Virtual Machines and Sandboxing

Mihai Ordean

VM/Sandboxing Goals

- Have a basic conceptual understanding of VMs and their use cases
- Have a basic conceptual understanding of containerized applications and their use cases
  - Know the difference between VMs and containers
- Have concrete experience using a VM locally (UTM/VirtualBox)
  - Spin up an Ubuntu (arm64/amd64) image locally
- Have concrete experience using docker locally
  - Deploy a pre-built container locally and expose a port

Computers Within Computers

- Isolation & Testing (without extra hardware)
  - Virtual Machines allow you to emulate an entirely separate system within an existing system.
  - The application that oversees running VMs is called a hypervisor.
  - The computer running the hypervisor is called the host. The computer inside the VM is called the guest.

Use case for VMs

- **Isolation & Sandboxing**
  - **Running untrusted (or dangerous) code**
  - **Allowing untrusted users access to an isolated environment**
- Resource Allocation
  - Limit the maximum amount of resources an application or user may use
  - You might rent only part of the compute power available to you
- System Management
  - It is very easy to back up and restore an entire VM; shut it down, restart it, etc, without losing access to the host computer.
- Cross-platform testing
  - Test your apps on other operating systems without using an entire computer, or needing to reinstall a computer you're already using.
- Prototyping
  - Virtualize an entire network with multiple VMs and emulated routers/switches/etc. to prototype an entire network!
- Emulation
  - Emulate entirely different CPU architectures - slower - but allows for an incredibly diverse range of applications (including legacy applications) to run on your computer.

Emulation vs Virtualization

- **Emulation** involves recreating and modelling fully different computer architectures – basically mimicking hardware.
  - This is necessary when the operating system or software you're trying to run are written for a different class of hardware than the host machine.

- Examples:
  - **32bit** architectures running in **64bit**: different hardware
  - **arm** running on **x86:** different instruction sets
  - **Linux** binaries running on **Windows** (or the other way around): different binary formats.

Emulation vs Virtualization

- **Emulation** involves recreating and modelling different computer architectures– basically mimicking hardware.
  - This is necessary when the operating system or software you're trying to run are written for a different class of hardware (or OS) than the host machine.

- **Virtualization** works when the OS (or software) is built for the same class of hardware as the host machine.
  - Virtualization means "making one thing look or act like multiple things."
  - In this case, it means your computer hardware!
  - Virtualization makes use of your computer hardware directly, and is therefore faster than emulation. Usually by a lot.

Security concerns w/ virtualization

- When we're running a virtual machine, one of the key guarantees is that the guest cannot (without permission) access or affect the host.
  - This allows us to try things (like `rm -rf /`) without worrying about destroying our host computer.
  - Used for security research, etc.
- **Ok, but what is sandboxing then?**

Virtualization is supported by hardware

In order to provide virtual machines with complete CPU access and ensure security of the host system, CPU manufacturers have extensions (special instructions!):

- Intel VT-x
- AMD-V

The role of hypervisors

- Okay, so CPUs have extensions for virtualization; do hypervisors need to do anything else interesting?
  - Yes!
- There's more to provide to the guest than just CPU & Memory:
  - network access? → Virtual Networks
  - hard drive space? → Virtual Hard Drives
  - CD-ROM access (yes still!) → Disk Images

Bare Metal vs Hosted Hypervisors

Two types of hypervisors: **bare metal** (type 1) and **hosted** (type 2).

- Bare Metal hypervisors is when the hypervisor is the host.
  - This means the hypervisor is the "operating system."
- Hosted hypervisors are programs that run on a host operating system.
  - We're using type 2 hypervisors when we use UTM and VirtualBox

Let's try to put these in the context of OS operation and security.

Hypervisors

- **Bare Metal Hypervisors**
  - VMWare ESXi
  - Hyper-V
  - Proxmox
  - Xen
- **Hosted Hypervisors**
  - VirtualBox
  - VMWare Workstation
  - UTM (uses qemu under the hood)
  - qemu (also supports emulation)
  - Hyper-V
  - Parallels

Containers e.g., docker

- Light(-er) weight
  - Allows them to be easily distributed
  - Rather than virtualizing the entire OS, it continues to use the host's **kernel**/operating system as a "base" to service whatever is running within the container.
- Faster than VMs (usually)
  - Can also use an emulated system if necessary → runs within a VM
- Designed for ephemerality
  - Containers are "disposable" – any long-term data should be stored in separate persistent "volumes"

Containerization (in general)

You can take an application and wrap it in a container to ensure a consistent running environment.

- You can define the **operating system** it expects to use (**but not the kernel**)
- You can define the **CPU architecture** the program expects (and if the CPU architecture differs from the host, it will have to run within a virtual machine)
- You can define dependencies and other programs that the application expects to be installed
- You can define the "hard drive" layout the program expects
- All this, and the application gets **a level of isolation** from other applications on the system.

docker

docker is a popular tool to create and manage containers

Operational components:

- **Containers** are an ephemeral object representing a copy of a program, based on an Image
- **Images** are built from **Dockerfiles**, and represent a frozen copy of an application and everything needed to run it
- **Dockerfiles** are special scripts that are used to build images, including:
  - Instructions on adding files (e.g. program files) from your local system
  - Instructions on adding dependencies
- **Volumes** store persistent data even past the lifetime of a container.

docker

docker intuition:

- **Containers** are like the downloaded application
- **Images** are like the .zip, .msi, or .dmg that you download from the website
- **Dockerfiles** are like scripts that create the .zip/.msi/.dmg
- **Volumes** are like the places on your computer where your applications store data

docker

Let’s run a basic ubuntu system using Docker!

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

runs a new container based on an image

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Keep **STDIN** open (allows us to write things into the container)

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Allocate a **TTY** to the container (allows the container to receive things we write)

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

All together: "**Run interactively**"

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Specifies the **image** to run (the **latest** version of **ubuntu**)

Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Specifies which command to run within the image (**bash**)

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`. They have the following syntax:

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`. They have the following syntax:

FROM specifies the base of the image you're building.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`.
They have the following syntax:

FROM specifies the base of the image you're building.
In this case, we're basing our image off the **latest** version of **ubuntu**.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`. They have the following syntax:

RUN commands will run the given command inside a shell.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`.

They have the following syntax:

RUN commands will run the given command inside a shell.

These commands form **intermediate containers**; each command builds off from the previous.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`. They have the following syntax:

WORKDIR specifies a new working directory from that point onwards. (it's like a cd that sticks)

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`.
They have the following syntax:

ADD will copy files from your local directory (relative to where the Dockerfile is located) into the image.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`. They have the following syntax:

We can run **make** at this point because we installed it earlier.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`. They have the following syntax:

ENTRYPOINT describes what should happen when we run the image.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`.

They have the following syntax:

ENTRYPOINT describes what should happen when we run the image.

Here it says we should run the ./`hello_world` program, which we just built

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```

Dockerfiles

Dockerfiles, are script files which always have the name `Dockerfile`.

They have the following syntax:

At the end, **docker** will combine all the intermediate containers into a static **image**. Each command forms a **layer** of that image.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /hello_world
ADD ./hello_world/ ./
RUN make clean; make
ENTRYPOINT ["./hello_world"]
```