PROJECT REPORT

ON

Operating System Virtualization
Containers, Docker and Unikernel !

ELEG-866 - INDEPENDENT STUDY

UNIVERSITY OF DELAWARE

Submitted by:

Kartik Khanna

CONTENTS

1. **Introduction**
2. **OS Virtualization Implementation**
3. **Docker**
   - **Terminologies**
   - **LXC vs Docker**
   - **Dockerfile**
4. **Terminologies behind Docker**
   - **Namespaces**
   - **Control Group**
5. **Security – Host System**
   - **Linux Capabilities**
   - **SELinux**
   - **AppArmor**
6. **Security – Containers**
   - **Images**
   - **Updates**
7. **References**

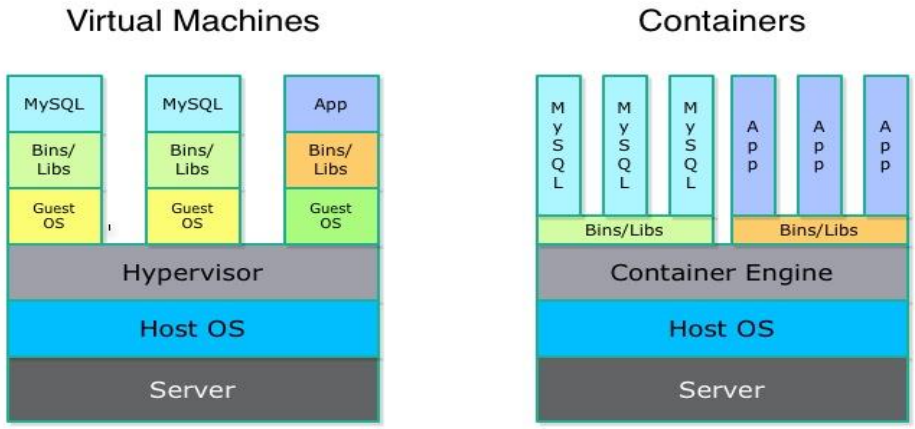
INTRODUCTION

Definition

- Operating-system-level virtualization is a server virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. Such instances, which are sometimes called containers, software containers. That means they are much more efficient than hypervisors in system resource terms. Instead of virtualizing hardware, containers rest on top of a single Linux instance. This in turn means you can “leave behind the useless 99.9% VM junk, leaving you with a small, neat capsule containing your application.
- They are more akin to an enhanced chroot than to full virtualization like Qemu or VMware, both because they do not emulate hardware and because containers share the same operating system as the host.

It’s NOT Virtual Machine

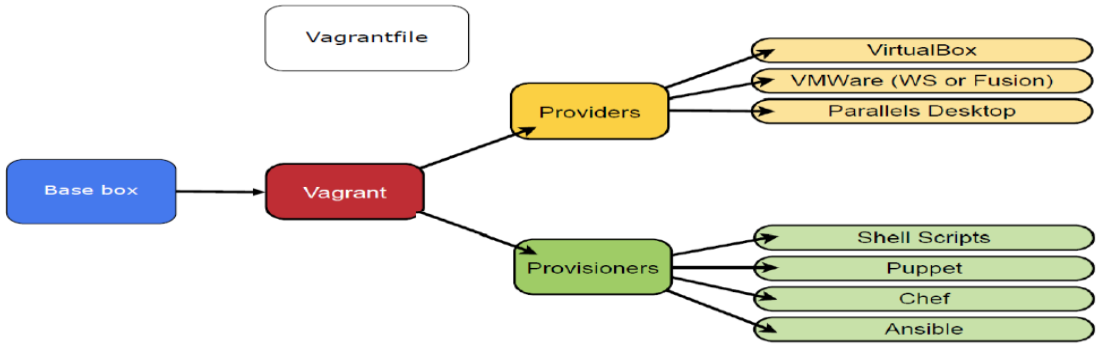
- Full machine virtualization offers greater isolation at the cost of greater overhead, as each virtual machine runs its own full kernel and operating system instance. A full virtualized system gets its own set of resources allocated to it, and does minimal sharing. You get more isolation, but it is much heavier (requires more resources)
- Any container technology, as far as the program is concerned, it has its own file system, storage, CPU, RAM, and so on. The key difference between containers and VMs is that while the hypervisor abstracts an entire device, containers just abstract the operating system kernel. VM's will emulate the hardware; each VM thinks it's a computer with its own CPU's, RAM, hard disk, kernel, etc



It's NOT Vagrant

If your code doesn't work on different environment then vagrant is the solution for you

- Instead of installing all the dependencies required for a specific code to run, vagrant allows you to script and package the VM config. So along a VM comes a provisional script which will automatically install all the necessary dependencies / applications required.
- Vagrant is still a virtual machine, albeit one with more powerful features than the bog-standard VM tools out there; for instance you can integrate Vagrant with CM tools such as Puppet and Chef to provision your own VM setups and configurations.
- Some special boxes are named basebox, which are the bare minimum virtual machines required for Vagrant to function. A basebox is often created manually or some tools like Packer according some requirements for interaction to Vagrant. Provider is a VM vendor, for example, VirtualBox. Vagrant uses providers to differentiate launching and configuring process of virtual machines. Provisioner is responsible to install and configure software packages required to develop or run applications. It often runs automatically without interactions with users



OS VIRTUALIZATION CAN BE IMPLEMENTED BY USING VARIOUS METHODS

- **Chroot** is an operation that changes the apparent root directory for the current running process and their children. A program that is run in such a modified environment cannot access files and commands outside that environmental directory tree
- **LXC** (**Linux Containers**) is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel. LXC combines kernel's cgroups and support for isolated namespaces to provide an isolated environment for applications.
- **Docker** is a project by dotCloud now Docker Inc released in March 2013, initially based on the LXC project to build single application containers. Docker has now developed their own implementation libcontainer that uses kernel namespaces and cgroups directly. The idea behind Docker is to reduce a container as much as possible to a single process and then manage that through Docker
- FreeBSD has Jails, Solaris has Zones and there are other container technologies like OpenVZ and Linux VServer that depend on custom kernels impeding adoption.

Getting Started with DOCKER

Terminologies

- **Docker File** - Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish
- **Docker Image** - Docker images are the basis of containers. Each time you've used docker run you told it which image you wanted. A Docker image is a series of data layers on top of a base image(Fig. 4). Every Docker image starts from a base image, such as Ubuntu base image or OpenSuse base image. When users make changes to a container, instead of directly writing the changes to the image of the container, Docker adds an additional layer containing the changes to the image. For example, if the user installs MySQL to an Ubuntu image, Docker creates a data layer containing MySQL and then adds to the image. This process makes the image distribution process more efficiently since only the update needs to be distributed.
- **Containers** – Running instance of an image is called containers. You can run many containers of a same image simultaneously

Installation guide

https://docs.docker.com/engine/installation/linux

Basic Commands

https://docs.docker.com/engine/userguide/containers/dockerizing/

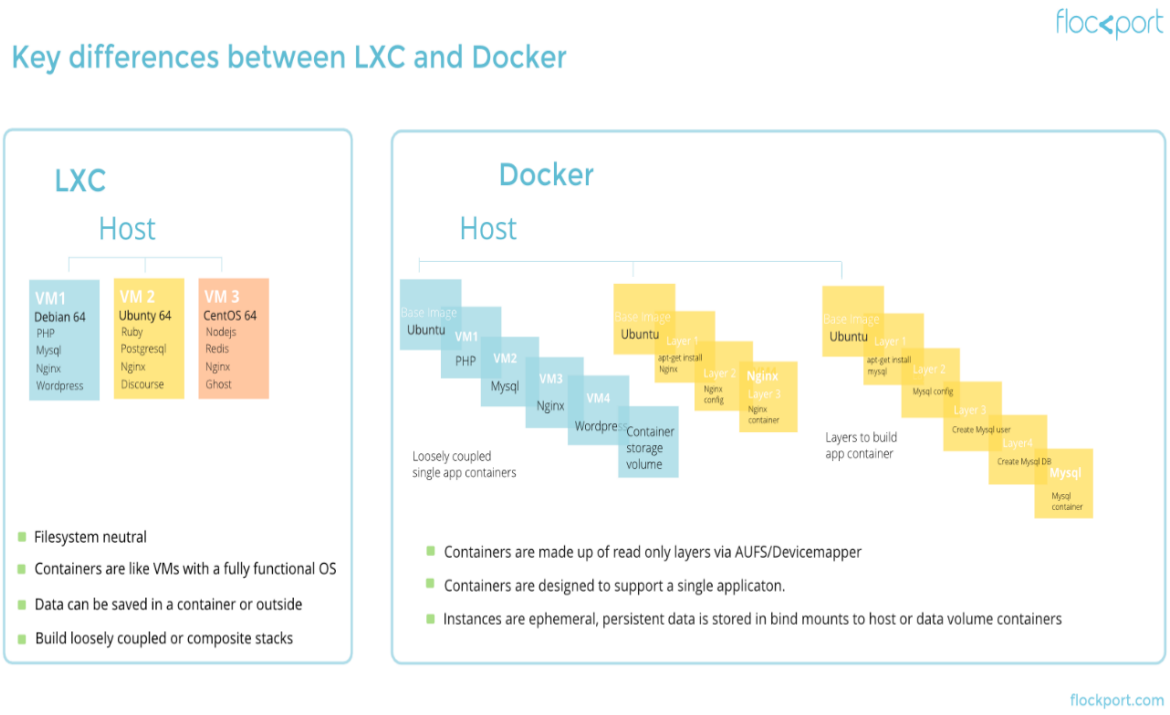
LXC vs Docker

Docker is not a replacement for LXC. "LXC" refers to capabilities of the linux kernel (specifically namespaces and control groups) which allow sandboxing processes from one another, and controlling their resource allocations.

On top of this low-level foundation of kernel features, Docker offers a high-level tool with several powerful functionalities:

- **Portable deployment across machines** Docker defines a format for bundling an application and all its dependencies into a single object which can be transferred to any docker-enabled machine, and executed there with the guarantee that the execution environment exposed to the application will be the same. LXC implements process sandboxing, which is an important pre-requisite for portable deployment, but that alone is not enough for portable deployment. If you sent me a copy of your application installed in a custom LXC configuration, it would almost certainly not run on my machine the way it does on yours, because it is tied to your machine's specific configuration: networking, storage, logging, distro, etc. Docker defines an abstraction for these machine-specific settings, so that the exact same docker container can run - unchanged - on many different machines, with many different configurations.

- **Application-centric**. Docker is optimized for the deployment of *applications*, as opposed to machines. This is reflected in its API, user interface, design philosophy and documentation. By contrast, the lxc helper scripts focus on containers as lightweight machines - basically servers that boot faster and need less ram. We think there's more to containers than just that.
- **Automatic build**. Docker includes a tool for developers to automatically assemble a container from their source code, with full control over application dependencies, build tools, packaging etc. They are free to use make, maven, chef, puppet, salt, debian packages, rpms, source tarballs, or any combination of the above, *regardless of the configuration of the machines*.
- **Component re-use**. Any container can be used as a "base image" to create more specialized components. This can be done manually or as part of an automated build. For example you can prepare the ideal python environment, and use it as a base for 10 different applications. Your ideal postgresql setup can be re-used for all your future projects. And so on.
- **Sharing**. Docker has access to a public registry (https://registry.hub.docker.com/) where thousands of people have uploaded useful containers: anything from redis, couchdb, postgres to irc bouncers to rails app servers to hadoop to base images for various distros. The registry also includes an official "standard library" of useful containers maintained by the docker team. The registry itself is open-source, so anyone can deploy their own registry to store and transfer private containers, for internal server deployments for example.



Format Of A Dockerfile

FROM – All the dockerfile starts with from which tell which base image you have to use.

MAINTAINER – It allows you to set author

RUN – Will execute that command and commits it. Two forms

1. Shell form, **RUN** /bin/bash -c echo $HOME'
2. Exec form, RUN ["/bin/bash", "-c", "echo hello"]

CMD - It's the default argument to container. There can only be only one CMD command in Dockerfile. If many then last CMD will execute

ENTRYPOINT - Without entrypoint, default argument is command that is executed. With entrypoint, cmd is passed to entrypoint as argument.

http://stackoverflow.com/questions/21553353/what-is-the-difference-between-cmd-and-entrypoint-in-a-dockerfile

LABEL - The LABEL instruction adds metadata to an image. A LABEL is a key-value pair.

EXPOSE - The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. The -p flag to publish a range of ports or the -P flag to publish all of the exposed ports

ENV - The ENV instruction sets the environment variable <key> to the value <value>

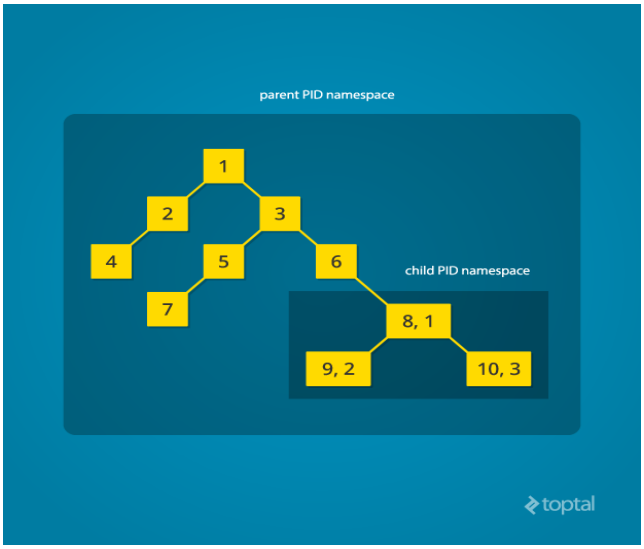
TECHNOLOGIES BEHIND DOCKER

- **LXC or libcontainer** – Virtualization format (since ver. 0.9 libcontainer is used)
- **Namespaces** wrap the operating system resources into different instances.
- **Cgroups** provide mechanism for accounting and limiting the resources which the processes in each container can access
- **Union file system** In order to work with multiple layers of an image as it were a single file system layer, Docker uses a special file system called Union File System (UnionFS). It allows files and directories in different file systems to be combined into a single consistent file system.

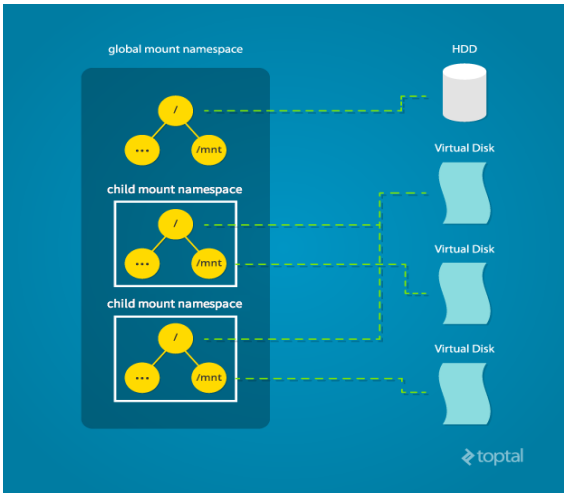
The use of these instances gives the processes running inside container the illusion that they have their own resources. Currently, Docker uses five namespaces to provide each container with a private view of the underlying host system **mount**, **hostname**, **inter-process communication** (IPC), **process identifiers** (PID), and **network**. Each of them works on specific types of system resources.

Namespaces

- **Process Isolation**
  This mechanism operates with the support of the PID namespaces, which isolate the process ID number space of a container from that of the host. Since PID namespaces are hierarchical a process can only see the other processes in its own namespace or in its "children" namespaces. So each container gets new PID namespace, host can affect the process but other container cannot.



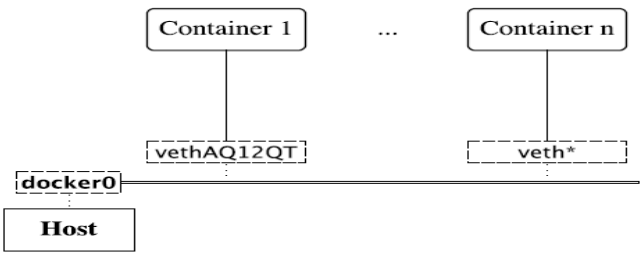
- **Filesystem Isolation**
  Docker uses the mount namespaces, also called the filesystem namespaces, to isolate the filesystem hierarchy associated with different containers. The mount namespaces provide the processes of each container a different view of the filesystem tree and restrict all the mount events occurring inside the container to only have impact inside the container. Docker also employs a mechanism called copy-on-write file system. When multiple containers are created on the same image, the copy-on-write file system allows each container to write content to its specific file system, thus preventing other containers from discovering the changes occurring inside the container.



- **Device Isolation**
  Container can access some important device nodes, such as /dev/mem (the physical memory), dev/sd_ (the storage) Thus, it is crucial to limit the set of device nodes that a container can access. The Device Whitelist Controller feature [3] of cgroups provides means to limit the set of devices that Docker allows a container to access. It also prevents the processes in containers from creating new device nodes Furthermore, Docker mounts container images with nodev, meaning that even if a device node was pre-created inside the image, the processes in the container using the image cannot use it to communicate with the kernel.

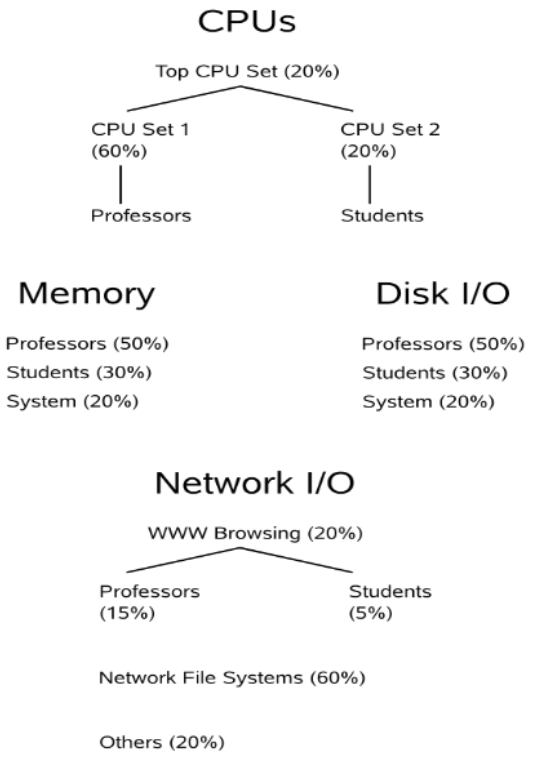
- **IPC Isolation**
  The IPC (inter-process communication) is a set of objects for exchanging data among processes, such as semaphores, message queues, and shared memory segments. Docker achieves IPC isolation by using the IPC namespaces, which allows the creation of separated IPC namespaces. The processes in an IPC namespace cannot read or write the IPC resources in other IPC namespaces

- **Network Isolation**
  Network isolation is important to prevent network-based attacks, such as Man-in-the-Middle (MitM) and ARP spoofing. Containers must be configured in such a way that they are unable to eavesdrop on or manipulate the network traffic of the other containers nor the host. For each container, Docker creates an independent networking stack by using network namespaces. Therefore, each container has its own IP addresses, IP routing tables, network devices, etc. This allows containers to interact with each other through their respective network interfaces, which is the same as how they interact with external hosts. By default, connectivity between containers as well as tithe host machine is provided using Virtual Ethernet Bridge. With this approach, Docker creates a virtual Ethernet bridge in the host machine, named docker0, that automatically forwards packets between its network interfaces. When Docker creates a new container, it also establishes anew virtual ethernet interface with a unique name and then connects this interface to the bridge. The interface is also connected to the eth0 interface of the container, thus allowing the container to send packets to the bridge. We note here that the default connectivity model of Docker is vulnerable to ARP spoofing and Mac flooding attacks since the bridge forwards all of its incoming packets without any filtering.



Control Groups (cgroups)

- Denial-of-Service (DoS) is one of the most common attacks on a multi-tenant system, where a process or a group of processes attempt to consume all of the resources of the system, thus disrupting normal operation of the other processes.
- Cgroups are the key component that Docker employs to deal with this issue. They control the amount of resources, such as CPU, memory, and disk I/O that any Docker container can use, ensuring that each container obtains its fair share of the resources and preventing any container from consuming all resources.
- cgroups are organized in a tree-structured hierarchy. There can be more than one hierarchy in the system. You use a different or alternate hierarchy to cope with specific situations.
- Every task running in the system is in exactly one of the cgroups in the hierarchy.



SECURING THE CONTAINER HOST SYSTEM

Linux Capabilities

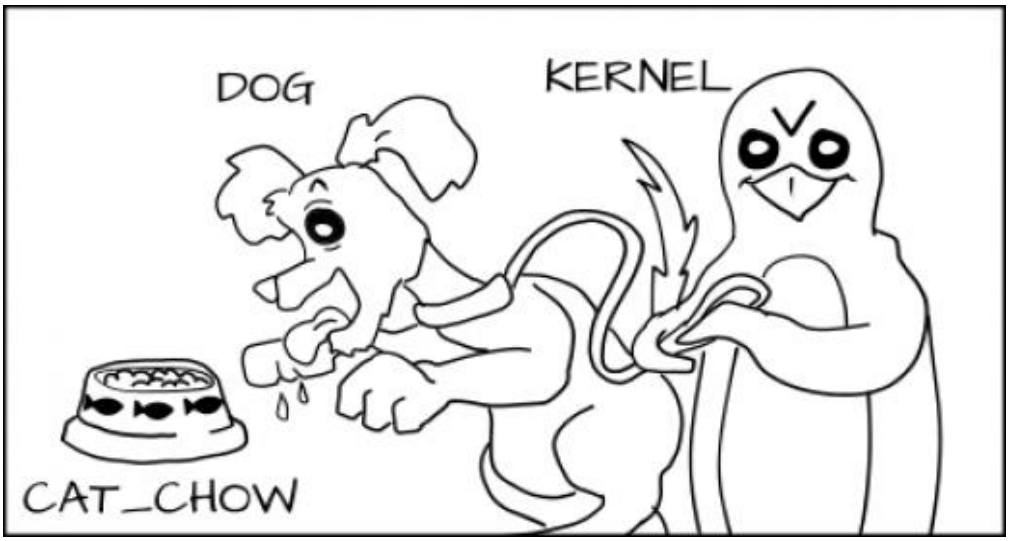
- First there are Capabilities as defined in computer science. A capability is a token used by a process to prove that it is allowed to do an operation on an object. The capability identifies the object and the operations allowed on that object.
- Previously the kernel skipped all permission checks on the privileged processes but conducted full permission checking on unprivileged processes. But from Kernel 2.2 Linux Capabilities allow you to break apart the power of root into smaller groups of privileges.
- Docker containers run on a kernel shared with the host system, it is unnecessary to provide full root privileges to a container, By default, Docker disables a large number of Linux capabilities from its containers in order to prevent an intruder to damage the host system even when the intruder has obtained root access within a container
  If you were building a container that would run ntpd or crony, which needs to be able to modify the host system time. The container would not run because it requires CAP_SYS_TIME. In older versions of docker, the container would have to run in --privileged mode, which turns off all security. Now in order to run an ntpd container, you could just run:

  ```
  docker run -d --cap-add SYS_TIME ntpd
  ```

SELinux

- SELinux is a security enhancement to the Linux system. Linux comes with the standard Discretionary Access Controls (DAC) mechanism (i.e., owner/group and permission flags of an object) to control the access to an object. SELinux provides an additional layer of permission checking, called Mandatory Access Control, after the standard DAC is performed.
- In SELinux, everything is controlled by labels. Every file/directory, process, and system object has a label. The administrator of the system uses these labels to write rules to control access between processes and system objects. These rules are called policies. The SELinux policies can be divided into three classes: Type enforcement, Multi-level security (MLS) enforcement, and Multi-category security (MCS) enforcement. With the DAC mechanism, owners have full discretion over their objects, meaning that if the owners are compromised, the attacker has control over all of their objects. In SELinux model, in contrast, the kernel manages and enforces all of the access controls over objects, not their owners.
- Docker uses two classes of policy enforcement: Type enforcement and MCS enforcement. The Type enforcement protects the host from the processes in containers, and the MCS enforcement protects a container from another container. With Type enforcement, Docker labels all container processes with svirt_lxc_net_t type and all content within a container with svirt_sandbox_file_t type. The processes running with svirt_lxc_net_t type can only access/write to the content labeled with svirt_sandbox_file_t type, but not to any other label on the system. Therefore, the processes running within containers can only use the content inside containers.

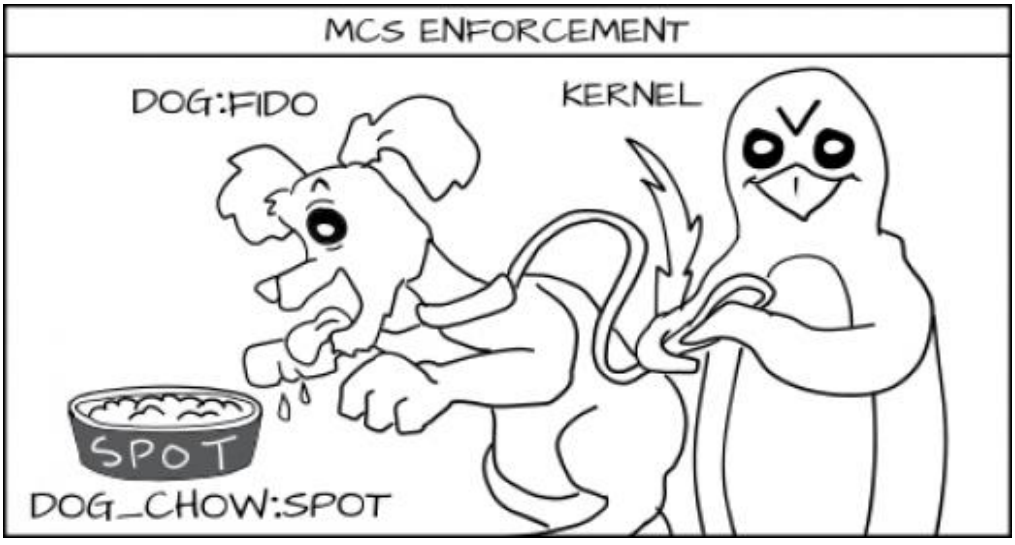
Type enforcement



Type Enforcement protects the host from the processes within the container

- However, only with this policy enforcement, Docker allows the processes in one container to have access to the content of other containers. MCS enforcement is necessary to solve this issue. When a container is launched, the Docker daemon picks a random MCS label and then puts this label on all of the processes and content of the container. The kernel only allow processes to access content with the same MCS label, thus preventing a compromised process in one container from attacking other containers.

Multi Category Security enforcement



Multi Category Security (MCS) protects one container from other containers

AppArmor

- AppArmor is also a security enhancement model to Linux based on Mandatory Access Control like SELinux, but restricting its scope to individual programs. It permits the administrator to load a security profile into each program, which limits the capabilities of the program.
- AppArmor supports two modes: enforcement mode and complain/learning mode. The enforcement mode enforces the policies defined in the profile. However, in the complain/learning mode, the violations of profile policies are permitted, but also logged. This log can be useful for developing new profiles later. On systems that support AppArmor, Docker provides an interface for loading a pre-defined AppArmor profile when launching a new container. This profile is loaded into the container in enforcement mode in order to ensure that the processes in the container are restricted according to the profile.

SECURITY WITHIN LINUX CONTAINERS

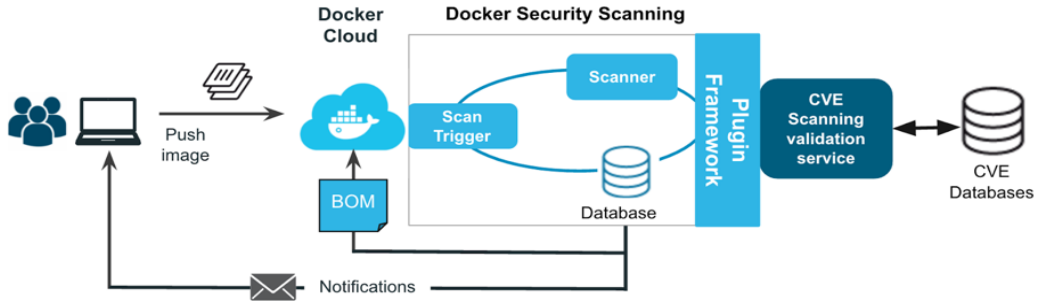
Securing the container host is critical but is only half the battle. The data inside the container, including the application, configuration files, and the operating system, is just as critical to the overall security of the environment.

Trusted images

All containers start with software that needs to run inside the container. Some containers, such as the ones started with systemd-nspawn, can simply re-purpose an existing executable from the host system and launch it in a container. Other containers are built using trusted packages from the host system's distribution with package management tools including debootstrap, yum, or dnf. This allows administrators to install the base OS from trusted sources and cryptographically verify each package. Docker takes a different approach. Users can simply download an existing image from Docker's public facing index using the docker pull command and apply customization on top of it. It is as simple as pulling a CentOS 7 image, writing a short file of customization instructions, and telling Docker to build the container. Docker stores the results of customizations, such as adding new packages or configuration files, as layers and applies those layers onto the original base image. This simplicity leads to security issues for administrators. Docker has implemented some initial checks for images when they are downloaded to ensure they were not altered during transit, but they do not vouch for the actual content found within the images themselves. It is up to the user to determine which images are trustworthy for their environment. In an automated study of images available in the public Docker showed that 30% of images contained serious security vulnerabilities. (Gummaraju, Desikan & Turner, 2015). In order to overcome Docker recently in May 2016 introduced security scanning of your images before they are deployed.

- **Binary image scan** providing detection of maliciously hidden code, including stripping and renaming of components.

- **Scans are container-aware**, addresses the Docker image by layers and conducts a binary scan of the bits. Does not require running an image, and reduces the risk of undetected security problems.
- **Notifications**: Nautilus automatically notifies the user when new vulnerabilities are discovered and reported into CVE database of images which have already been scanned.
- **Comprehensive Language support**: Nautilus supports all popular programming languages, including C, C++, Java, JavaScript, Ruby, Python, C#, Go, etc.
- **Support for major OS distributions**, including Ubuntu, Debian, CentOS, etc. In addition, Docker is committed to bringing this solution to all the OSs we support in the future.
- **Displays scanning data via visualization**, provides actionable intelligence for users and allowing publishers to address newly discovered vulnerabilities.



Container operating system updates

Unlike virtual machines, containers do not provide simple methods for applying operating system updates. Containers are generally considered immutable once they are built, and any updates to the container are often done by building a new container. Docker makes this process easier since the user can simply rebuild the container image with the same Dockerfile and get the latest operating system updates, so long as the Dockerfile contains commands for updating the operating system. Deploying the new container with the updated images to replace the old container can occur via a number of methods. For one-off or small container deployments, the old container could be stopped and the new container could be started in its place. Downtime is minimal due to the short time required to stop and start containers. A more automated approach would be to start a new container, or group of new containers, alongside the old containers. Monitoring systems would check the new containers to verify that they are responding properly, and then a quick load balancer change would shift traffic over to the new containers. Kubernetes has features that allow new pods to start alongside old ones, and then the service can be adjusted to point to the new containers.

Communication between containers

Host protections, such as namespaces and MAC policies, provide strong protection between containers on the host, but it is important to consider how containers can communicate with each other and with the host outside of those protections. Containers with unfiltered network access can communicate with

each other and the host if they are on the same network segment. Since containers have their own virtual network interface (thanks to network namespaces), users can attach the network interface to a variety of network devices. The simplest solution could be to use a Linux bridge and place all of the containers on the bridge. Communication is simple and fast, but not secure. A stronger solution would be to place containers on bridges with filtering applied or use VLANs to carve up new network segments. Modern systems may be able to use virtual network switching via OpenvSwitch on the host and achieve greater network separation. It is also important to consider other objects that may be shared between containers, including sockets or shared storage devices. System administrators should carefully consider the consequences of sharing objects between containers to limit the spread of a compromise.

Security responsibility

Developers appreciate containers because they can package their application, test it alongside its libraries, and verify that it will work in production. Operations teams appreciate containers because they get the applications in a cohesive package along with their dependencies and configurations. However, who owns the security of the container operating system, configuration files, and the application in this new world of containers? The responsibility of securing the operating system normally falls onto the operations team. However, if developers are writing applications and building a container with their application in it, how do operations teams ensure that the base operating system is secure? This is where frameworks with layered images, including Docker, can help. Operations teams can carefully maintain a base image with appropriate security controls, configurations, and package updates. As part of that configuration, they can specify where the package manager will receive trusted packages. Development teams can use that base image as the foundation for their containers and then add packages from those trusted repositories. If a serious vulnerability appears, the operations team would quickly update the base image and let the development team know that a container rebuild and redeployment is needed

- **DOCKER – TRY IT YOURSELF!!**

https://www.youtube.com/watch?v=VeiUjkiqo9E by **dotcloud**

https://archive.org/details/Containing_An_Attack_With_Linux_Containers by **Jay Beale**

REFERENCES

- https://www.flockport.com/lxc-vs-docker/
- https://www.researchgate.net/publication/270906436_Analysis_of_Docker_Security
- https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces
- https://docs.docker.com/
- https://opensource.com/business/14/9/security-for-docker