Missing Semester

Missing Semester is back for semester two!

Missing Semester is a series of lectures which teach you the stuff you'll need as a computer scientist out in industry.

If you have missed the first semester than that's okay! The talks for this semester are more high level and all of the course content is available online.

Find out more at
missingsemester.afnom.net

Schedule

03/02	Introduction to IDEs 🍕🍕🍕
10/02	Constructive Generative AI use
17/02	Cloud Computing
24/02	Using CLI tools at your job

Every **Monday @ 2pm** -- SportEx LT1
starting from week three

Missing Semester is student-led and ran by volunteers from **AFNOM** and **CSS** <3

A.F.N.O.M

Virtual Machines and Sandboxing

Mihai Ordean

The Sandbox…

- You have a lot of bad code that take input from **hostile users**
  - Often written in C/C++
- The sandbox generally covers an entire process
  - That way one can take advantage of operating-specific features that allow a process to restrict what it is allowed to do

Why Sandbox at all?

- The sandbox is mostly good at making C/C++ memory exploits no longer exploitable
  - Now the attacker needs to both exploit the C code AND exploit a weakness in the sandbox

- Defence in depth…
  - But why bother with defence in depth?
- Because it is cheaper!
  - Cheaper to keep using the same bad C and C++ code and put it in a letterbox than it is to actuallyrewrite the code in a secure language!

Assume Sandboxed Code is Malicious Code

- The sandbox **must** work if the code within it is compromised by an attacker
  - So simply assume for the purposes of the sandbox that the code you are running is already compromised

- Must ensure **complete mediation**:
  - After the sandbox itself hands control over to the running code, that code **must** not be able to access any resource beyond that necessary to perform its operation
  - And that which it can access **must** be checked and treated as potentially hostile input

Emulation is Not Security

- Emulation primitives (Virtual Machines, etc.) are often not designed as security sandboxes!
  - Relying on something not designed for sandboxing can be a problem!

Old School Unix Sandboxing: the chroot jail

- People have wanted sandboxes for a long time…
  - Far longer than the OSs have provided fine grained mediation necessary to create sandboxes
- The gen-1 Unix Sandbox:
  - The **chroot** system call changes the definition / for the invoking process
- This enforces the following property:
  - A process (and all processes it invokes, directly or indirectly) can not read or write to any new file outside the new directory
  - But can still access existing files

Limitations of chroot

- It is a privileged operation! Because you can do things that would compromise the system otherwise:
  - Create a directory with a file name etc/sudoers with the appropriate context
  - Now chroot to that directory
  - Now invoke sudo
  - Voila, you have root!
  - So, any program using chroot must then drop it privileges to run as "nobody" or an otherwise unknown user

- It does not affect **system call** operation
  - A "jailed" process can still access the network, call the kernel (and therefore perhaps kernel bugs).

Using `seccomp` to create a sandbox

- Split things conceptually into a **broker** and one or more **targets**
  - The targets are the sandboxed elements
  - The broker controls the targets and is the trusted base
- The broker starts up the targets
  - Using fork or clone
  - Establishes communication channels to the targets
  - Starts the target processes running
- The target process then invokes the **`seccomp`** system call to give up any privileges

Process communication

- Standard Unix pipes/file descriptors
  - Shove bits to/from the sandboxed process
  - Just like a network program
- Shared Memory
  - A common pool of memory that both can read and write to
  - Need to also use a semaphore to ensure coherency

Software using seccomp (or seccomp-bpf)

- **Android:** uses a seccomp-bpf filter in the zygote since Android 8.0 Oreo.
- **Systemd**: sandboxing options are based on seccomp.
- **QEMU**: the Quick Emulator, the core component to the modern virtualization together with KVM uses seccomp
- **Docker**: Docker can associate a seccomp profile with the container using the --security-opt parameter.
- **Chrome:** seccomp-bpf is used to sandbox the renderers.
- **OpenSSH:** has supported seccomp-bpf since version 6.0.
- **Mbox:** uses *ptrace* along with seccomp-bpf to create a secure sandbox with less overhead than **ptrace** alone.
- **LXD:** a Ubuntu "hypervisor" for containers
- **Firefox and Firefox OS**
- **Tor:** since 0.2.5.1-alpha

https://en.wikipedia.org/wiki/Seccomp

Understanding VMs

- 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 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.

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

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**)

Use this shell to run a command inside your container!

- e.g. `cat /etc/os-release`

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
```

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
```

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
```

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
```

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
```

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
```

Dockerfiles

Using the Dockerfile:

Building a container using the Dockerfile

```
docker build -t mycontainer .
```

Running a container

```
docker run –it mycontainer
```

Building software inside containers

Write a Dockerfile **to:**

1. Download **nginx** from:
   - https://nginx.org/download/nginx-1.25.3.tar.gz
   - you can use the `wget` Linux tool for downloading links

2. Build **nginx:**
   - Command to build on linux: `./configure && make && make install`
   - **TIP**: to change directories inside a container in the DOCKERFILE use the `WORKDIR` instruction

3. Automatically run **nginx** using the ENTRYPOINT DOCKERFILE instruction:
   - **TIP**: use `/usr/local/nginx/sbin/nginx -g 'daemon off;'` to prevent nginx from forking to the background

!If you don't know the list of dependencies required to build nginx try running the commands inside an interactive container initially!

Dockerfiles

```
FROM ubuntu:latest
RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get -y install wget tar sed make gcc libpcre3-dev
libssl-dev zlib1g-dev libgd-dev
RUN wget https://nginx.org/download/nginx-1.25.3.tar.gz
RUN tar -xvf nginx-1.25.3.tar.gz
WORKDIR nginx-1.25.3
RUN ./configure
RUN make
RUN make install
ENTRYPOINT /usr/local/nginx/sbin/nginx -g 'daemon off;'
```

Dockerfiles

You can run a container in the background using:

```
docker run -d mynginx
```

You can map ports between the container and the broker using p:

```
docker run -dp 8080:80 mynginx
```

Dockerfiles

You can run a container in the background using:

```
docker run -d mynginx
```

You can map ports between the container and the broker using p:

```
docker run -dp 8080:80 mynginx
```

Now try to access your webserver on 127.0.0.1:8080 on the **host!**

Dockerfiles

Now let's use a **volume**:

```
docker run -dp 8080:80 \
-v ./html:/usr/local/nginx/html mynginx
```

`./html` a directory on the host/broker

`/usr/local/nginx/html` a directory inside the container

Dockerfiles

Now let's use a **volume**:

```
docker run -dp 8080:80 \
-v ./html:/usr/local/nginx/html mynginx
```

`./html` a directory on the host/broker

`/usr/local/nginx/html` directory inside the container

Now try to modify the contents of **./html**!

Dockerfiles

Try and create a **DOCKERFILE !!!** that performs some malicious/faulty action:

e.g.,: `mv /etc /etc.old`

Run the **DOCKERFILE**! interactively and assess the "damage", by running some commands e.g.,: `whoami`

Dockerfiles

```
# docker build -t sec .
# docker run -it sec
# whoami
FROM ubuntu:latest
RUN rm -rf /etc || true
```