Hello everyone! My name is ZhouTianyu, today I will introduce a paper named "To Docker or Not to Docker: A Security Perspective". This article is mainly about docker and its security problems. Now let's start from the overview.

There are four parts in this presentation. At first, we will take a brief look at the background of this paper. Next, I will introduce the basic components of Linux Containers and Docker. Finally, we focus on the Docker Security mechanisms and the challenges in Docker usages.

Cloud computing is inherently rooted in virtualization technologies.

Recently, new lightweight virtualization technologies such as containers have become increasingly popular and nowadays an essential part of cloud offerings.

Containers also tightly integrate into the host operating system, reducing the software overhead imposed by virtual machines (VMs).

However, this tighter integration also increases the attack surface, raising security concerns.

Existing work on container security focuses mainly on the relationship between the host and the container.

However, containers are now part of a complex ecosystem, which includes containers and various repositories and orchestrators.

And there are third-party elements running on different platforms provided by different providers. This can cause multiple vulnerabilities that an adversary could exploit to penetrate the system.

These two parts have not been fully investigated yet.

This paper then focus investigation on the Docker.

It’s mainly because there are plenty of users and the Docker needs to ensure security, besides, Docker is already running in some environments, making it possible to run experiments and explore the practicality of some attacks.

Here we take a brief look at Containers and the Docker.

Figure 1 shows how virtualization hypervisors (Figures 1a and 1b) compare to a container (Figure 1c).

The container provides near-bare-metal performance and offers the possibility of seamlessly running multiple versions of applications on the same machine. New instances of containers can be created quickly to face a customer demand.

Recent Linux-based container solutions rely on kernel support—that is, a userspace library to provide an interface to system calls and frontend applications. There are two main kernel implementations: Linux container (LXC) implementations using cgroups and namespaces, and the OpenVZ patch.

Containers can be integrated in a multitenant environment, thus profiting from resource sharing to increase average hardware use. This is achieved by sharing the kernel with the host machine.

Unlike VMs, containers don’t embed their own kernel, they run directly on the host kernel. This shortens the syscalls execution path by removing the guest kernel and the virtual hardware layer. Additionally, containers can share software resources (such as libraries) with the host, hence avoiding code duplication. The absence of kernel and some system libraries make containers very lightweight (image sizes can shrink to a few megabytes), which enables a quick boot process.

As Figure 2 shows, the Docker ecosystem includes various components.

Docker provides a **specification** for container images and runtime, including Dockerfiles that allow a reproducible building process (Figure 2a).

Docker software implements this specification using the **Docker daemon**. The repositories include a central repository, the **Docker hub**, which lets developers upload and share their images, along with a trademark and bindings with third-party applications (Figure 2b).

Finally, the build process fetches code from external repositories and holds the packages that will be embedded in the images (Figure 2c).

**Docker specification**

The specification’s scope is container images and runtime. Docker disk images are composed of a set of layers, along with metadata in the JavaScript Object Notation (JSON) format.

Docker can build images in two ways. It can launch a container from an existing image (docker

run), perform modifications and installations inside the container, and then stop the container and save its state as a new image (docker commit). This process is close to the classical VM installation, but must be performed at each image rebuild (such as for updates); because the base image is standardized, the sequence of commands is exactly the same. To automate this process, Dockerfiles (Figure 2a) let users specify a base image and a sequence of commands

to be performed to build the image, along with other options—such as exposed ports—specific to the image. The image is then built with the docker build command.

**Docker internals**

Docker containers create a wrapped, controlled environment on the host machine, and we can run arbitrary code within it safely. This isolation is achieved through two main kernel features—kernel namespaces and control groups (cgroups).

Namespaces are used to split the view that processes have of the system. Currently, the kernel has six different namespaces—PID, IPC, NET, MNT, UTS, and USER—that isolate various aspects of the system. Each of these namespaces has its own kernel internal objects related to its type, and each gives processes a local instance of some paths in the /proc and /sys filesystems.

The cgroups are a kernel mechanism to restrict the resource usage of a process or group of processes. Their goal is to prevent a process from taking all available resources and starving other processes and containers on the host. Controlled resources include CPU shares, RAM, network bandwidth, and disk I/O.

**Docker daemon**

The Docker software runs as a daemon on the host machine. It can launch containers, control their isolation level, monitor them to trigger actions (such as restart).

The software can change IP tables rules on the host and create network interfaces.

It’s also responsible for managing container images, including pulling and pushing images on a remote registry (such as the Docker hub), building images from Dockerfiles, and signing images. The daemon itself runs as root (with full capabilities) on the host and is remotely controlled through a Unix socket. Alternatively, the daemon can listen on a classical TCP socket. We call discuss the remote access later.

**Docker hub**

The Docker hub online repository lets developers upload their Docker images and lets

users download them.

Developer repositories are namespaced—that is, their name is “developer/repository.” Official

repositories also exist, directly provided by Docker authority.

The Docker daemon, hub, and repositories are similar to a package manager, with a local daemon installing software on both the host and the remote repositories. Some of these repositories are official, while others are unofficial and provided by third parties.

**Docker Security Overview**

Docker security relies on three factors: isolation of processes at the userspace level managed by the Docker daemon, enforcement of this isolation by the kernel, and network operations security.

**Isolation**

Docker containers rely on Linux kernel features, including namespaces, cgroups, hardening, and capabilities.

Namespace isolation and capabilities drop are enabled by default, but cgroups limitations

aren’t; they must be enabled through -a -c options on container launch.

The default isolation configuration is very strict.

The only flaw is that all containers share the same network bridge, enabling Address Resolution Protocol (ARP) poisoning attacks between containers on the same host.

However, Docker’s global security can be lowered by options, triggered at container launch, that give extended access on some parts of the host to containers. Additionally, security configuration can be set globally through options passed to the Docker daemon.

This includes options lowering security, such as the –insecure-registry option, which disables the Transport Layer Security (TLS) certificate check on a particular registry.

Options that increase security—such as the –icc=false parameter, which forbids network communications between containers and mitigates the ARP poisoning attack—are available, but they prevent multicontainer applications from operating properly, and hence are rarely used.

**Host Hardening**

Host hardening through Linux kernel security modules enforces security-related limitation constraints imposed on containers (such as compromising a container and escaping to the host operating system). Currently SELinux, Apparmor, and Seccomp are supported with available default profiles.

While default hardening protects the host from containers, it doesn’t protect containers from other containers. This security aspect can be addressed by writing specific profiles that depend individually on the containers.

**Network Security**

To distribute images, Docker verifies images downloaded from a remote repository with a hash and the connection to the registry is made over TLS (unless explicitly specified otherwise). Moreover, the Docker Content Trust architecture now lets developers sign their images before pushing them to a repository.

Content Trust relies on the update framework (TUF), which was specifically designed to address package manager flaws. TUF can recover from a key compromise, mitigate replay attacks by embedding expiration timestamps in signed images, and so on.

The Docker daemon is remote-controlled through a socket, making it possible to perform any Docker command from another host. By default, the socket used to control the daemon is a Unix socket, but it can be changed to a TCP socket. Access to this socket lets attackers pull and run any container in privileged mode, thereby giving them root access to the host.

Therefore, the connection must be secured with TLS (–tlsverify), which enables both encryption and authentication of the two sides of the connection(and requires additional certificate management).

**Docker Usages: Security Challenges**

Now it’s term for the last section. Let go through the usages of the Docker first.

**Docker Usages**

We can distinguish three types of Docker usages.

Recommended usages are those that Docker was designed for, as explained in the official documentation. Docker developers recommend a micro-service approach—that is, a container must host a single service. Therefore, a Docker container isn’t considered a VM: ~~there’s no package manager, no init process, no sshd to manage it~~. All administration tasks ~~(container stop, restart, backups, updates, builds, and so on)~~ must be performed via the host machine, which implies that the container’s admin has root access to the host.

Docker developers also recommend a reproducible and automated deployment of applications. Docker images should be built anywhere through a generic build file (Dockerfile) which specifies the steps to build the image from a base image. This generic way of building images makes the process and the resulting images depending only on the kernel and not on the installed libraries.

Some system administrators or developers use Docker as a way to ship complete virtual

environments and update them regularly, turning their containers into VMs. Although this is convenient because it limits system administration tasks to minimum, as we describe later, it has several security problems.

With containers embedding enough software to run a full system ~~(logging daemon, ssh server,~~

~~and even sometimes an init process)~~, it’s tempting to perform administration tasks from within the container, which is completely opposed to Docker’s design.

Indeed, some of these administration tasks require root access on the container, while other administration actions (~~such as mounting a volume in a container~~) could require extra capabilities that Docker drops by default.

**Adversary Model**

Because time is limited, here we only discuss the direct adversaries.

Direct adversaries can sniff, block, inject, or modify network and system communications, and they directly target the production machines.

Locally or remotely, direct adversaries can compromise several system components:

*In-production containers*

With containers from an Internet-facing container service, for example, attackers can gain root privileges on a related container. Then, from the compromised container, they can make a denial-of-service (DoS) attack on containers located on the same host operating system.

*In-production host operating system*

From a compromised container, for example, attackers can gain access to critical host operating system files—that is, launch a container escape attack.

*In-production Docker daemons*

In this case, for example, attackers might lower the default security parameters to launch Docker containers from a compromised host operating system.

*The production network*

From a compromised host operating system, attackers can redirect network traffic and so on.

Vulnerabilities found in Docker and the libcontainer mostly concern file system isolation:

Chroot escapes, path traversals and access to special fie systems on the host. These specific vulnerabilities are all patched as of Docker 1.6.2.

Because container processes often run with user ID 0, they have read and write access on the whole host filesystem when they escape, which lets them overwrite host binaries, leading to a delayed arbitrary code execution with root privileges.

**Vulnerabilities Affecting Docker Usages**

*Insecure local configuration*

Docker’s default configuration on local systems following recommended usages is relatively secure as it provides isolation between containers and restricts containers’ access to the host.

However, widespread usages take advantage of options~~—given either to the Docker daemon on startup or to the command launching a container—~~ that give containers extended access to the host. When used with untrusted containers, these options trigger many security concerns.

Along with these runtime container options, several settings on the host can influence potential attacks. Even basic properties can at a minimum trigger DoS.

*Weak local access control*

The docker-default profile gives containers complete access on network devices and filesystems with a full set of capabilities, and contains a small list of deny directives, which forms a blacklist.

These vulnerabilities are relevant to all usages and could lead to the attacks such as DoS or container escapes.

*Image distribution vulnerabilities*

While doing automated deployment, there will be a setup to build new image. It adds several external intermediary steps to the code path, each of which has its own authentication and attack surface, increasing the global attack surface.