by means of sampling from the shared memory allocated by using mmap. To obtain system information through Perf events, a thorough analysis should be first conducted on the frequency of events in normal and extreme conditions (for example, during attacks) to avoid false negatives or resource overflows.

4. Use eBPF to monitor process activities.

eBPF is short for extended Berkeley Packet Filter. It is a powerful tool that allows programs written by users in certain syntax to be put in the kernel and triggered for execution when a given event happens. The kernel checks the programs to ensure that infinite loops and out-of-bounds operations will not occur. eBPF was originally designed to filter network packets, but, with functions gradually enhanced, has now grown into a secure, full-featured, and high-speed tool in the Linux kernel. eBPF not only controls network flows but also filters system calls to enhance the system monitoring and debugging performance.

Working with Perf events or Netlink Connector, eBPF can monitor system process activities. This significantly improves the performance of detection software and the accuracy of detection results.

File systems of containers

Attackers can forge software packages or plant malicious code in open-source software libraries/software packages. If imported without verification and then loaded by container processes, such malicious packages may tamper with sensitive files in containers. Once successful, they will attempt to escalate their own privileges to root for further infection. For example, a recent discovery reveals that an open-source SSH library contains malicious code that is used for stealing users' passwords. This library has been widely used in many commercial applications and containers over years.

Therefore, the container security system should trace every change in file systems and ensure the consistency

of important system files. At the same time, it should automatically trigger vulnerability scanning and generate alerts.

Data in containers

Attackers try to steal data in containers usually by employing various techniques, including creating a reverse shell to receive and mine data, and encrypting confidential data or sending confidential data stealthily. The traditional data loss prevention (DLP) strategy cannot effectively protect data in containers. The security system should use next-generation container DLP, real-time container behavior monitoring, and kill chain event analysis techniques to detect theft of such data.

(2) Host monitoring

Host monitoring plays a major part in container security because the host of containers is at risk of privilege escalation and isolation policy breach that may threaten the security of containers. In addition, suspicious behaviors in containers may sometimes be detected from the host. For example, when the port scanning or reverse shell process starts running in a container, the host's security system can usually detect such exception in the network and process before immediately generating an alert and responding to the event.

The host running containers may be exposed to various attacks from the container layer. For example, a Linux kernel vulnerability exploited by DirtyCow[59] allows users to escalate their privileges from container root to host root. Then they can escape from the container to the host node, posing a severe impact on the security of the entire container environment.

(3) Image monitoring

As a basis for container execution, images saved on the host should also be monitored to prevent them from being maliciously tampered with (for example, change of exposed ports or planting of a virus/trojan). Image information under monitoring is generally static, including the number of images, image ID, image name, version, size, historical build information, and exposed ports. The *docker images inspect containerid* command can be used to view the current information of images, which can then be compared with historical information for any suspicious changes.

```
# docker image inspect c8c29d842c09
[
    {
        "Id": "sha256:c8c29d842c09d6c61f537843808e01c0af4079e9e74079616f57dfcfa91d4e25",
        "RepoTags": [
            "nginx:1.9"
        ],
        "RepoDigests": [
            "nginx@sha256:54313b5c376892d55205f13d620bc3dcccc8e70e596d083953f95e94f071f6
            db"
        ],
    }
……
]
```

（4）System container monitoring

The container platform hosts a large number of auxiliary/management container instances, which are special system containers with higher privileges and more diversified and complicated communication methods and behaviors. The security of such system containers is also of significant importance. For example, the Tesla[60] container security breach discovered in early 2018 was an event caused by the hacking of its administrative console for Kubernetes. After infiltrating Tesla's Kubernetes, the hacker deployed cryptocurrency mining software in some containers for profiteering. Besides, lots of Tesla's technical data under management of Kubernetes was disclosed. Therefore, the container security system should put system containers' network and behaviors under monitoring as well.

4.6.2.3 Configuration Audit

Section 4.6.1 provides recommendations on security configuration for launching containers. Such information should be monitored and audited for security at runtime.

The *docker inspect* command can be used to get underlying details of images or container instances, thus presenting complete build information for security audits. This detailed information includes basic configuration, host configuration, network configuration, and status information of images or containers.

For example, users can run the following commands to audit the exposure of a container's ports:

```
# docker ps --quiet | xargs docker inspect --format '{{.Name}}:Ports = {{.NetworkSettings.
Ports}}'
/floodlight.1.xqcd9jio5p0j4jax56a0sz5uc:Ports = map[4200/tcp:[] 6633/tcp:[] 6655/tcp:[]
8081/tcp:[] 9001/tcp:[]]
```

Moreover, some open-source configuration verification projects can also be helpful for configuration checks. For example, docker-bench-security (see section 5.1.5 for details) is suitable for checking configuration parameters.

4.6.2.4 Image Vulnerability Assessment

The entire lifecycle of images goes like this: building an image → uploading it to a repository → downloading it to a local host → running container instances. From this lifecycle, we can find that the local container host stores images and container instances, which should be subject to vulnerability assessment for the sake of security.

For image vulnerability assessment methods, see section 4.4.3. The Docker image scanning database presents scanning results in a bill of materials (BOM) containing details of images along with the security profile of each component. When a new vulnerability is reported to the CVE databases, the Docker image scanner checks which image-related packages are affected by this vulnerability and then pushes the result to the administrator for timely remediation and update of affected images. Assessment vulnerability of container instances can also be conducted by using cloud-hosted scanners.

4.7 Orchestration Security

The maturity of the container technology pushes the development and implementation of microservices. More and more enterprises choose to adopt a mircoservice architecture to build their applications. Container orchestration tools are responsible for managing container clusters that carry various services. Arguably, it is container orchestration tools that support core services in a variety of projects adopting a mircoservice architecture. This document takes the most popular orchestration tool in the community, Kubernetes, as an example to describe security protection measures that container orchestration tools should take.

4.7.1 Kubernetes Security Protection

Both the Kubernetes community and third-party security management authorities have improved and enhanced the security of components and resources of Kubernetes. Main improvements are as follows:

4.7.1.1 Computing Resouce Security

The Kubernetes community provides a Pod Security Policy[61], which is a cluster-level resource that controls security sensitive aspects of the pod specification. The PodSecurityPolicy objects define a set of conditions that a pod must run with in order to be accepted into the system, as well as defaults for the related fields. They allow an administrator to control the following:

Table 4.1 PodSecurityPolicy fields

Control Aspect	Field Name
Running of privileged containers	privileged
Usage of host namespace	hostPID hostIPC
Usage of host networking and ports	hostNetwork hostPorts
Usage of volume types	Volumes
Usage of the host files ystem	allowedHostPaths
Whitelist of Flexvolume drivers	allowedFlexVolumes
Allocating an FSGroup that owns the pod's volumes	fsGroup
Requiring the use of a read only root file system	readOnlyRootFilesystem
The user and group IDs of the container	runAsUser supplementalGroups
Restricting escalation to root privileges	allowPrivilegeEscalationdefault AllowPrivilegeEscalation
Linux capabilities	defaultAddCapabilities requiredDropCapabilities allowedCapabilities
The SELinux context of the container	SELinux
The AppArmor profile used by containers	annotations
The seccomp profile used by containers	annotations
The sysctl profile used by containers	annotations

4.7.1.2 Cluster Security

（1）Controlling access to the Kubernetes API

The API server uses the Transport Layer Security (TLS) protocol for all API traffic, so as to ensure the inter-service access security. It also provides such access control mechanisms as access control, access authorization, and admission control.

（2）Controlling access to Kubelet

Kubelets expose HTTPS endpoints which grant powerful control over the node and containers. By default, Kubelets allow unauthenticated access to this API. Production clusters should enable Kubelet authentication and authorization.

（3）Controlling the capabilities of a workload or user at runtime

- Limiting resource usage on a cluster

Resource quota limits the number or capacity of resources granted to a namespace. This is most often used to limit the amount of CPU, memory, or persistent disk a namespace can allocate, but can also control how many pods, services, or volumes exist in each namespace.

- Preventing containers running with root privileges

A pod definition contains a security context that allows it to request access to running as a specific Linux user on a node (like root), access to run privileged or access the host network, and other controls that would otherwise allow it to run unfettered on a hosting node. The pod security policy of Kubernetes can limit which users or service accounts can provide dangerous security context settings, for example, volume mounts.

- Restricting network access

The network policies for a namespace allow application authors to restrict which pods in other namespaces may access pods and ports within their namespaces. Quota and limit ranges can also be used to control whether users may request node ports or load balanced services. Additional protections may be available that control network rules on a per plug-in or per environment basis, such as per-node firewalls and physically separating cluster nodes.

- Controlling access to the pod

By default, there are no restrictions on which nodes pods may access. However, Kubernetes provides users with various policies for controlling the location of nodes where the pod runs, including NodeSelector, affinity, and anti-affinity policies.

（4）Protecting cluster components

- Restricting access to etcd

For APIs, write access to the etcd backend for the API is equivalent to gaining root on the entire cluster. Administrators should always use strong credentials from the API servers to their etcd server, such as mutual

authentication via TLS client certificates. A recommended practice is to isolate the etcd servers behind a firewall that only the API servers may access.

- Enabling audit logging

The audit logger is a feature that records actions taken by the API for later analysis in the event of a compromise.

- Restricting access to alpha or beta features

Alpha and beta Kubernetes features are in active development and may have limitations or bugs that result in security vulnerabilities. Therefore, users are advised to disable features they do not use.

- Receiving alerts for security updates and reporting vulnerabilities

Those who join the kubernetes-announce[62] group can receive emails about security announcements. See the security reporting page[63] for more on how to report vulnerabilities.

- Encrypting etcd data

In general, the etcd database contains any information accessible via the Kubernetes API and may grant an attacker significant visibility into the state of clusters. Therefore, users are advised to always encrypt their backups using a well reviewed backup and encryption solution, and consider using full disk encryption where possible.

Kubernetes 1.7 contains encryption at rest, an alpha feature that will encrypt Secret resources in etcd, preventing parties that gain access to users' etcd backups from viewing the content of those secrets. While this feature is currently experimental, it may offer an additional level of defense at critical times.

- Reviewing third-party integrations before enabling them

Many third-party integrations to Kubernetes may alter the security profile of clusters. Therefore, when enabling an integration, users should always review the permissions that an extension requests before granting it access. The pod security policy of Kubernetes helps improve the security.

- Rotating infrastructure credentials frequently

The shorter the lifetime of a secret or credential is, the harder it is for an attacker to make use of that credential. Setting short lifetimes on certificates and automating their rotation is a good way for security protection. Therefore, set short lifetimes on certificates for authentication.

4.7.1.3 Summary

Attention should be paid to the following risks during the use of the orchestration tool.

（1）Whether the authentication and authorization mechanisms are standards-compliant

An independent privilege control policy can be provided for each access interface of the orchestration tool. For example, for the Kubernetes' access interface, API Server, since the communication between Kubernets components is all API-driven, restricting users to access the cluster and controlling operations allowed for users are the first line of defense. Much the same thing is found in Docker CLI.

（2）Whether the orchestration tool fully validates user inputs

A regular expression rule can be set for filtering user inputs, so as to block illegitimate content.

（3）Whether the orchestration tool thoroughly processes anomalies at runtime

The process of anomaly processing at runtime is essential for ensuring stable running of the orchestration tool. Currently, logs of components within the Kubernetes cluster are available for query in the case of an anomaly at runtime. If an application runs improperly in the cluster, troubleshooting should be conducted first from the perspective of the cluster and then the application itself.

（4）Whether the running environment is secure

The security of the external running environment is a prerequisite to that of the orchestration tool. The tools for protecting the security of the host system can be used to enhance the security of the external running environment. In addition, the container host should be hardened to mitigate host security configuration errors. The container engine should be updated to the latest version, so as to avoid damage caused to the host by vulnerabilities existing in old versions. Last but not the least, importance should also be attached to some privilege and audit issues.

4.8 Application Security

The ecosystem of the container technology is gradually established and various solutions become available in specific segments of containers, both of which lay a solid foundation for the container deployment. On the basis of the enterprise container deployment, the emergence of business processes revolving around container applications, especially application logic-oriented microservice architecture, brings challenges to the application security.

4.8.1 Microservice Security

The preceding sections describe the basic concept and framework of microservices. Common components of the container microservice architecture mainly consist of the API server and container-based microservice applications.

Microservice security protection includes the following aspects:

- The authentication and authorization mechanism of the API server
- The security mechanism for guaranteeing the integrity and confidentiality of the communication between microservices in the architecture
- Interface specification of microservices
- High availability (HA) of microservices

Details about security protection of microservices are as follows.

（1）Authentication and authorization

With the conversion from the monolithic architecture to the microservice architecture, various microservices are exposed via REST APIs, which brings privilege control-related security issues to the system, mainly the callers' authentication and API-level operation privilege control issues. These two issues are in a particular order. The system first authenticates the user identity and then checks whether this user has the operation privilege corresponding to his/her specific privilege for the requested operation.

- Authentication

Currently, there are two common authentication schemes: session-based authentication and OAuth2 protocol token-based authentication. The latter is more applicable to microservice scenarios, because it has such advantages as stateless server side, high performance, support for mobile devices, and seamless integration with Spring Security. For different scenarios, OAuth2 provides different grant types: authorization code, implicit, password credential, and client credential, with the latter two as the common grant types.

- Operation privilege control

Currently, there are mainly two protection solutions. One is Apache Shiro[64], which is a powerful, flexible and open-source security framework that performs authentication, authorization, session management, and cryptography and is widely recommended due to its ease-of-use and seamless integration with Spring.

The other solution is Spring Security[65]. Since it has a powerful community, supports OAuth and OpenId, and covers all functions of Shiro, Spring Security is strongly recommended by developers. Its disadvantage lies in its steep learning curve.

（2）Secure communication

In microservices, there are a great number of application services, which contain internal or external data transmission. Considering efficiency, many REST APIs adopt unreliable HTTP-based communication mode, bringing such risks as information disclosure, information theft, and information forgery. For a large-scale microservice architecture, the secure and reliable SSL/TLS communication protocol can be used to encrypt data transmission via user-oriented API pipes, in a bid to provide security assurance.

（3）Security specification of microservice interfaces

In the microservice architecture, services are provided via remote procedure calls (RPCs) or REST APIs/web service interfaces, with HTTP/HTTPS-based REST API as the mainstream. Developing security interface specifications and designing APIs for ensuring the security of microservices are necessary measures to guarantee the security of system applications. APIs can be designed according to the following guidelines:

- Checking the validity of request packets sent to the API gateway
- Filtering requests sent to the API gateway according to the API whitelist policy
- Checking the validity of the format and content of the data sent to the API gateway
- Monitoring operations contained in requests, disrupting sensitive operations, and initiating the authorization

- Limiting the number of access requests to the API gateway from the same user

(4) HA of microservices

Many services in the microservice architecture are mutually dependent. Since each service is likely to get offline and the number of microservices increases, the failure rate increases accordingly. Besides, network timeout and congestion happen from time to time, which can cause service unavailability.

We recommend to improve the availability of microservices from the following perspectives:

- Setting the timeout for service calls, so as to avoid resources being used by abnormal service call threads
- Building independent thread pools for different service traffic, so that resource consumption of one type of abnormal traffic will not affect other legitimate traffic
- Limiting the number of concurrent threads and the call rate of each service interface, guaranteeing the availability of core services
- Configuring a breaker to stop abnormal services timely
- Improving the callback processing for call failures between microservices by lowering service-specific privileges
- Monitoring the availability of microservices and setting a procedure for handling abnormal microservices

4.8.2 DevOps Security

As the methodology in the field of software lifecycle management, DevOps is adopted by more and more IT enterprises. Its value is undoubted. As the security issues facing DevOps have been described in preceding sections, this section details how to ensure the DevOps security.

4.8.2.1 CI/CD Security Suggestions

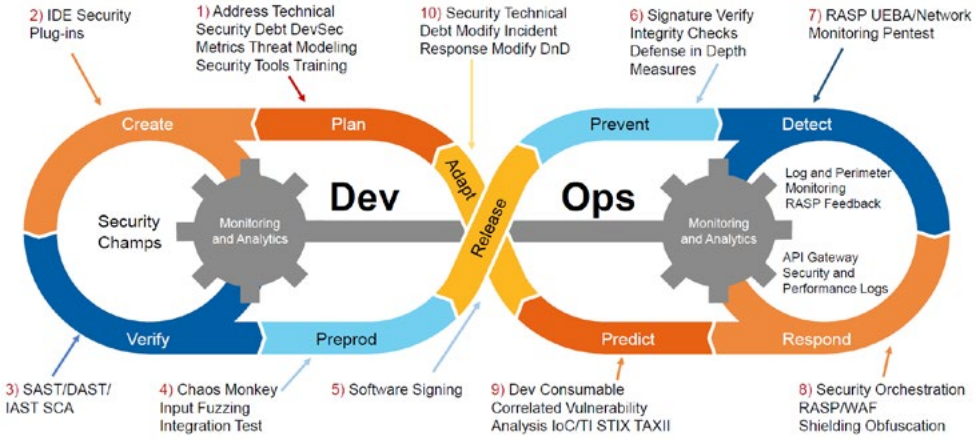
The development and O&M teams adopts an automatic process of continuous integration and continuous deployment (CI/CD) to improve cooperation efficiency and accelerate product development with faster iteration flows. Security suggestions for CI/CI are as follows:

1. Execute code in an isolated and controlled Docker container, protecting the CI server from being threatened.
2. Back up code and programs in the process of CI, avoiding unnecessary debugging operations due to an environment error.
3. If login authentication is involved in the CI/CD process, brute-force password cracking possibly occurs. Therefore, the maximum number of failed login attempts, SMS verification code, or third-party authentication integration should be configured for authentication.
4. During the process of CI/CD, manual authorization should be performed for critical test scenarios and deployment scenarios.

4.8.2.2 DevSecOps

Speaking of DevOps, we have to mention DevSecOps[66]. The former is an abbreviation of "development" and "operations", and the latter is an abbreviation of "development", "security" and "operations". As its name indicates, IT security must be included in the entire process of development and operations. The core is that the responsibilities of all IT members (including development, O&M, and security teams) should be seen in the entire lifecycle (from development to O&M).

Figure 4.8 DevSecOps



The key of the DevOps security lies in the risk control and management, so attention should be paid to the following aspects during the Security and Risk Management (SRM):

- Seamlessly integrate security and compliance testing into DevOps, so that developers do not have to separate from the CI/CD environment.
- Scan all open-source or third-party components for known vulnerabilities and improper configurations. Ideally, you are advised to make a complete asset list for the analysis of all software components.
- Do not try to remove all unknown vulnerabilities in custom code, which may increase false positives. Developers should be those who are the most rigorous and confident.
- Encourage use of new types of tools and methods, so as to reduce the friction between developers. For example, interactive application security testing (IAST) can take place of the traditional static and dynamic testing.
- Use the security champion model to involve the information security team in the DevOps process.
- Use a unified specification for processing all automated scripts, templates, images, and designs, making sure that all source code is covered.

According to Gartner[67], by 2019, more than 70% of enterprise DevSecOps initiatives will have incorporated automated security vulnerability and configuration scanning for open-source components and commercial packages, up from less than 10% in 2016. By 2021, DevSecOps practices will be embedded in 80% of rapid

development teams, up from 15% in 2017. Though, in most cases, DevOps does not properly handle security and compliance issues, we are happy to see that enterprises are taking it more and more seriously.

DevSecOps implementation suggestions are as follows:

（1）Use security test tools and processes catering for developers.

Do not force DevOps developers to adapt themselves to traditional security processes. Instead, security should be integrated to developers' tools and processes. For example, security test tools can be integrated into developers' development environment (IDE) and CI/CD tool chains; security and compliance scanning platforms can be built via APIs, making all security tools and services automatic; threat modeling tools can be included as part of non-functional requirements.

（2）Do not try to eliminate all vulnerabilities during the development.

There is no absolute security. For DevSecOps, what you should do is make continuous risk and trust assessment and prioritize application vulnerabilities. In the implementation of DevSecOps, information security personnel must admit that it is impossible to have no vulnerability, especially when such pursuit obviously hinders the pace of new service development and innovation.

Instead, the runtime protection mechanism can be used to remediate known and low-risk vulnerabilities or unknown vulnerabilities. For example, in the runtime environment, a network- and host-based intrusion protection system (IPS) and web application firewall (WAF) can be used together to monitor the application performance and security and provide in-depth protection at the application layer, including load balancing, denial-of-service (DoS) protection, and distributed denial-of-service (DDoS) protection.

（3）Identify and remove known critical vulnerabilities.

Generally, modern software system are assembled but not developed. Developers can collect and integrate a number of pre-compiled components, code libraries, containers, and frameworks from open-source source code and code repositories (such as GitHub and Docker Hub). Therefore, the wide usage of open-source software poses severe challenges on the security and the focus of security scanning also shifts.

From the perspective of security, identifying known vulnerabilities in known code is much easier than identifying unknown vulnerabilities in custom code. Among various implementation methods, the simplest way is to match files with the vulnerability base. Security vendors should adjust their scanning capabilities by integrating such operations into DevSecOps for an automatic implementation. Docker, as security vendor, also provide such integration capability. The importance of best practice should not be underestimated.

（4）Enhance developers' common sense of basic secure coding

Each member of the DevOps team, especially developers, should receive security training, in a bid to prevent security risks at the very beginning rather than detect and handle such risks after events.

For example, the software system architecture should be securely designed. Input data should be detected, preventing SQL injection and cross-site scripting (XSS) attacks. Databases, configuration files, and network traffic should be detected.

（5）Prohibit the use of known vulnerable components in the source code.

In the real development environment, code repositories downloaded by developers may contain old and known vulnerable software versions. To address this issue, some vendors provide tools such as an open-source firewall to disclose the security trend of code repositories to developers, so as to help them decide which software version to use. In this way, developers can have a clear understanding of which components and code repositories have severe vulnerabilities and will not download and use them.

Organizations that do not allow developers to directly user open-source software should consider using a secure code storage model. The information security team should cooperate with the development team to build, audit, and maintain component storage libraries within their organization.

Besides, some institutions ask developers to use qualified, approved, and standardized code repositories or components (such as authentication, authorization, key management, encryption, clickjacking protection, and input filtering), with the purpose of further reducing risks.

In a word, DevSecOps is such a concept that emphasizes the integration of DevOps and security, which brings a challenge to traditional software development modes and security management modes but impressive security gains at the same time.

2018 NSFOCUS
Technical Report
on Container Security

5. Security Tools

5.1 Open-Source Security Tool Kubernetes 91
5.2 NeuVector 101
5.3 StackRox 104

5.1 Open-Source Security Tool Kubernetes

In addition to commercial software, open-source software projects can also provide some security functions. This document describes several open-source projects that are usually used for protection of non-critical business.

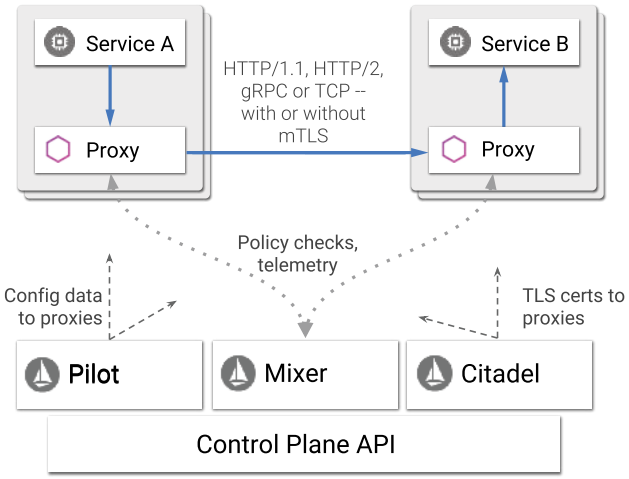
5.1.1 Kubernetes's Native Security Policies

Kubernetes's network policies provide L3/L4-level (IP address/port) network isolation. Kubernetes does not provide network functions, but opens network interfaces and uses network plug-ins (such as Flannel[25], Calico[26], Contiv[68], Cilium[69], and Kube-router[70]) to support the execution of network policies.

5.1.2 Istio

Istio is an open-source framework jointly developed by Google, IBM, and Lyft to manage, secure, and monitor microservices. It creates a service mesh to manage service-to-service communication, including routing, authentication, authorization, and encryption. The following figure shows the Istio architecture.

Figure 5.1 Istio architecture



In a word, Istio can be defined as a simple way to build a network of services deployed. It provides load balancing, service-to-service authentication, monitoring, and the like, without requiring any changes in service code.

As shown in the preceding figure, Istio consists of a control plane and a data plane.

（1）Data plane

- Envoy[71]: As a high-performance proxy written in C++, Envoy mediates inbound and outbound traffic for all services within the service mesh.

（2）Control plane

- Mixer: Mixer executes access control and usage policies across the service mesh and collects telemetry data from Envoy and other services.
- Pilot: collects and verifies configuration policies and spreads them to Istio components. Independent of the underlying platform, it extracts environment-specific implementation details from Mixer and Envoy and provides abstract representations of user services for them.
- Citadel: Citadel provides strong service-to-service and end-user authentication with interactive TLS, built-in identity, and credential management.

Meanwhile, Istio provides a security mechanism which, without any modification to service code, secure services by using a sidecar container deployed near the container to which the microserivce belongs. This mechanism provides authentication and authorization functions.

（1）Authentication

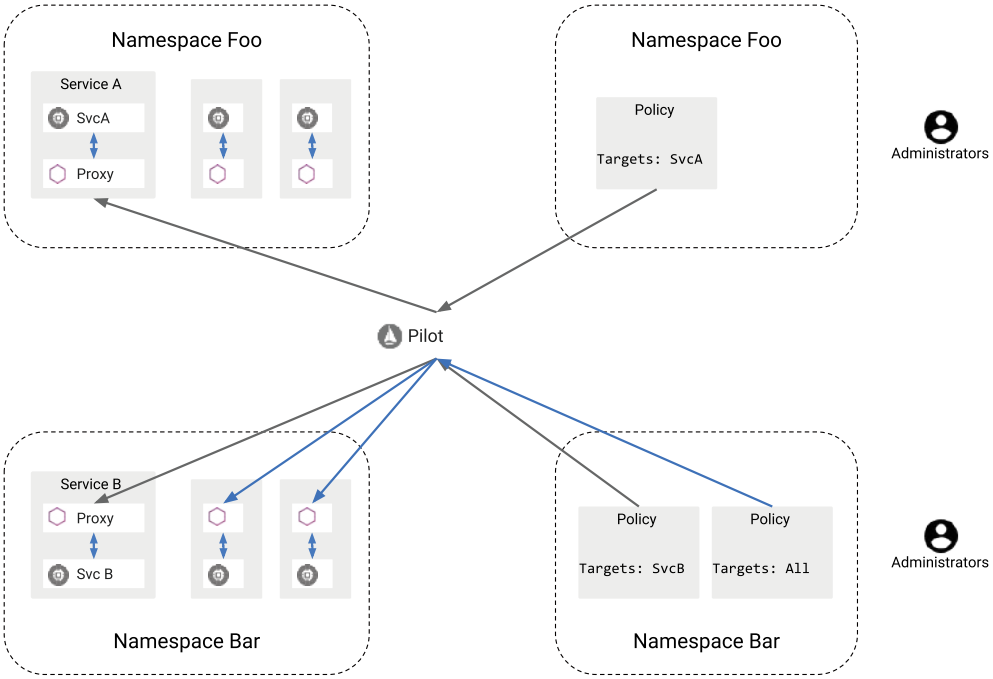
Istio provides two types of authentication:

- Transmission authentication (service-to-service authentication): Istio provides bidirectional TLS authentication as a complete authentication solution.
- Source authentication (authentication of end users): Istio authenticates the original client requesting to be an end user or device. With JSON Web Token (JWT), Auth0, Firebase Auth, Google Auth, and custom authentication methods, Istio simplifies the developer experience and easily implements request authentication.

In the preceding circumstances, Istio stores authentication policies in the Istio Config Store through a custom Kubernetes API. Pilot, when appropriate, will keep the latest status and key for each proxy. Besides, Istio supports authentication in license mode to help users learn about what impact policy changes will have on the user's security status before taking effect.

With authentication policies, mesh operators can specify authentication requirements for services that receive requests in the Istio mesh. Figure 5.2 shows Istio's authentication architecture. Administrators set authentication policies using the yaml configuration file and deploy them on Namespace Foo and Namespace Bar. After deployment, these policies will be stored in Istio Config Store which is monitored by Pilot. Once changes occur to policies, Pilot will convert new policies into appropriate configurations and notify the Envoy Sidecar proxy, which is at the same position as the service instance, of how to execute the required authentication mechanism. For example, the policy defined in Namespace Foo targets SvcA and therefore will work for SvcA's Envoy Sidecar proxy. Likewise, two policies created in Namespace Bar respectively target ALL and SvcB and therefore work for proxies of the two services.

Figure 5.2 Authentication architecture of Istio



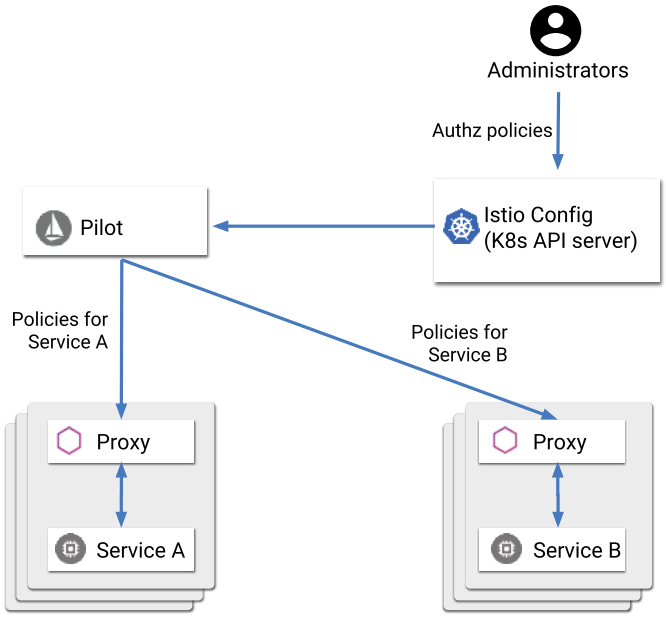
(2) Authorization

Istio's authorization feature is also known as role-based access control (RBAC) which provides namespace-level, service-level, and method-level access control for services in an Istio mesh. This function has the following characteristics:

- Role-based semantics, which makes for ease of use.
- Service-to-service and end user-to-service authorization.
- High flexibility as it allows users to define attributes such as bindings between conditions and roles.
- High performance as Istio authorization is enforced natively on Envoy.

Figure 5.3 shows the official Istio authorization architecture. The network administrator can set the Istio authorization policy using the yaml configuration file and deploy it. After deployment, this policy is stored in Istio Config Store and under the monitoring of Pilot. Once finding any changes are made to the policy, Pilot will re-obtain the policy and dispatch it to the Envoy Sidecar proxy which is at the same position as the service instance. Each Envoy Sidecar proxy will run an authorization engine which authorizes requests at runtime. When a request arrives at the proxy, its running authorization engine will evaluate the request context against the authorization policy and return the authorization result (ALLOW or Deny).

Figure 5.3 Istio authorization architecture



5.1.3 Grafeas

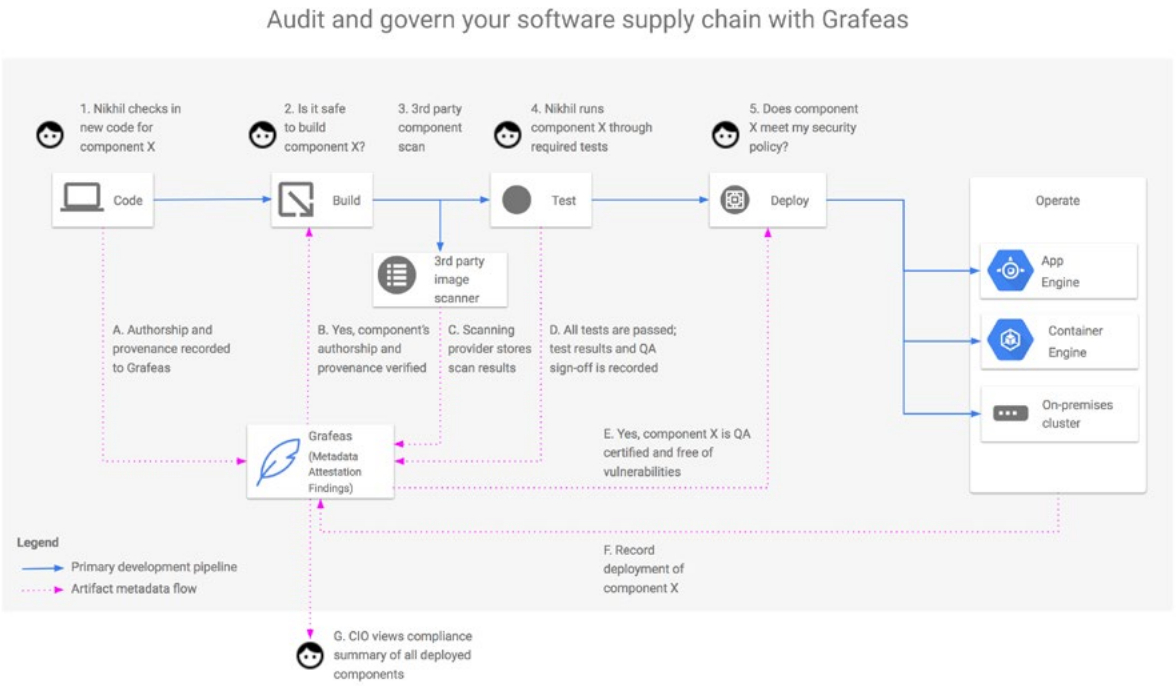
Grafeas[72] is a tool launched in 2017 by Google in collaboration with JFrog and IBM among others for auditing and governing the software supply chain.

With DevOps and microservices constantly develop, software is delivered at an increasingly rapid pace, with delivered binaries (including Docker images) showing exponential increase. Multiple languages for software development differ materially in data specifications. For example, the GO language uses URLs to define its own component addresses; Java has its unique specifications; Docker identifies each layer with a Sha256 number. Such specification differences make it difficult to determine the quality of and vulnerabilities in binaries of multiple languages included in container images before these images are delivered for production.

Grafeas was originally designed for container security quality management. The official definition, however, suggests that Grafeas goes beyond its original purpose to provide an open-source component metadata API for auditing and monitoring software components in a centralized way. Software mentioned here can be container images or packages in WAR, JAR or ZIP format.

Figure 5.4 shows a software supply chain (code, build, test, deploy, and operate) of multi-side collaboration in the container environment before images are delivered for production. In each stage, Grafeas collects key metadata via its API. Especially in the deployment stage, Grafeas can conduct effective checks on the compliance with final security specifications.

Figure 5.4 Flowchart of auditing and governing a software supply chain with Grafeas

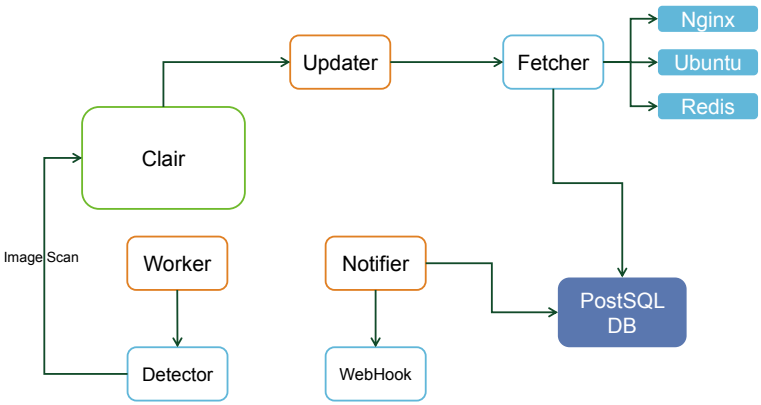


Security policies provided by Grafeas can be enforced through integration with third-party tools. Such policies can be used for security management of CI/CD pipelines rather than containers at runtime.

5.1.4 Clair

Clair is an open-source container analysis tool released by CoreOS in November 2015 to fix security vulnerabilities in container images. Clair makes a static analysis of container images, associating image contents with public vulnerability databases and finally forming analysis reports.

Figure 5.5 Clair architecture

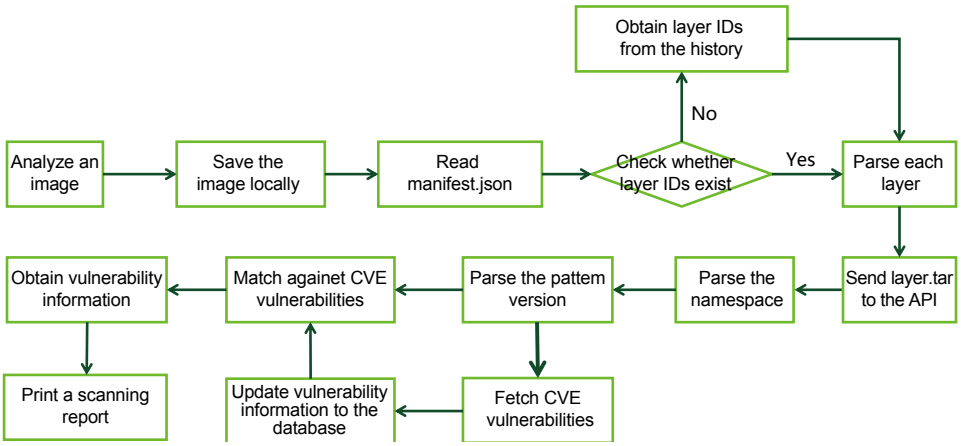


Clair consists of a detector, fetcher, notifier, and PostSQL database among other modules. For image detection, Clair first extracts characteristics of an image and then matches them against the Common Vulnerabilities and Exposures (CVE). The detailed process is as follows:

- For an image to be detected, the user invokes a RESTful API of Clair, which will upload all layers of the image and, for each layer, create a worker.
- Each worker starts a detector to scan the received layer.
- The detector first decompresses the compressed layer package that is uploaded to obtain the operating system type and version as well as the name and version of the installed application.
- The updater starts a fetcher which will search the CVE database for vulnerability information of the obtained operating system type to check the existence of vulnerabilities in the current operating system and applications.

To handle vulnerabilities, Clair will start a Notifier+WebHook asynchronous process in which WebHook keeps crawling up-to-date vulnerability information from CVE websites, then notifies newly exposed CVE vulnerabilities to the notifier, and finally updates the vulnerability information to the database.

Figure 5.6 Image scanning process



Users can execute simple command lines with Clairctl[73], the lightweight Clair client tool, to upload and scan images as well as output reports.

5.1.5 CIS Benchmarks

The Center for Internet Security (CIS)[74] is a nonprofit entity that harnesses the power of a global IT community to safeguard private and public organizations against cyber threats. CIS Benchmarks is a recognized global standard and best practice for securing IT systems and data against attacks. CIS Benchmarks provide a wide variety of security assessment standards covering operating systems, system software, cloud providers, and mobile devices among others.

5.1.5.1 Docker CIS Benchmark

Docker and CIS co-produce a security benchmark document on security audit of container environments. Five versions[1] are available for this document:

- CIS v1.0.0 for Docker 1.11.0 Benchmark;
- CIS v1.0.0 for Docker 1.12.0 Benchmark;
- CIS v1.0.0 for Docker 1.13.0 Benchmark;
- CIS v1.0.0 for Docker 1.6 Benchmark;
- CIS v1.1.0 for Docker Community Edition Benchmark

For example, *CIS V1.0.0 for Docker* 1.13.0 Benchmark, released on January 19, 2017, presents more than 100 security configuration and audit methods covering host configuration, Docker configuration, Docker daemon configuration files, container image and Build file, container runtime, and Docker security operations among others.

The Docker community open-sourced this tool[75] which supports *CIS Docker Community Edition Benchmark V1.1.0.*

Figure 5.7 Docker Benchmark Security

Docker Hub provides the image of this Benchmark which needs to run on Docker 1.13.0 or later. Users can use the following command to rapidly deploy this program:

1 Data as of July 31, 2018

```
# docker run -it --net host --pid host --userns host --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /etc:/etc --label docker_bench_security \
    docker/docker-bench-security
```

5.1.5.2 Kubernetes CIS Benchmark

Meanwhile, CIS provides security Kubernetes benchmark documents to audit the operating environment of Kubernetes. Currently, two versions are available for such documents: *CIS v1.0.0 for Kubernetes 1.6 Benchmark and CIS v1.2.0 for Kubernetes 1.8 Benchmark*[1]. The latter, CIS V1.2.0, is the latest version which supports Kubernetes 1.8.

CIS v1.2.0 for Kubernetes 1.8 Benchmark, released in October 2017, provides more than 100 security audit check items covering API server configuration, Scheduler configuration, controller configuration, and Etcd among others.

Many organizations have open-sourced Kubernetes's compliance and audit check tools such as NeuVector's NeuVector CIS Kubernetes Benchmark[76] NeuVector kubernetes-cis-benchmark and Aqua's kube-bench[77] (both tools support Kubernetes 1.6 and 1.8).

Figure 5.8 NeuVector CIS Kubernetes Benchmark

```
# ------------------------------------------------------------------------
# Kubenetes CIS benchmark 1.6
#
# NeuVector, Inc. (c) 2016-
#
# ------------------------------------------------------------------------

[WARN] 1.1.1  - Ensure that the --allow-privileged argument is set to false
[WARN] 1.1.2  - Ensure that the --anonymous-auth argument is set to false
[PASS] 1.1.3  - Ensure that the --basic-auth-file argument is not set
[PASS] 1.1.4  - Ensure that the --insecure-allow-any-token argument is not set
[PASS] 1.1.5  - Ensure that the --kubelet-https argument is set to true
[PASS] 1.1.6  - Ensure that the --insecure-bind-address argument is not set
[PASS] 1.1.7  - Ensure that the --insecure-port argument is set to 0
[PASS] 1.1.8  - Ensure that the --secure-port argument is not set to 0
[WARN] 1.1.9  - Ensure that the --profiling argument is set to false
[WARN] 1.1.10  - Ensure that the --repair-malformed-updates argument is set to false
[PASS] 1.1.11  - Ensure that the admission control policy is not set to AlwaysAdmit
[WARN] 1.1.12  - Ensure that the admission control policy is set to AlwaysPullImages
[WARN] 1.1.13  - Ensure that the admission control policy is set to DenyEscalatingExec
[WARN] 1.1.14  - Ensure that the admission control policy is set to SecurityContextDeny
[PASS] 1.1.15  - Ensure that the admission control policy is set to NamespaceLifecycle
[WARN] 1.1.16  - Ensure that the --audit-log-path argument is set as appropriate
[WARN] 1.1.17  - Ensure that the --audit-log-maxage argument is set to 30 or as appropriate
[WARN] 1.1.18  - Ensure that the --audit-log-maxbackup argument is set to 10 or as appropriate
[WARN] 1.1.19  - Ensure that the --audit-log-maxsize argument is set to 100 or as appropriate
[PASS] 1.1.20  - Ensure that the --authorization-mode argument is not set to AlwaysAllow
[PASS] 1.1.21  - Ensure that the --token-auth-file parameter is not set
[WARN] 1.1.22  - Ensure that the --kubelet-certificate-authority argument is set as appropriate
[PASS] 1.1.23  - Ensure that the --kubelet-client-certificate and --kubelet-client-key arguments are set as appropriate
[PASS]        * kubelet-client-certificate: /etc/kubernetes/pki/apiserver-kubelet-client.crt
[PASS]        * kubelet-client-key: /etc/kubernetes/pki/apiserver-kubelet-client.key
[WARN] 1.1.24  - Ensure that the --service-account-lookup argument is set to true
[WARN] 1.1.25  - Ensure that the admission control policy is set to PodSecurityPolicy
```

1 Data as of July 31, 2018

5.1.6 TUF and Notary

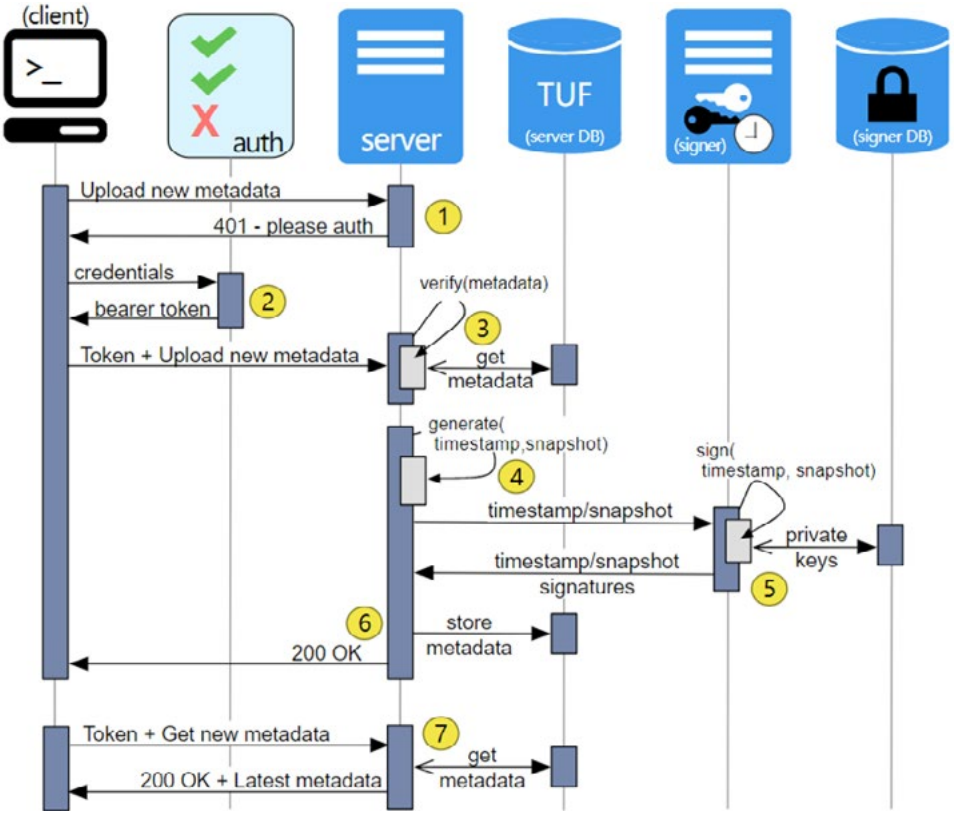
The Update Framework (TUF)[78] and Notary[79] are two security incubation projects that are accepted in to the Cloud Native Computing Foundation (CNCF)[80]. TUF is an open-source security and verification standard, while Notary is an implementation of TUF. The Update Framework was originally developed by New York University professor Justin Cappos and his team at Tandon School of Engineering. Notary was originally an image signing system component for Docker1.8.

Notary adopts a server/client architecture in which the server end runs and maintains the trusted collection of a container, while the client end interacts with the server. In this way, Notary implements more secure content delivery and verification in a simple way.

Usually, we can use the Transport Layer Security (TLS) protocol to secure the communication of both ends. This method, however, is less than perfect as any compromises to the server end will lead valid contents to be replaced by malicious ones.

With the aid of Notary, container image publishers could sign container images with strongly secure keys for encryption. In the wake of encryption, images upon publication are first pushed to the Notary's trusted collection. In this case, a visitor can set up communication with the server in Notary and access the appropriate image as long as he or she obtains the public key of the publisher through a security channel.

Figure 5.9 Notary workflow



5.1.7 SPIFFE

The Secure Production Identity Framework for Everyone (SPIFFE)[81] is an open and community-driven service security framework actively promoted by Google. It is used for identifying and securing communications between web-based services.

Currently, most service implementations are bound to IP addresses. An IP address, however, may run a number of services. For example, an IP address runs more than one container and each container has its own task. The problem is that how we can know which container is legal or illegal. Here is another example: Service A serves service B. How can we know service A truly serves service B rather than being used for other malicious purposes?

SPIFFE defines how services identify themselves to each other. Actually, this framework uses an X.509 certificate to verify the identity of each object in the production environment. Such an X.509 certificate contains a SPIFFE ID which identifies an object and takes the form of Uniform Resource Identifiers (URIs).

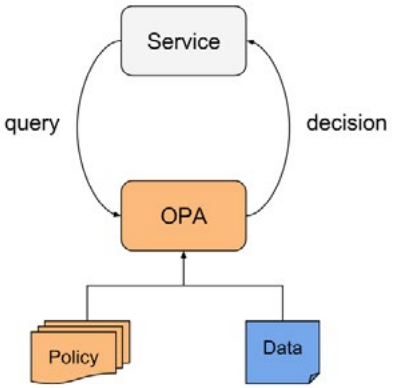
SPIRE[82](short for SPIFFE Runtime Environment) is an open-source implementation of SPIFFE. As a tool-chain, SPIRE is used to establish trust between software systems across a wide variety of hosting platforms. Specifically, SPIRE exposes the SPIFFE Workload API, which can prove the identities of running software systems before issuing SPIFFE IDs and SVIDs to them. This in turn allows two workloads to establish trust between each other, for example by establishing an mTLS connection or by signing and verifying a JWT token.

SPIFFE is hosted by CNCF as a sandbox-level project.

5.1.8 Open Policy Agent (OPA)

Open Policy Agent (OPA) [83] is an open-source general-purpose policy engine which specifies how to enforce policies in a cloud native environment. OPA enables unified and context-aware policy enforcement. Such engine, when integrated with a service, allows users to establish and deploy custom policies to control the policy-based function of this service, as shown in the following figure. OPA is hosted by CNCF as a sandbox-level project.

Figure 5.10 OPA architecture example



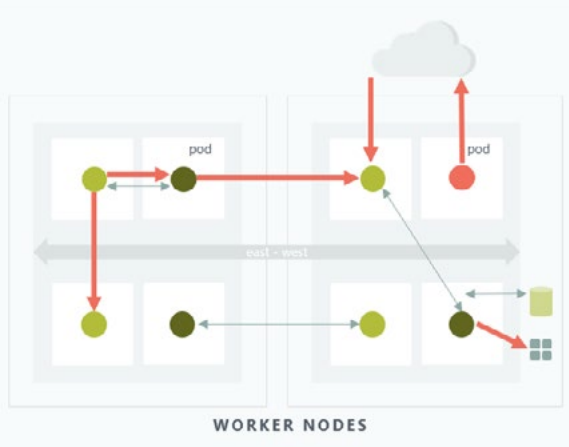
5.2 NeuVector

NeuVector[84] is the first company to take up development of Docker/Kubernetes security products. With a commitment to assuring the security of enterprise-wide container platforms, the company provides products that are suitable for deployment across multi-cloud and on-premises production environments.

5.2.1 Cloud-Native Container Deployment

NeuVector provides in-depth runtime visibility into the container network, monitors "east-west" container traffic, performs proactive isolation and protection, and ensures the security of hosts and within containers. Through seamless integration with container management platforms, it achieves automation of application-level container security.

Figure 5.11 NeuVector deployment topology

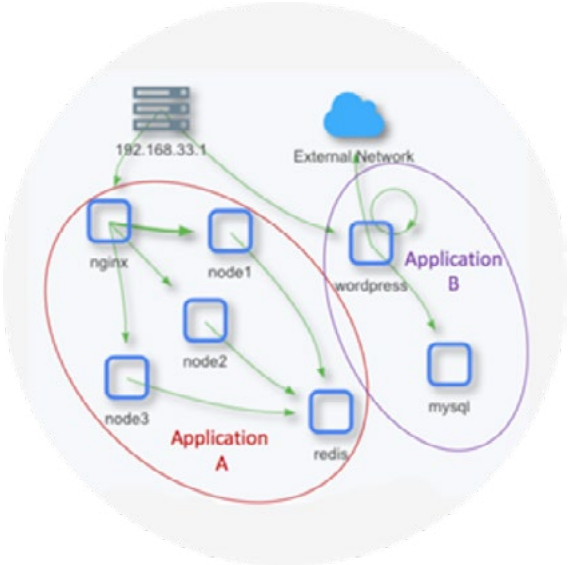


NeuVector is distributed as a cloud-native container, which can be quickly deployed, managed, and upgraded via any standard container management platforms, such as Kubernetes, Docker, OpenShift, Rancher, and Mesosphere. In addition, it supports various public cloud- and private cloud-hosted container service platforms like AWS ECS, EKS, Google GKE, IBM Cloud, Docker Datacenter, Azure, Alibaba Cloud, and VMware PKS.

5.2.2 Container Environment Visibility

NeuVector provides flexible support for various installation environments and can be deployed in advance or at a later time to automatically discover and protect running containers and newly deployed containers in real time. After being deployed, NeuVector will automatically learn all containers' network behaviors and the behaviors within containers. Based on a comprehensive analysis, it will present a real-time container networking diagram. Such powerful container visibility is useful for security monitoring, topology analysis, instant adjustment, event response, container network packet capture, and even application container debugging.

Figure 5.12 NeuVector's container environment visibility



5.2.3 Security Audit

NeuVector supports various security audit functions. It provides a wide range of security audit reports regarding Docker Bench scanning, Kubernetes CIS Benchmark scanning, container vulnerability scanning, and container host scanning results. Audit results can inform proactive security policies, enabling the security system to automatically respond, detect, control, and prevent any security events when containers are deployed in an active or passive manner. NeuVector supports scanning of container image repositories and provides a Jenkins plug-in to help developers test container security risks.

Figure 5.13 NeuVector security audit

Test numb...	Level	Message
3.19	PASS	Verify that /etc/default/docker file owne...
3.20	PASS	Verify that /etc/default/docker file perm...
⊟4	INFO	*Container Images and Build Files*
⊟4.1	WARN	*Create a user for the container*
	WARN	Running as root: wordpress
	WARN	Running as root: mysql
4.2	NOTE	Use trusted base images for containers
4.3	NOTE	Do not install unnecessary packages in ...
4.4	NOTE	Scan and rebuild the images to include ...
4.5	WARN	Enable Content trust for Docker
⊟4.6	WARN	*Add HEALTHCHECK instruction to the co...*

NeuVector can help customers conduct various standards compliance checks and industry compliance checks. Such standards include Payment Card Industry Data Security Standard (PCI DSS), EU General Data Protection Regulation (GDPR), and Health Insurance Portability and Accountability Act of 1996 (HIPPA).

5.2.4 Multidimensional Protection

NeuVector protects containers at runtime from various aspects. Its functions include automatic container isolation at the level of network applications and automatic learning of container security policies according to containers' behaviors to detect malicious behaviors within containers, such as abnormal processes, abnormal file system read/write, and privilege escalation.

NeuVector also comes with a proactive protection function to prevent various prevalent cyberattacks, such as Slowloris DDoS, SQL injection, DNS attacks, SSH heartbleed, and reverse shell. Moreover, NeuVector can monitor container hosts for abnormal behaviors to implement all-round protection of container production environments. With such multivector and multidimensional protection come huge amounts of event data, making it possible to anatomize an event throughout the kill chain.

Figure 5.14 Multidimensional protection of containers at runtime

In terms of security response and management, NeuVector can generate alerts upon detection of suspicious behaviors, block malicious behaviors, isolate containers or services, tear down network sessions, and automatically capture network sessions and packets for security analysis and forensics. NeuVector can integrate with standard security information and event management (SIEM) systems and provide support for wbhooks, syslog, REST APIs, and CLI.

NeuVector on Docker/Kubernetes provides advanced security functionality to ensure containers' security throughout the lifecycle of containers that extends from development, testing, and deployment to O&M, upgrade, and production.

Figure 5.15 NeuVector's security assurance throughout the lifecycle of containers

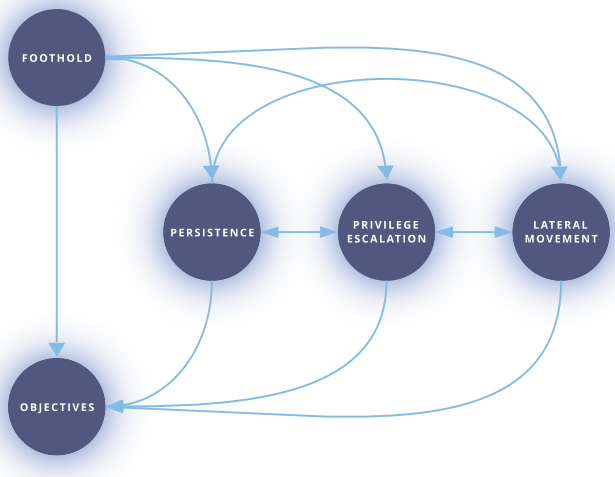
		KUBERNETES/ DOCKER	NEUVECTOR
Build, Infrastructure & Deployment	Registry Scanning, Jenkins Plug-In RBAC, Secrets Management Pod Resource & Security Management	✓ ✓ ✓	✓ RBAC Integration
Network Security	Network Policy – L3/L4 Isolation	✓	✓
	Layer 7 Deep Packet Inspection, Isolation Threat Detection Automated Packet Capture		✓
Endpoint Security	Connection Visualization, Monitoring		✓
	Suspicious Process & File System Activity		✓
Compliance	CIS Benchmarks & Run-Time Scanning		✓
Response	Automated Response – Multi-Vector		✓

5.3 StackRox

StackRox features a distributed architecture that collects and analyzes data throughout the application lifecycle to detect and block malicious actors, and finally meet the requirement for protecting containerized cloud-native applications. StackRox delivers continuous detection through its unique combination of distributed sensors and centralized analysis and machine learning to provide context and correlation at the speed and scale of containers.

The Adversarial Intent Model (AIM) is a product design idea proposed by StackRox. With the rapid development of modern information technology (IT), conventional scanning, patching, prevention, monitoring, and methods for securing independent operations seem to lag behind and be insufficient for effective protection. StackRox came up with an idea of developing defense strategies from the standpoint of adversaries. With the overall operating environment of containers taken into account and based on the five attack lifecycle phases, StackRox proposed a new multidimensional threat detection model, namely AIM.

Figure 5.16 Attack lifecycle model



Currently, the company's products mainly include StackRox Prevent and StackRox Detect and Respond.

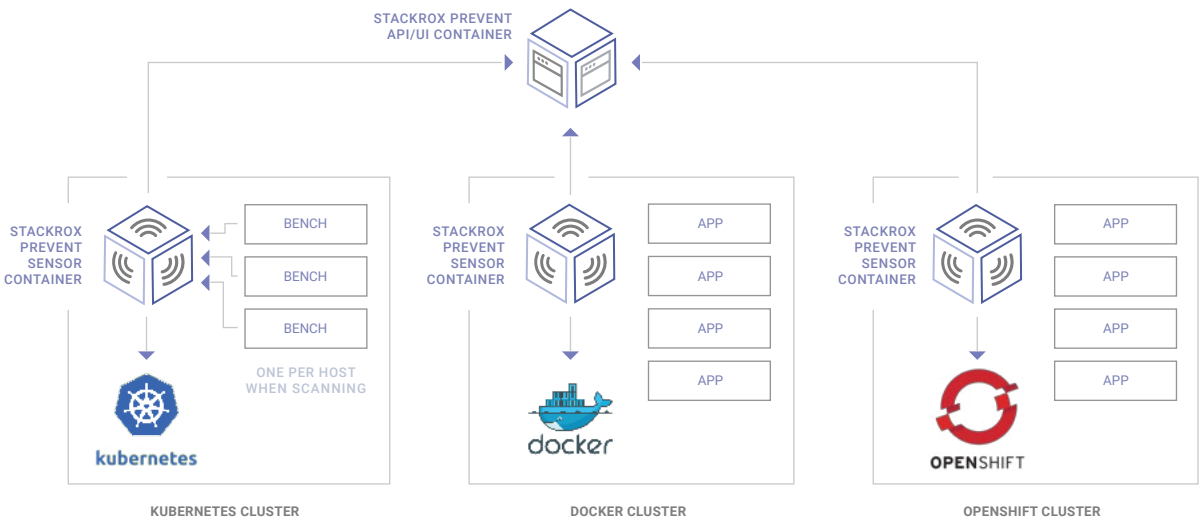
5.3.1 StackRox Prevent

StackRox Prevent is a product that automates security and compliance for container deployments based on monitoring and analysis of information across multiple dimensions. It provides a contextual profile of overall container security posture by checking whether containers have adopted appropriate controls and configurations, thereby minimizing the attack surface from the initial phase.

Core functions of StackRox Prevent are as follows:

- Dashboard: provides a holistic view of the entire container cluster, events, images, security configurations, alerts, and the like.
- Security policy management: creates and modifies security detection policies; reevaluates and updates security policies after disclosure of new vulnerabilities.
- Compliance check: scans Docker, Swarm, and Kubernetes configurations against CIS Benchmarks to determine the gap between current environment configurations and security baselines.
- Product integration: can easily integrate with container-related platforms or tools.

Figure 5.17 Logical view of StackRox Prevent



In the preceding logical view, StackRox Prevent consists of the sensor and API/UI parts, both of which are deployed on a container platform or orchestration platform as containers.

Sensor containers are deployed in various container clusters in a distributed manner to implement detection. The API/UI container is a logical centralized controller, which exchanges control and result information with the distributed sensors through APIs and presents such information on the user interface (UI).

StackRox Prevent has the following characteristics:

- Risk evaluation and forecast based on rich environment data such as image vulnerabilities, container configurations, and host and network configurations
- Fast generation of aggregate analysis reports based on detection results to quantify and benchmark risks
- Providing configuration baselines and supporting security policy customization
- Adaptation to and support for a variety of platforms (DevOps, Orchestration, and PaaS) and tools

Up to now, StackRox Prevent has adapted itself to various platforms and tools, as listed in the following table.

Container Platform	Docker Enterprise Edition, Google Kubernetes Engine (GKE), RedHat OpenShift
IaaS	Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Bluemix, Microsoft Azure, OpenStack, Oracle Cloud, virtual machines (KVM, Hyper-V, VMware, Xen), bare metal
Image Repository	Docker Hub, Docker Trusted Registry (DTR), Tenable, Quay
Vulnerability Scanner	Docker Security Scanning, Tenable, Quay, CoreOS Clair
Identity Management	Auth0
Workflow	Jira, Slack, e-mail

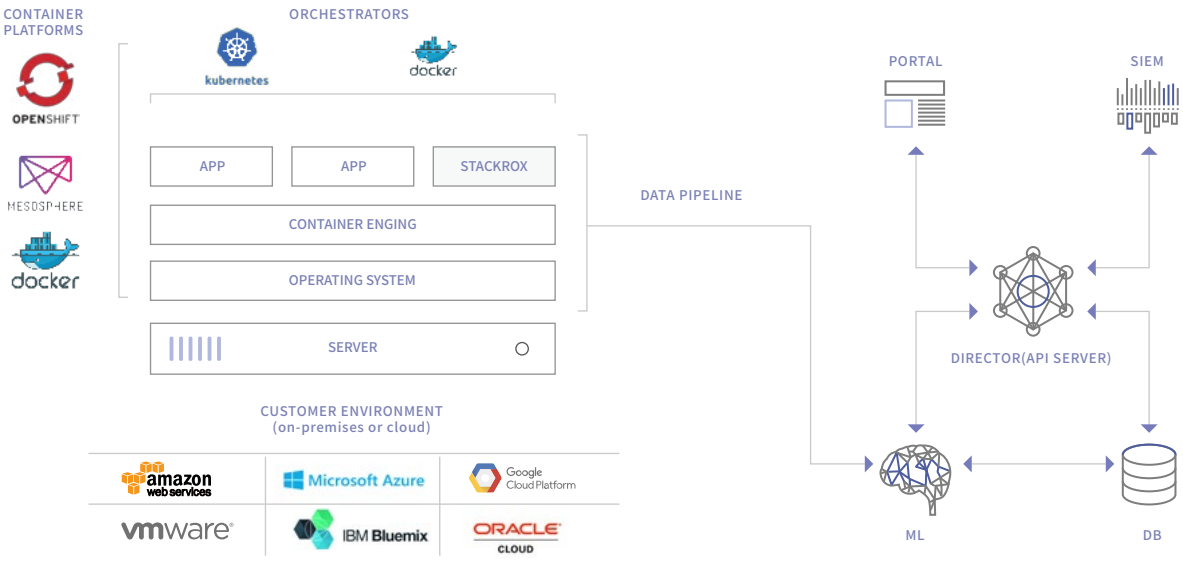
5.3.2 StackRox Detect and Respond

StackRox Detect and Respond, through integration with containers or cloud platforms, provides built-in security for local applications and visibility into the container attack surface, reveals malicious activities, and prevents attacks by employing a new detection and response method.

Core functions of StackRox Detect and Respond are as follows:

- Attack visibility: Through deployed sensors, the product can continuously monitor millions of signals, detect anomalous activities in each container at runtime, and promptly nip the threat in the bud.
- Kill chain analysis: Through machine learning, the product can learn how an attacker gains access, infiltrates the business environment, and finally launches attacks, and conduct an in-depth analysis of the root cause.
- Threat prevention: By automatically blocking, isolating, and narrowing down threats, the product can effectively mitigate the impact of threats.

Figure 5.18 Logical view of StackRox Detect and Respond



StackRox Detect and Respond combines a brand-new security architecture with ongoing monitoring and machine learning capabilities to protect the container environment from new threats. The product can be deployed on any type of infrastructure through container-based microservices.

The logical view in Figure 5.18 shows that the product consists of two parts, the same as StackRox Prevent. In a container cluster, StackRox can be deployed for data collection and runtime monitoring. Director, as the controller for logically centralized management, analyzes the collected data by means of machine learning and makes decisions accordingly on how to handle the perceived threats. However, until now, there is no detailed explanation about how such machine learning is implemented and how accurate that will be.

StackRox Detect and Respond has the following characteristics:

- Deep visibility into threats

  Asset visibility: discovers all containers in a cluster and groups services within applications to provide a holistic view of assets for threat detection.

  Threat visibility: provides low-noise data from ongoing monitoring of file systems, networking, processes, and container-related events to constitute a massive web of threat monitoring points.

  Business visibility: provides valid data sources for anomaly monitoring and identification by monitoring communication between all containers.

- Adaptive detection and response

  Expanded monitoring scope: Currently, the monitoring can be conducted from five dimensions: foothold, privilege escalation, persistence, lateral movement, and objectives.

Autotuned machine learning: implements fully automated learning according to container activity changes.

Automated orchestration and response: automatically responds to discovered threats by blocking unauthorized instructions, terminating system calls, or isolating the related container.

Policy customization: While StackRox provides common preconfigured templates, users can also customize their protection policies according to their own business workflows.

Alert context: provides detailed context for security events to inform decision-making on threat responding.

- Fast deployment and ease of management

StackRox Detect and Respond can be launched using the existing deployment toolchain or directly deployed on the current orchestration system to be subject to unified management and orchestration with other containers. StackRox allows operations through web interfaces, command lines, and APIs. Up to now, StackRox Detect and Respond has adapted itself to various platforms and tools, as listed in the following table.

Container Platform	Amazon Elastic Container Service for Kubernetes (EKS), Azure Container Service (AKS), Docker Enterprise Edition, Google Kubernetes Engine (GKE), IBM Bluemix Container Service, Mesosphere DC/OS, Red Hat OpenShift
Operating System	CentOS, Debian, Red Hat Enterprise Linux (RHEL), Ubuntu
IaaS	Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Bluemix, Microsoft Azure, OpenStack, Oracle Cloud, virtual machines (KVM, Hyper-V, VMware, Xen), bare metal
Image Repository	Amazon EC2 Container Registry (ECR), Artifactory, Azure Container Registry (ACR), Docker Hub, Docker Trusted Registry (DTR), Google Container Registry (GCR)
Identity Management	SAML 2.0-compliant identity providers including Google, Okta, Ping Identity
Event Alert	PagerDuty, Slack

6. Summary

The container technology provides a lightweight mode of virtualization, greatly conveniencing DevOps and development of cloud-native applications. After several years' evolution, it has been accepted as a technology with increasingly strong advantages. Containers were initially popular as a useful tool for developers to isolate the development and testing environment from the ongoing integration environment thanks to their lightweight as well as ease of configuration and use.

In the meantime, cloud-native applications have been gradually accepted by more and more people because of their agility, high scalability, and high availability. As a typical example, the Cloud Native Computing Foundation (CNCF) initiated by Google has drawn wide attention from numerous vendors and open-source communities.

From containerization of applications to development of cloud-native applications, the container technology has always been the most fundamental and essential support. While new technologies can convenience and benefit our work, they also bring in new security threats, which should never be neglected.

In the past few years, a succession of security risks and events incurred by containers and container application environments have been reported. Container networking, container images, exposed APIs, and container isolation have become top concerns for the use of containers.

Revolving around security risks of containers from aspects of software vulnerabilities and security threats along with application security threats, this article presents security issues facing containers and container application in a systematic manner. To deal with these security issues, the author proposes detection and protection recommendations on host security, image security, networking security, and application security. The last part of this article gives a brief account of some solutions to container security from the perspectives of open-source communities and vendors.

7. References

[1] Kata Containers, https://katacontainers.io/

[2] Docker, https://github.com/docker

[3] Rocket, https://github.com/rkt/rkt

[4] Windows Containers, https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/

[5] Alibaba Pouch, https://github.com/alibaba/pouch

[6] Tainted, crypto-mining containers pulled from Docker Hub, https://techcrunch.com/2018/06/15/tainted-crypto-mining-containers-pulled-from-docker-hub/

[7] Containers At-Risk, A Review of 21,000 Cloud Environments, https://info.lacework.com/containers-at-risk-cloud-environments-review?_ga=2.217313366.1971398103.1529458438-615050303.1529458438

[8] CNNVD, http://www.cnnvd.org.cn/web/vulnerability/queryLds.tag?pageno=3&repairLd=

[9] https://docs.docker.com/enterprise/17.06/#17062-ee-6-2017-11-27

[10] https://github.com/moby/moby/pull/35484

[11] https://github.com/moby/moby/pull/35482

[12] https://github.com/moby/moby/pull/35424

[13] Gartner Identifies the Top Technologies for Security in 2017，https://www.gartner.com/newsroom/id/3744917

[14] Gartner Releases the Hype Cycle for Cloud Security in 2017，https://www.gartner.com/newsroom/id/3797963

[15] Docker – Containers and Container Cloud, V2.0

[16] Open Container Initiative, https://github.com/opencontainers/image-spec/releases/tag/v1.0.0

[17] Docker Hub, https://hub.docker.com/

[18] VMware Harbor, https://vmware.github.io/harbor/cn/

[19] Docker images commandline reference, https://docs.docker.com/engine/reference/commandline/images/

[20] Docker storage drivers, https://docs.docker.com/storage/storagedriver/select-storage-driver/

[21] CoreOS Flannel, https://github.com/coreos/flannel

[22] Calico, https://github.com/projectcalico

[23] Kubernetes, https://kubernetes.io/

[24] Apache Mesos, http://mesos.apache.org/

[25] Docker Swarm, https://github.com/docker/swarm

[26] Docker Compose, https://github.com/docker/compose

[27] Rancher, https://rancher.com/

[28] Docker Machine, https://github.com/docker/machine

[29] A Survey on the Development of Open-Source Cloud Computing Technologies in China (2018), http://www.caict.ac.cn/kxyj/qwfb/ztbg/201804/t20180426_158539.htm

[30] Google Kubernetes Engine https://cloud.google.com/kubernetes-engine/

[31] Azure Kubernetes Service https://azure.microsoft.com/en-us/services/container-service/

[32] Azure Container Instances https://azure.microsoft.com/en-us/services/container-instances/

[33] Amazon Elastic Container Service https://amazonaws-china.com/cn/ecs/

[34] Amazon Elastic Container Service for Kubernetes https://amazonaws-china.com/cn/eks/?nc2=h_m1

[35] 2017 State of the Cloud Report, https://www.rightscale.com/lp/2017-state-of-the-cloud-report

[36] Linkerd, https://linkerd.io/

[37] Istio, https://github.com/istio

[38] Kolla, https://www.openstack.org/software/releases/queens/components/kolla

[39] Docker Documentation, https://docs.docker.com/

[40] CNNVD, http://www.cnnvd.org.cn/index.html

[41] NTI, https://nti.nsfocus.com/

[42] NetEase Cloud, https://www.163yun.com/product/repo

[43] USTC Mirror, http://mirrors.ustc.edu.cn/

[44] DaoCloud, https://www.daocloud.io/account/signup

[45] Aliyun, https://opsx.alibaba.com/mirror

[46] Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities, https://www.banyanops.com/blog/analyzing-docker-hub/

[47] CoreOS Clair, https://github.com/coreos/clair

[48] Shocker https://github.com/gabrtv/shocker

[49] 10 Things to Get Right for Successful DevSecOps, Gartner, October 2017, http://www.mottoin.com/107385.html

[50] Linux Capabilities, http://man7.org/linux/man-pages/man7/capabilities.7.html

[51] Tomoyo Linux, https://tomoyo.osdn.jp/index.html.en

[52] Linux AppArmor, https://gitlab.com/apparmor/apparmor/wikis/home/

[53] SELinux, https://selinuxproject.org/page/Main_Page

[54] Linux-grsec, https://wiki.archlinux.org/index.php?title=Linux-grsec&redirect=no

[55] CIS Benchmarks, https://www.cisecurity.org/cis-benchmarks/

[56] Docker Registry, https://docs.docker.com/registry/

[57] 2018 Docker Usage Report, https://sysdig.com/blog/2018-docker-usage-report/

[58] 2015 NSFOCUS SDS White Paper, http://blog.nsfocus.net/software-defined-security-whitepaper/

[59] Dirty Cow, https://dirtycow.ninja/

[60] Tesla, https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/

[61] Kubernetes Pod Security Policies, https://kubernetes.io/docs/concepts/policy/pod-security-policy

[62] Kubernetes-announce, https://groups.google.com/forum/#!forum/kubernetes-announce

[63] https://kubernetes.io/docs/reference/issues-security/security/

[64] Shiro, https://shiro.apache.org/

[65] Spring Security, https://spring.io/projects/spring-security

[66] What is DevSecOps, https://www.redhat.com/en/topics/devops/what-is-devsecops

[67] 10 Things to Get Right for Successful DevSecOps, https://www.gartner.com/doc/reprints?id=1-4I8FBLQ&ct=171016&st=sg

[68] Contiv, https://github.com/contiv

[69] Cilium, https://github.com/cilium/cilium

[70] kube-router, https://github.com/cloudnativelabs/kube-router

[71] Envoy Proxy, https://www.envoyproxy.io/

[72] Grafeas, https://grafeas.io/

[73] Clairctl, https://github.com/jgsqware/clairctl

[74] CIS, Center for Internet Security, https://www.cisecurity.org/

[75] Docker CIS benchmark, https://github.com/docker/docker-bench-security

[76] NeuVector Kubernetes-cis-benchmark, https://github.com/neuvector/kubernetes-cis-benchmark

[77] Aqua Security, kube-bench, https://github.com/aquasecurity/kube-bench

[78] The Update Framework, https://github.com/theupdateframework/specification

[79] Notary, https://github.com/theupdateframework/notary

[80] Cloud Native Computing Foundation, https://www.cncf.io/

[81] Spiffe, https://github.com/spiffe/spiffe

[82] Spire, https://github.com/spiffe/spire

[83] Open Policy Agent, OPA, https://www.openpolicyagent.org/

[84] NeuVector, https://neuvector.com/

NSFOCUS Innovation Center

The NSFOCUS Innovation Center is a department committed to researching state-of-the-art technologies, which is made up of the Star Cloud Laboratory (Cloud Security Laboratory), Security Big Data Analysis Laboratory, and IoT Security Laboratory. As an important member of the "Haidian Postdoctoral Workstation of Zhongguancun Haidian Science Park", the NSFOCUS Innovation Center has established a joint postdoctoral fellowship program with Tsinghua University. Their technological outcomes include various national research subjects, national standards, patents, high-level academic papers, and specialized publications. The NSFOCUS Innovation Center keeps exploring frontier technologies in the information security field and, based on its years of security practices and by leveraging the company's resources and state-of-the-art proprietary technologies, proposes conceptual prototypes before delivering incubation products and creating significant economic value.

NSFOCUS Star Cloud Laboratory

The NSFOCUS Star Cloud Laboratory is a team that researches on cloud computing security, with doctoral and master graduates from top universities in China. Its main research directions include security issues and solutions related to cloud computing systems (public/government/industry/private clouds), virtual network security issues (such as SDN/NFV), Docker/Kubernetes/Service Mesh/Serverless and other cloud-native security issues and solutions, 5G security issues and solutions in aspects of access security, slicing network security, and privacy protection, as well as security issues and solutions of edge computing in aspects of computing, storage, and networks. Currently, the NSFOCUS Star Cloud Laboratory has released several white papers/special themed books, such as *2015 NSFOCUS SDS White Paper, 2016 NSFOCUS SDS White Paper, SDS: Network Security of SDN/NFV. Also*, it is a member of the Cloud Security Alliance (CSA) Greater China and Software Defined Network Function Virtualization (SDNFV), and participated in making industrial standards and specifications together with China Mobile, China Unicom, and China Telecom. The NSFOCUS Star Cloud Laboratory has participated in a host of national, provincial, municipal, and industrial innovational research subjects, including the 863 Program, research subjects of the Technical Projects of Beijing Municipal Science & Technology Commission, and those of the China Information Technology Security Evaluation Center. In terms of product implementation, it has successfully launched the NSFOCUS cloud security solution, which has been applied to standard products and solutions by the NSFOCUS product team to provide comprehensive cloud computing security protection solutions for private clouds and industrial clouds of governments and carriers, and financial and education sectors.

NSFOCUS

THE EXPERT
BEHIND GIANTS

Over years, NSFOCUS has been committed to defense researches in the cybersecurity realm, providing most competitive security products and solutions for governments, carriers, and financial, energy, Internet, education, and medical sectors, ensuring customers' business continuity. To these customers, NSFOCUS lives up to the reputation of a trustworthy expert.

www.nsfocusglobal.com