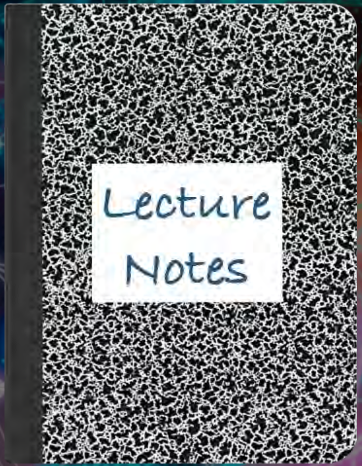



## CS 419: Computer Security

# Week 8: Part 2
# Containment & Isolation

Paul Krzyzanowski

# Risk of application compromise

*Everything we've discussed so far assumes we can prevent attacks through perfect code. But what if we can't?*

- Some services run as root
- What if an attacker compromises the app and gets root access?
  - Create a new account
  - Install new programs
  - Exfiltrate any data
  - "Patch" existing programs (e.g., add malicious changes)
  - Modify configuration files or services, change the IP address of the system
  - Add new startup scripts (launch agents, cron jobs, etc.)
  - Change file permissions (or ignore them!)
- Even without root, what if you run a malicious app – or exploit a path traversal bug?
  - It has access to all your files
  - Can install new programs in your search path
  - Communicate on your behalf

# Is access control good enough?

- Limit damage via access control
  - E.g., run services as a low-privilege user
  - Set proper read/write/search controls on files … or role-based policies
- ACLs are based on users, not applications
  - Processes run with the privilege of the user
  - Workaround: create a dummy user and run a *setuid* process with that user as the owner
  - Cannot set permissions for a specific process: “access this file and nothing else”
  - At the mercy of default (other) permissions
- We are responsible for setting the protections of every file on the system that could be accessed by an application
  - And hope users don’t change that
  - Or use more complex mandatory access control mechanisms … if available

*Limited and not high assurance*

# Problems with traditional access control

- All-or-nothing privileges
  - User runs program – it gets ALL the user's privileges
  - If an attacker compromises it, the attacker has all the user's privileges
- No granular control of file access
  - Program needs to read one config file
  - But gets access to ALL the files a user can read
- Privilege escalation is catastrphic
  - A small vulnerability may lead to root access – root can do anything!
- Trust boundaries
  - We trust our code (maybe)
  - We don't trust dependencies (100s of them)
  - We definitely don't trust user input
  - No way to express these trust levels

# Containment: prepare for the worst

- We realize that an application may be compromised
  - We want to run applications we may not completely trust

- Limit an application to use a subset of the system's resources
  - Defense-in-depth: even if we have other protection mechanisms in place, create another layer of defense

- Prevent a misbehaving application from harming the rest of the system

# The sandboxing concept

**sand•box**, ′san(d)-"bäks, *noun*. Date: 1688

: a box or receptacle containing loose sand: as **a:** a shaker for sprinkling sand on wet ink **b:** a box that contains sand for children to play in

A restricted area where code can run

# Not just files

## Other resources to protect

- CPU time
- Amount of memory used: physical & virtual
- Disk space
- Network identity & access
  - Each system has an IP address unique to the network
  - A compromised application can exploit address-based access control
    - E.g., log in to remote machines that think you're trusted
  - Intrusion detection systems can get confused

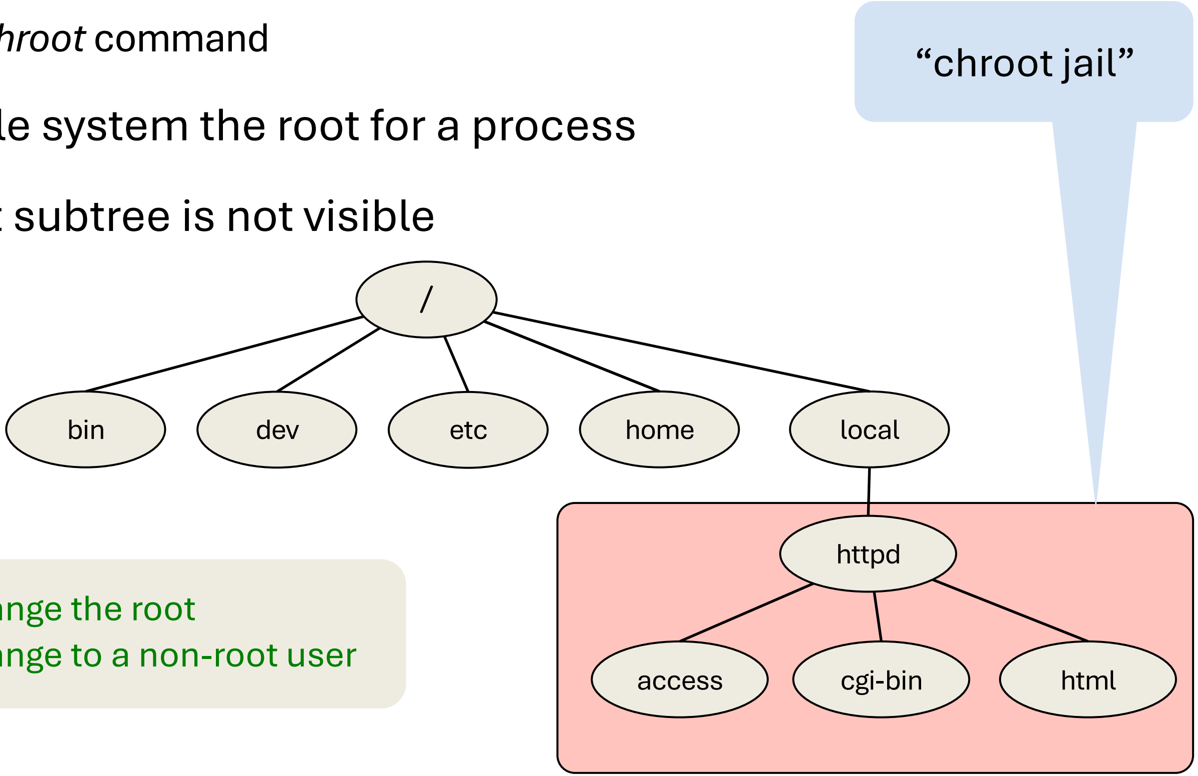
# Application containment goals

- Enforce security – enable a broad set of access restrictions for an application
- High assurance – know it works
- Simple setup – minimize comprehension errors
- General purpose – works with any (most) applications

# Origins: chroot & BSD Jails

# chroot: the granddaddy of containment

- Oldest containment mechanism (Unix v7 – 1982)
  - *chroot* system call and *chroot* command
- Make a subtree of the file system the root for a process
- Anything outside of that subtree is not visible




```
chroot /local/httpd    change the root
su httpuser            change to a non-root user
```

# Problems?

1. Root can escape
2. Requires root access to set up: otherwise an attacker could get root privileges
3. Shared kernel: process has full access to system calls, network stack, devices, processes, ...
4. Resource sharing
   - Fork max # of processes; allocate all memory; create network connections, send signals to processes outside the jail
5. Mount namespaces and filesystems (access full disk)
6. Access files if opened before the chroot

Normal users are not allowed to run chroot because they can get admin privileges

chroot is NOT a security boundary. It's not a security tool. Don't rely on it for isolation

# FreeBSD Jails (2000)

- Enhancement to chroot: added security features chroot lacked
- Run via

  `jail` *jail_path hostname ip_addr command*

Main ideas:

- Confine an application, just like *chroot*
- Network isolation: each jail gets its own IP address
- Process isolation: can't see processes outside the jail
- Restricted root privileges: cannot mount file systems, load modules, access raw devices
- Resource limits: max processes, max open files, memory limits

Pioneered concepts later used in Linux containers – but is a FreeBSD-only solution

# Controlling Access to System Calls

# Restricting System Calls

- Restricting system calls that a process can call provides a level of access control beyond file-system access controls
- For example:
  - Disallow a process from creating other processes
  - Allow a process to open only one specific file
  - Don't allow a process to create network sockets
  - Don't allow a process to create new files
- These controls could be added to a process even if it runs as root

# Restricting system calls

We will look at three ways that system call restrictions can be implemented

1. Via user-level processes or libraries
2. Via kernel-level filtering
3. In an interpreted environment

# 1. Application sandboxing

via system call hooking &
user-level validation

# System Call Interposition

System calls interface with system resources

An application must use system calls to access any resources, initiate attacks ... and cause any damage

- Modify/access files/devices:

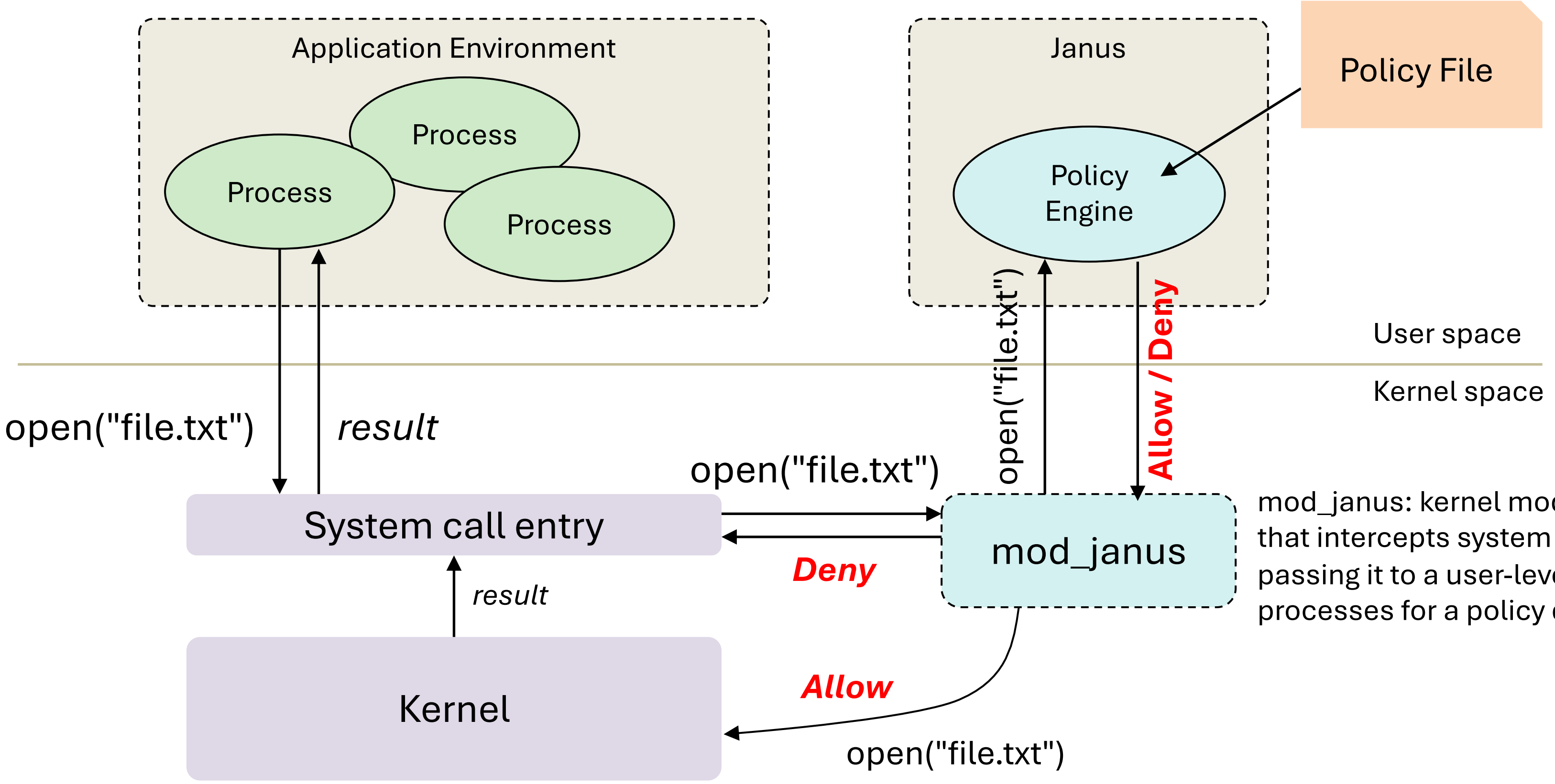
  *creat, open, read, write, unlink, chown, chgrp, chmod, ...*
- Access the network:

  *socket, bind, connect, send, recv*

- Sandboxing via system call interposition
  - Intercept, inspect, and approve an app's system calls
- Various mechanisms:
  - Linux *ptrace* interface, user-level interposition libraries (that wrap system calls), or kernel-level interposition libraries

# Example: Janus – system call interposition

App sandboxing tool implemented as a loadable kernel module




mod_janus: kernel module that intercepts system calls, passing it to a user-level processes for a policy decision

# Implementation Challenge

**Janus must mirror the state of the operating system!**

- If process forks, the Janus monitor must fork
- Keep track of the network protocol
- Does not know if certain operations failed
- Gets tricky if file descriptors are duplicated
- Remember filename parsing?
  Deal with `../` – and track changes to the current directory for relative paths
- App namespace can change if the process does a *chroot*
- What if file descriptors are passed via Unix domain sockets?
- Significant overhead and vulnerable to race conditions: **TOCTTOU**

# Application sandboxing

via integrated OS support

# Linux seccomp-BPF

seccomp-BPF = SECure COMPuting with Berkeley Packet Filters

Allows the user to attach a system call filter to a process and its descendants

Operations per system call: *Allow*, *Deny*, *Trap*

- Seccomp-BPF can also filter simple (integer) parameters to system calls
- Example: allow only specific socked types or modes for files

seccomp-BPF is the primary sandbox in the Chrome browser

See:https://www.chromium.org/chromium-os/developer-library/guides/development/sandboxing/#seccomp-filters

# Linux AppArmor (Application Armor)

**Linux Security Module for Mandatory Access Control via path-based policies**

seccomp-BPF does not allow policies to check string arguments:

- Names of files to open, programs to execute, …

AppArmor adds this

Goal:

Confine programs by defining files & capabilities they can access, regardless of user

# Apple Sandbox

Create a list of rules that is consulted to see if an operation is permitted

Components:

- Set of libraries for initializing/configuring policies per process
- Server for kernel logging
- Kernel extension using the TrustedBSD API for enforcing individual policies
- Kernel support extension providing regular expression matching for policy enforcement

Simplify sandbox setup with pre-written profiles:

- Prohibit TCP/IP networking
- Prohibit all networking
- Prohibit file system writes
- Restrict writes to specific locations (e.g., /var/tmp)
- Perform only computation: minimal OS services

# Application sandboxing

via the language & execution environment

The Java Sandbox

# Java Language

- Type-safe & easy to use
  - Memory management and range checking
- Designed for an interpreted environment: JVM
- No direct access to system calls

# Java Sandbox

1. **Bytecode verifier**: verifies Java bytecode before it is run
    - Disallow pointer arithmetic
    - Automatic garbage collection
    - Array bounds checking
    - Null reference checking
2. **Class loader**: determines if an object is allowed to add classes
    - Ensures key parts of the runtime environment are not overwritten
    - Runtime data areas (stacks, bytecodes, heap) are randomly laid out
3. **Security manager**: enforces *protection domain*
    - Defines the boundaries of the sandbox (file, net, native, etc. access)
    - Consulted before any access to a resource is allowed

# OS-Level Isolation:

## *namespaces, capabilities, control groups*

# Linux Namespaces

- *chroot* only changed the root of the filesystem namespace
- Linux ***namespaces*** provides control over the following namespaces:

| | |
|---|---|
| IPC | System V IPC, POSIX message queues |
| Network | Network devices, stacks, ports |
| Mount | Mount points (including file system root) |
| PID | Process IDs |
| User | User & group IDs |
| UTS | Hostname and NIS domain name |

# Linux Namespaces

Unlike *chroot*, unprivileged users can create namespaces

unshare() – system call that dissociates parts of the process execution context

- Examples
  - Unshare IPC namespace, so it's separate from other processes
  - Unshare PID namespace, so the thread gets its own PID namespace for its children

clone() – system call to create a child process

- Like *fork()* but allows you to control what is shared with the parent
  - Open files, root of the file system, current working directory, IPC namespace, network namespace, memory, etc.

setns() – system call to associate a thread with a namespace

- A thread can associate itself with an existing namespace in /proc/[pid]/ns

# Linux Capabilities

## How do we restrict privileged operations in a namespace?

- UNIX systems distinguished *privileged* vs. *unprivileged* processes
  - Privileged = UID 0 = root ⇒ *kernel bypasses all permission checks*
- With capabilities, privileges are assigned to a process and are *not* based on whether it's running as user ID 0 (root)
- A process running as root can be restricted to limited privileges
  - E.g., no ability to set UID to root, no ability to mount filesystems
- A process running as non-root can be granted limited privileges
  - E.g., ability to send an ICMP packet (ping message)

N.B.: These *capabilities* have nothing to do with *capability lists*

# Linux Capabilities

## Assign subsets of privileges to programs

- Linux divides privileges into 38 distinct controls, including:

| | |
|---|---|
| CAP_CHOWN | make arbitrary changes to file owner and group IDs |
| CAP_DAC_OVERRIDE | bypass read/write/execute checks |
| CAP_KILL | bypass permission checks for sending signals |
| CAP_NET_ADMIN | network management operations |
| CAP_NET_RAW | allow RAW sockets |
| CAP_SETUID | arbitrary manipulation of process UIDs |
| CAP_SYS_CHROOT | enable chroot |

- These are per-thread attributes
  – Can be set via the *prctl* system call

# Linux Capabilities Example

Unprivileged processes cannot bind to network port #s below 1024

Because of this programs that needed to do this (like *ping*) had to run setuid to root.

With capabilities, we can allow the command *`my_program`* to do this without having it run as root

```
sudo setcap 'cap_net_bind_service=+ep' my_program
```

- `cap_bind_service` is the capability to bind to special ports
- `+ep` means:
  - `e`: add the capability to the *Effective* set (what the process can currently do)
  - `p`: add the capability to the *Permitted* set (the maximum capabilities the process is allowed to enable)
  - Without being in the permitted set, a capability can't be used, and without being in the effective set, it isn't currently used.

# Linux Control Groups (cgroups)

Limit the amount of resources a process tree can use

- CPU, memory, block device I/O, network
  - E.g., a process tree can use at most 25% of the CPU
  - Limit # of processes within a group
  - Help with denial-of-service attacks
- Interface = `cgroups` file system: `/sys/fs/cgroup`

Namespaces + cgroups + capabilities = **lightweight process virtualization**

**A group of processes can have the illusion that they are running on their own Linux system, isolated from other processes in the system**

# Vulnerabilities

Bugs have been found

– User namespace: unprivileged user was able to get full privileges

But **comprehension** is a bigger problem

- Namespaces do not prohibit a process from making privileged system calls
  - They control resources that those calls can manage
  - The system will see only the resources that belong to that namespace
- Capabilities grant non-root users increased access to privileged operations
  - Design concept: instead of dropping privileges from root, provide limited elevation to non-root users
- A real root process with its admin capability removed can restore it
  - If it creates a user namespace, the capability is restored to the root user in that namespace – although limited in function

# Containers

# Motivation for containers

- Installing software packages can be a pain
  - Dependencies
- Running multiple packages on one system can be a pain
  - Updating a package can update a library or utility another uses
    - Causing something else to break
  - No isolation among packages
    - Something goes awry in one service impacts another
- Migrating services to another system is a pain
  - Re-deploy & reconfigure

# How did we address these problems?

- Sysadmin effort
  - Service downtime, frustration, redeployment
- Run every service on a separate system
  - Mail server, database, web server, app server, …
  - Expensive!  … and overkill
- Deploy virtual machines
  - Kind of like running services on separate systems
  - Each service gets its own instance of the OS and all supporting software
  - Heavyweight approach – resource intensive
    - Administer multiple machines (keep the OS and services updated)
    - The system must time share between operating systems

# What are containers?

## Containers: created to package & distribute software

- Focus on services, not end-user apps
- Software systems usually require a bunch of stuff:
  - Libraries, multiple applications, configuration tools, …
- Container = image containing the application environment
  - Can be installed and run on any system

Key insight:

*Encapsulate software, configuration, & dependencies into one package*

# A container feels like a virtual machine

- It gives you the illusion of separate
  - Set of apps
  - Process space
  - Network interface
  - Network configuration
  - Libraries, …
- But limited root powers

And … all containers on a system share the same OS & kernel modules

# How are containers built?

- Control groups
  - Meters & limits on resource use
    - Memory, disk (I/O bandwidth), CPU (set %), network (traffic priority)
- Namespaces
  - Isolates what processes can see & access
  - Process IDs, host name, mounted file systems, users, IPC
  - Network interface, routing tables, sockets
- Capabilities
  - Restrict privileges on a per-process basis
- Copy on write file system
  - Instantly create new containers without copying the entire package
  - Storage system tracks changes
- AppArmor
  - Pathname-based mandatory access controls
  - Confines programs to a set of listed files & capabilities

# Docker

- First super-popular container
  - LXC (Linux Containers) were the first
- Designed to provide Platform-as-a-Service capabilities
  - Combined Linux cgroups & namespaces into a single easy-to-use package
  - Enabled applications to be deployed consistently anywhere as one package
- Docker Image
  - Package containing applications & supporting libraries & files
  - Can be deployed on many environments
- Make deployment easy
  - Git-like commands: docker push, docker commit, ...
  - Make it easy to reuse image and track changes
  - Download updates instead of entire images
- Keep Docker images immutable (read-only)
  - Run containers by creating a writable layer to temporarily store runtime changes

# Later Docker additions

- Docker Hub: cloud-based repository for docker images
- Docker Swarm: deploy multiple containers as one abstraction

# Not Just Linux

Microsoft introduced Containers in Windows Server 2016 with support for Docker

- Windows Server Containers
  - Assumes trusted applications
  - Misconfiguration or design flaws may permit an app to escape its container
- Hyper-V Containers
  - Each has its own copy of the Windows kernel & dedicated memory
  - Same level of isolation as in virtual machines
  - Essentially a VM that can be coordinated via Docker
  - Less efficient in startup time & more resource intensive
  - Designed for hostile applications to run on the same host

# Container Orchestration

- We wanted to manage containers across systems
- Multiple efforts
  - Marathon/Apache Mesos (2014), Kubernetes (2015), Nomad, Docker Swarm, …

Google designed Kubernetes for container orchestration

- Handle multiple containers and start each one at the right time
- Handle storage
- Deal with hardware and container failure: automatic start & migration
- Integrates with the Docker engine
- Scale rapidly by adding/removing containers based on demand (e.g., Pokemon Go)
- Open source

# Not build for security – containers have security benefits

- Containers use namespaces, control groups, & capabilities
  - Restricted capabilities by default
  - Isolation among containers
- Containers are usually minimal and application-specific
  - Just a few processes
  - Minimal software & libraries
  - Fewer things to attack
- They separate policy from enforcement
- Execution environments are reproducible
  - Easy to inspect how a container is defined
  - Can be tested in multiple environments
- Watchdog-based re-starting: helps with availability
- Containers help with comprehension errors
  - Decent default security without learning much
  - Also ability to enable other security modules

# Security Concerns

- Kernel exploits: All containers share the same kernel
- Privileges & escaping the container: Privileged vs. unprivileged containers
- Users in multiple containers may share the same real ID
- Denial of service attacks
- Network spoofing
- Origin integrity
  - Where is the container from and has it been tampered?

# Containment via Virtual Machines

# Machine Virtualization

- Normally all hardware and I/O managed by one operating system
- Machine virtualization
  - Abstract (virtualize) control of hardware and I/O from the OS
  - Partition a physical computer to act like several computers
    - Manipulate memory mappings
    - Set system timers
    - Access devices
  - Migrate an entire OS & its applications from one computer to another
- 1972: VMs released with IBM System 370
  - Originally created to allow kernel developers to share a computer

# What made VMs popular?

- Wasteful to dedicate a computer to each service
  - Mail, print server, web server, file server, database
- If these services run on a separate computer
  - Configure the OS just for that service
  - Attacks and privilege escalation won’t hurt other services

# The Hypervisor

Hypervisor: Program in charge of virtualization

- Also called the Virtual Machine Monitor
- Provides the illusion that the OS has full access to the hardware
- Arbitrates access to physical resources
- Presents a set of virtual device interfaces to each host

# Machine Virtualization

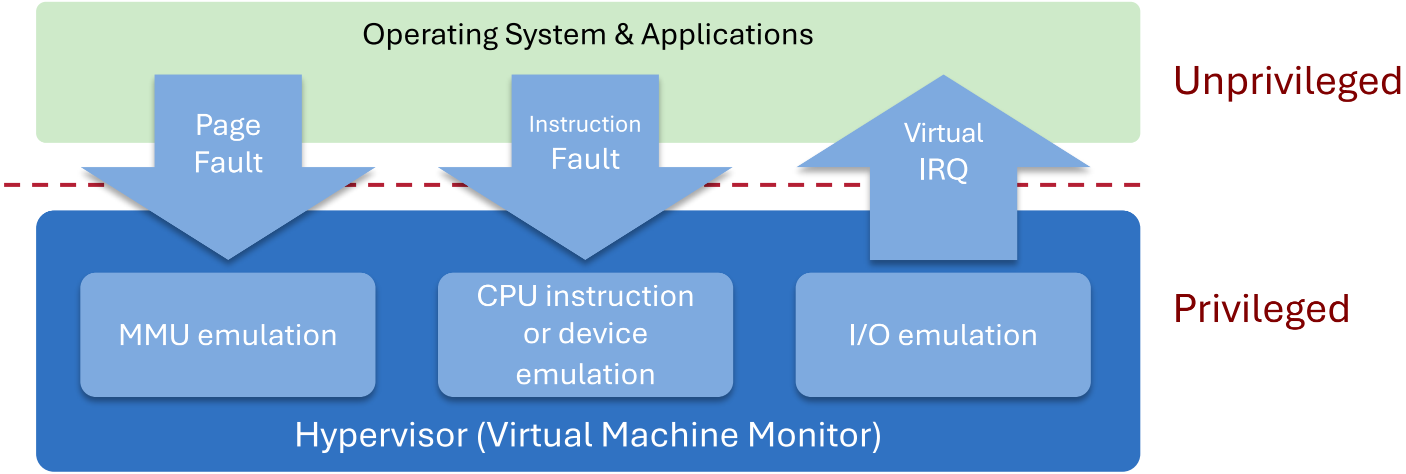
## An OS is just a bunch of code!

- Privileged vs. unprivileged instructions
  - If regular applications execute privileged instructions, they trap
  - Operating systems are allowed to execute privileged instructions
- With machine virtualization
  - We deprivilege the operating system
  - The VMM runs at a higher privilege level than the OS
- The VMM catches the trap
  - If it turns out that the attempt to execute the privileged instruction occurred in the kernel code, the hypervisor (VMM) emulates the instruction
  - **Trap & Emulate**

# Hypervisor = Virtual Machine Monitor = VMM

Application or Guest OS runs until:

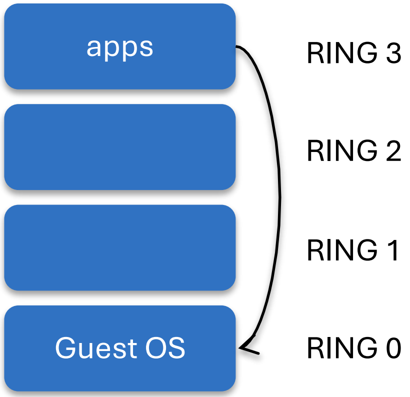
- Privileged instruction traps
- System interrupts
- Exceptions (page faults)
- Explicit call: `VMCALL` (Intel) or `VMMCALL` (AMD)



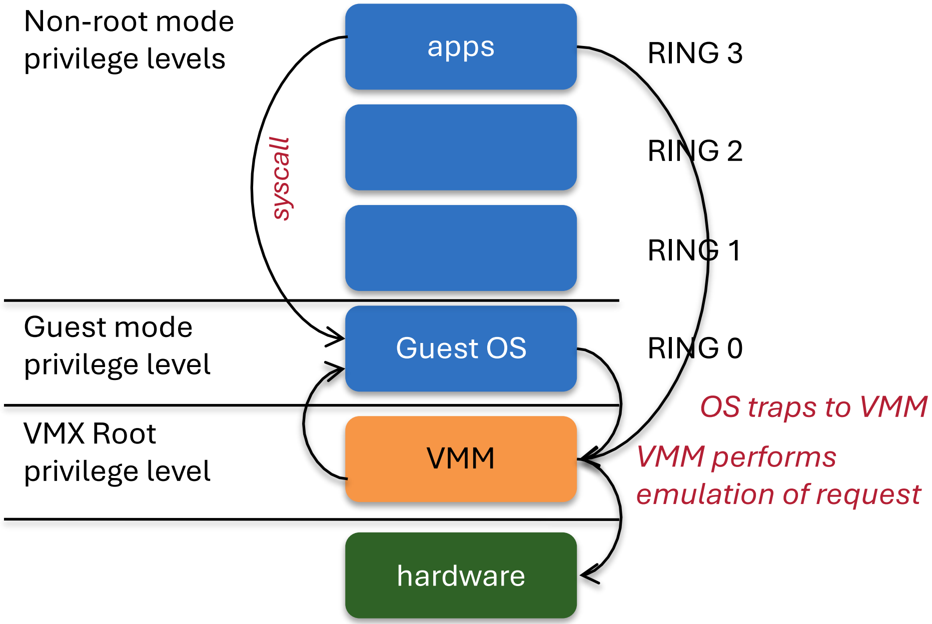
# Hardware support for virtualization

Root mode (Intel example)

– Layer of execution more privileged than the kernel




Without virtualization



# Architectural Support

- Intel Virtual Technology, AMD-V
- ARM Virtualization Extensions
  - New mode (HYP) and new privilege level (non-secure privilege level 2)

---

**Guest mode execution**: can run privileged instructions directly

- E.g., a system call does not need to go to the VM
- Certain privileged instructions are intercepted as VM exits to the VMM
- Exceptions, faults, and external interrupts are intercepted as VM exits
- Virtualized exceptions/faults are injected as VM entries

# CPU Architectural Support

- Setup
  - Turn VM support on/off (usually in BIOS)
  - Configure what controls VM exits
  - Processor state: saved & restored in guest & host areas
- VM Entry: go from hypervisor to VM
  - Load state from the guest OS area
- VM Exit
  - VM-exit: like a trap – information contains the cause of the exit
  - Processor state saved in guest area
  - Processor state loaded from host area

# Two Approaches to Running VMs

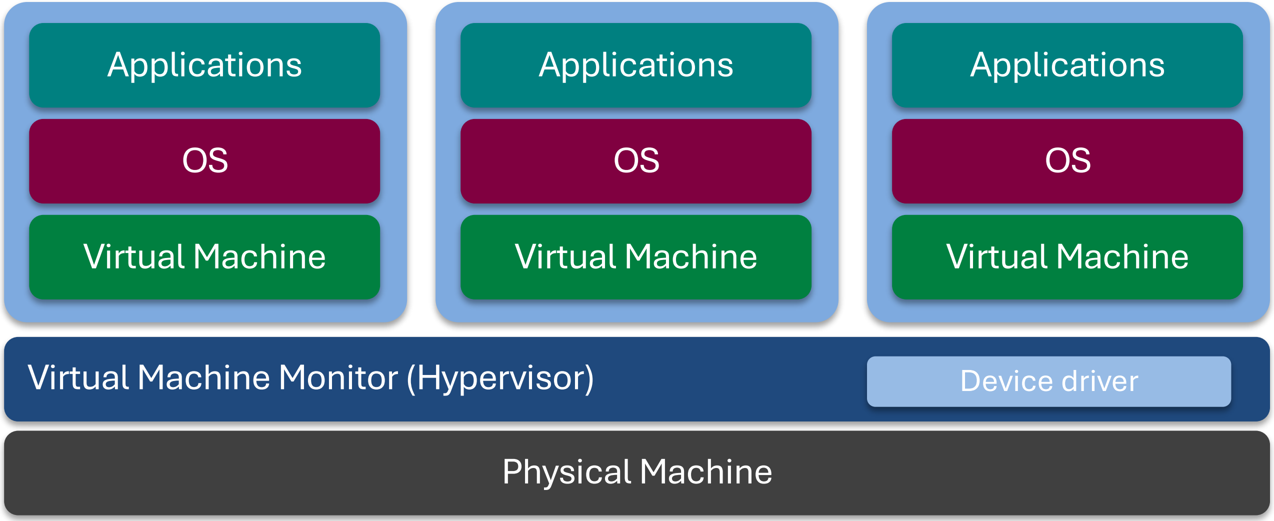
1. Type 1: Bare Metal – Native VM (hypervisor model)
2. Type 2: Hosted VM

# Type 1: Native Virtual Machine (Bare Metal)

Example:
VMware ESX

## Native VM (or *Type 1* or *Bare Metal*)

- No primary OS
- Hypervisor is in charge of access to the devices and scheduling
- OS runs in “kernel mode” but does not run with full privileges

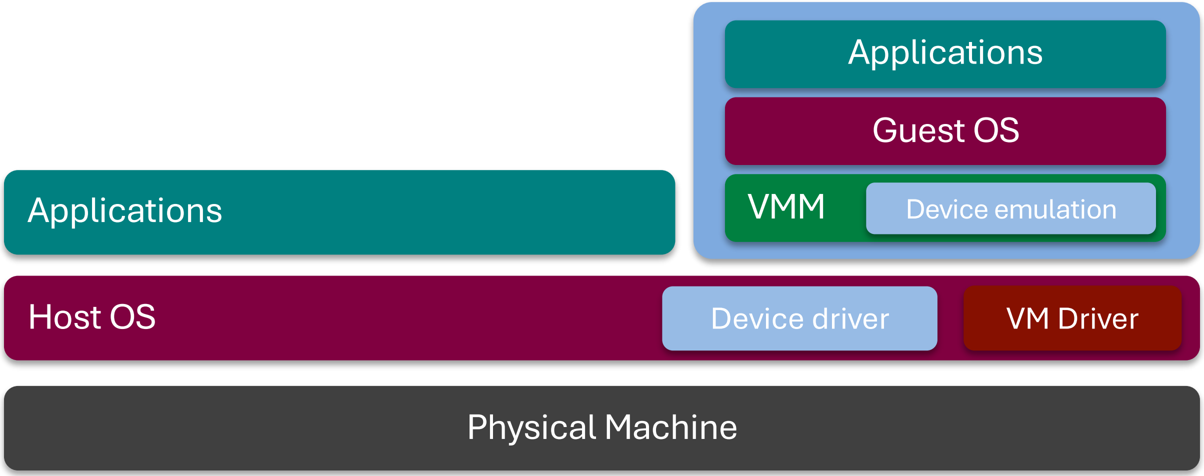


# Type 2: Hosted Hypervisor

## Hosted VM

- VMM runs without special privileges
- Primary OS responsible for access to the raw machine
- Guest operating systems run under a VMM
- VMM invoked by host OS

Example: VMware Workstation



# Security Benefits

- Virtual machines provide isolation of operating systems
- Attacks & malware can target the guest OS & apps
- Malware cannot escape from the infected guest OS
- Recovery from snapshots
- Easy to replicate virtual machines
- Operate as a test environment

# Risks

- Same as with introducing other new computers
  - Poorly configured access policies
  - Untrusted or unpatched software
  - "Default" system installations (e.g., full Linux distributions)
- An attacker may enable virtualization
  … and install a new virtual machine in a computing environment
  - It acts like a real computer
  - Private file system
  - Undetected by other VMs
  - Admins might not notice one more system on the network

# The End