Computer Security

05. Confinement

Paul Krzyzanowski

Rutgers University

Spring 2019

Confinement: prepare for the worst

- We realize that an application may be compromised
  - We want to run applications we may not completely trust
- Not always possible
- Limit an application to use a subset of the system's resources
- Make sure a misbehaving application cannot harm the rest of the system

How about access control?

- Limit damage via access control
  - E.g., run servers as a low-privilege user
  - Proper read/write/search controls on files … or role-based policies
- ACLs don't address applications
  - Cannot set permissions for a process: “don’t allow access to anything else”
  - At the mercy of default (other) permissions
- We are responsible for changing protections of every file on the system that could be accessed by *other*
  - And hope users don’t change that
  - Or use more complex mandatory access control mechanisms … if available

Not high assurance

Compromised applications

- Some services run as root
- What if an attacker compromises the app and gets root access?
  - Create a new account
  - Install new programs
  - “Patch” existing programs (e.g., add back doors)
  - Modify configuration files or services
  - Add new startup scripts (launch agents, cron jobs, etc.)
  - Change resource limits
  - Change file permissions (or ignore them!)
  - Change the IP address of the system

We can regulate access to some resources

POSIX `setrlimit()` system call

- Maximum CPU time that can be used
- Maximum data size
- Maximum files that can be created
- Maximum memory a process can lock
- Maximum # of open files
- Maximum # of processes for a user
- Maximum amount of physical memory used
- Maximum stack size

Other resources to protect

- CPU time
- Amount of memory used: physical & virtual
- Disk space
- Network identity & access

Network identity

- Each system has an IP address unique to the network
- 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

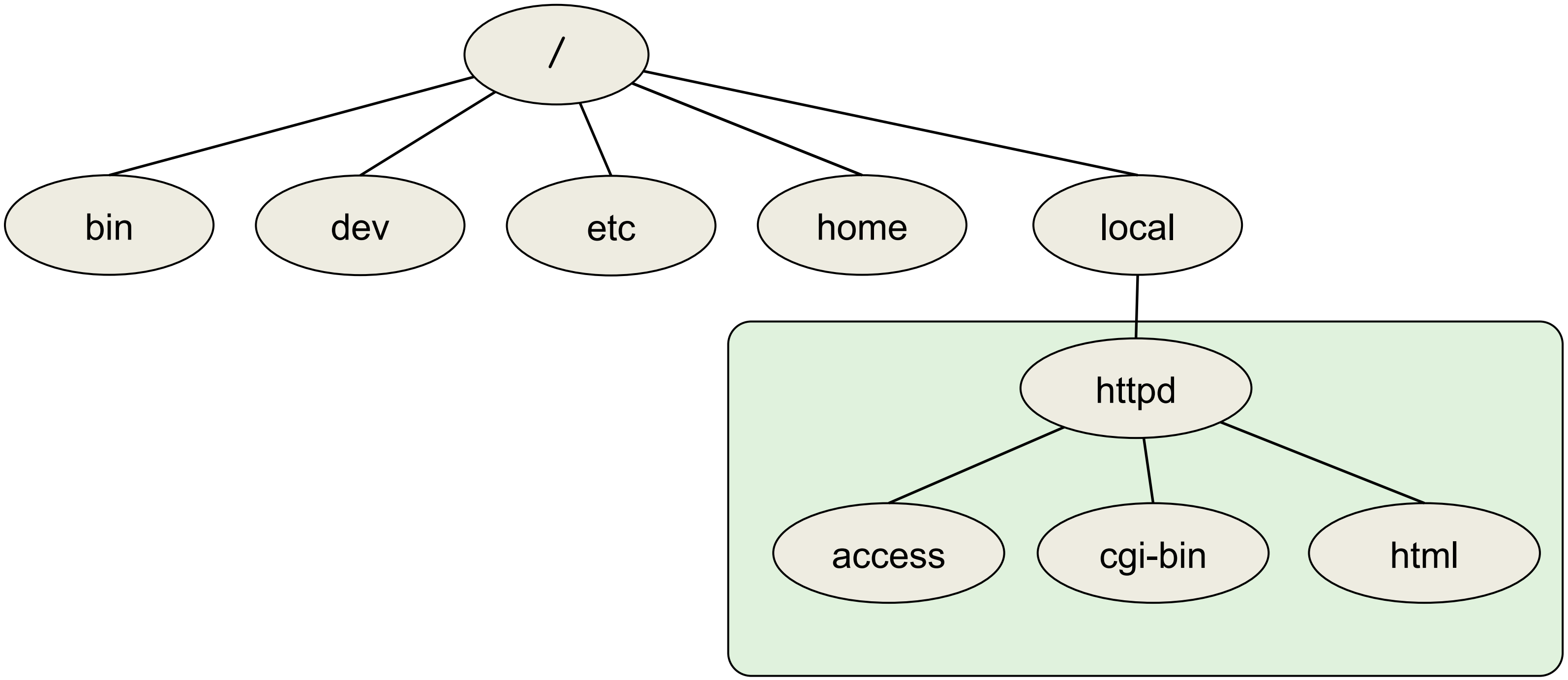
Application confinement goals

- Enforce security – broad access restrictions
- High assurance – know it works
- Simple setup – minimize comprehension errors
- General purpose – works with any (most) applications

We don't get all of this …

chroot: the granddaddy of confinement

- Oldest confinement mechanism
- Make a subtree of the file system the root for a process
- Anything outside of that subtree doesn't exist



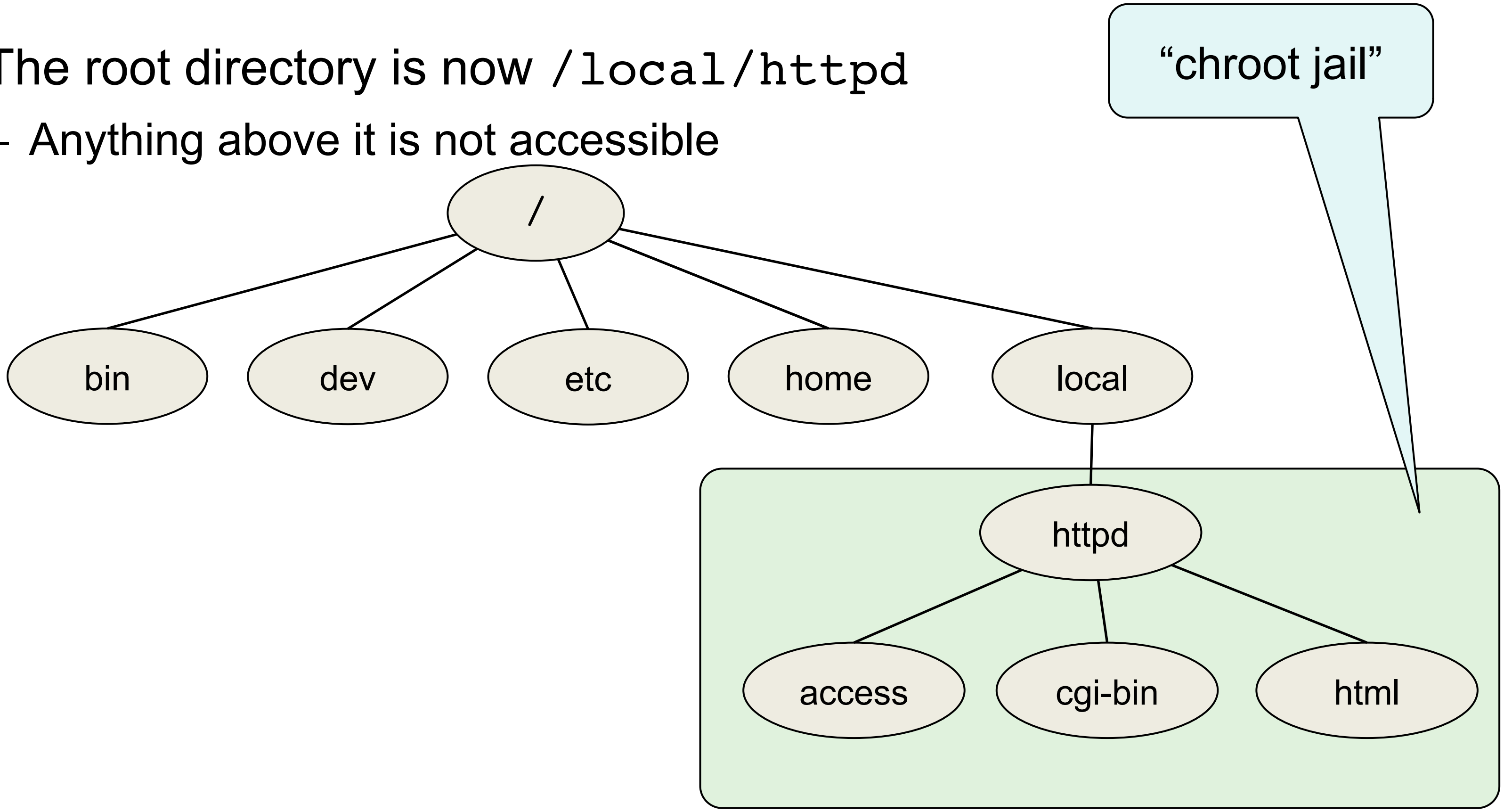
chroot: the granddaddy of confinement

- Only root can run *chroot*

  `chroot /local/httpd` *change the root*

  `su httpuser` *change to a non-root user*

- The root directory is now `/local/httpd`
  - Anything above it is not accessible



Jailkits

- If programs within the jail need any utilities, they won't be visible
  - They're outside the jail
  - Need to be copied
  - Ditto for shared libraries

- Jailkit (https://olivier.sessink.nl/jailkit/)
  - Set of utilities that build a chroot jail
  - Automatically assembles a collection of directories, files, & libraries
  - Place the **bare minimum** set of supporting commands & libraries
    - The fewer executables live in a jail, the less tools an attacker will have to use
  - Contents
    - jk_init: create a jail using a predefined configuration
    - jk_cp: copy files or devices into a jail
    - jk_chrootsh: places a user into a chroot jail upon login
    - jk_lsh: limited shell that allows the execution only of commands in its config file
    - …

https://olivier.sessink.nl/jailkit/

Problems?

- Does not limit network access
- Does not protect network identity
- Applications are still vulnerable to root compromise
- *chroot* must be available only to root

  If not…
  - Create a jail directory `mkdir /tmp/jail`
  - Create a link to the su command `ln /bin/su /tmp/jail/su`
  - Copy or link libraries & shell …
  - Create an /etc directory `mkdir /tmp/jail/etc`
  - Create password file(s) with a known password for root
  - Enter the jail `chroot /tmp/jail`
  - `su root` – su will validate against the password file in the jail!

Escaping a chroot jail

If you can become root in a jail, you have access to *all* system calls

Example: create a device file for the disk

- On Linux/Unix/BSD, all non-network devices have filenames
- Even memory has a filename (/dev/mem)

- Create a memory device (*mknod* system call)
  - Change kernel data structures to remove your jail
- Create a disk device to access your raw disk
  - Mount it within your jail and you have access to the whole file system
  - Get what you want, change the admin password, …
- Send signals to kill other processes
  (doesn't escape the jail but causes harm to others)
- Reboot the system

chroot summary

- Good confinement
- Imperfect solution
- Useless against root
- Setting up a working environment takes some work (or use jailkit)

FreeBSD Jails

- Enhancement to chroot
- Run via

  `jail` *jail_path hostname ip_addr command*

- Main ideas:
  - Confine an application, just like *chroot*
  - Restrict what operations a process within a jail can perform, even if root

https://www.freebsd.org/doc/en/books/arch-handbook/jail.html

FreeBSD Jails: Differences from chroot

- Network restrictions
  - Jail has its own IP address
  - Can only bind to sockets with a specified IP address and authorized ports
- Processes can only communicate with processes inside the jail
  - No visibility into unjailed processes
- Hierarchical: create jails within jails
- Root power is limited
  - Cannot load kernel modules
  - Ability to disallow certain system calls
    - Raw sockets
    - Device creation
    - Modifying network configuration
    - Mounting/unmounting file systems
    - set_hostname

https://www.freebsd.org/doc/en/books/arch-handbook/jail.html

Problems

- Coarse policies
  - All or nothing access to parts of the file system
  - Does not work for apps like a web browser
    - Needs access to files outside the jail (e.g., saving files, uploading attachments)
- Does not prevent malicious apps from
  - Accessing the network & other machines
  - Trying to crash the host OS
- BSD Jails is a BSD-only solution
- Pretty good for running things like DNS servers and web servers
- Not all that useful for user applications

Linux Namespaces

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

IPC	System V IPC, POSIX message queues	Objects created in an IPC namespace are visible to all other processes *only* in that namespace
Network	Network devices, stacks, ports	Isolates IP protocol stacks, IP routing tables, firewalls, socket port #s
Mount	Mount points	Mount points can be different in different processes
PID	Process IDs	Different PID namespaces can have the same PID – child cannot see parent processes or other namespaces
User	User & group IDs	Per-namespace user/group IDs. You can be root in a namespace with restricted privileges
UTS	Hostname and NIS domain name	*sethostname* and *setdomainname* affect only the namespace

See namespaces(7)

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 what *root* can do in a namespace?

- UNIX systems distinguished *privileged* vs. *unprivileged* processes
  - Privileged = UID 0 = root ⇒ *kernel bypasses all permission checks*
- If we can provide limited elevation of privileges to a process:
  - If a process becomes root, it would still be limited in what it could do
  - E.g., no ability to set UID to root, no ability to mount filesystems

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

Linux Capabilities

We can explicitly grant subsets of privileges that root users get

- 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 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
- Interface = `cgroup` file system: `/sys/fs/cgroup`

Namespaces + cgroups + capabilities = lightweight process virtualization

- Process gets the *illusion* that it is running on its own Linux system, isolated from other processes

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
  - User namespaces grant non-root users increased access to system capabilities
    - 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

Summary

- *chroot*
- FreeBSD Jails
- Linux namespaces, capabilities, and control groups
  - Control groups
    - Allow processes to be grouped together – control resources for the group
  - Capabilities
    - Limit what root can do for a process & its children
  - Namespaces
    - Restrict what a process can see & who it can interact with:
      PIDs, User IDs, mount points, IPC, network

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
    - 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

- 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**
  - Keep root ID but enumerate what it is allowed to do
- **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

Initially … Docker

- First super-popular container
- 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 and 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
  - Google invented Linux control groups
  - Standard deployment interface
  - Scale rapidly (e.g., Pokemon Go)
  - Open source (unlike Docker Swarm)

Container Orchestration

Kubernetes orchestration

- Handle multiple containers and start each one at the right time
- Handle storage
- Deal with hardware and container failure
  - Automatic restart & migration
- Add or remove containers in response to demand
- Integrates with the Docker engine, which runs the actual container

Containers & Security

Primary goal was software distribution, not security

- Makes moving & running a collection of software simple
  - E.g., Docker Container Format
- Everything at Google is deployed & runs in a container
  - Over 2 billion containers started per week (2014)
  - lmctfy ("*Let Me Contain That For You*")
    - Google's old container tool – similar to Docker and LXC (Linux Containers)
  - Then Kubernetes to manage multiple containers & their storage

Containers & Security

But there are 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 restarting**: helps with availability
- Containers help with **comprehension errors**
  - Decent default security without learning much
  - Also ability to enable other security modules

Some things to watch out for

- Privileges & escaping the container
  - Privileged containers map uid 0 to the host's uid 0

    Prevention of escape is based on MAC (apparmor), capabilities & namespace configuration
  - Unprivileged containers map uid 0 to an unprivileged user outside the container

    No possibility of root escalation

- DoS attacks possible
  - Untrusted users may launch attacks within containers
  - Cgroup limits are often not configured

- Users in multiple containers may share the same real ID
  - If users map to the same parent ID, they share all the limits of that ID
  - A user in one container can perform a DoS attack on another user

- Network spoofing
  - A container can transmit raw ethernet packets and spoof any service

Security Concerns

- **Kernel exploits**
  - All containers share the same kernel
- **Denial of service attacks**
  - If one container can monopolize a resource, others suffer
- **Privilege escalation**
  - Shouldn't happen with capabilities ... But there might be bugs
- **Origin integrity**
  - Where is the container from and has it been tampered?

Machine Virtualization

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 real machines
  - Manipulate memory mappings
  - Set system timers
  - Access devices
- Migrate an entire OS & its applications from one machine to another

1972: IBM System 370

- Allow kernel developers to share a computer

Why are 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

Hypervisor

Hypervisor: Program in charge of virtualization

- Aka 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!

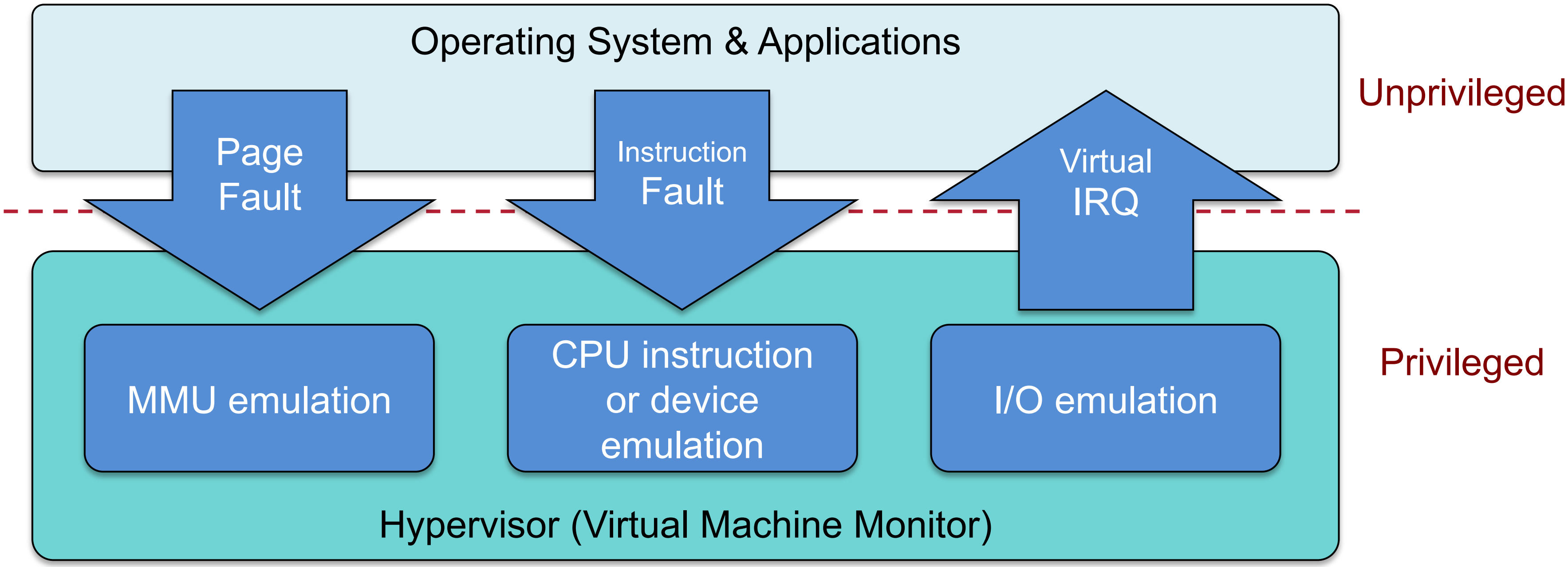
- Privileged vs. unprivileged instructions

- If regular applications execute privileged instructions, they trap
- Operating systems are allowed to execute privileged instructions
- If running kernel code, the VMM catches the trap and emulates the instruction
  – Trap & Emulate

Hypervisor

Application or Guest OS runs until:

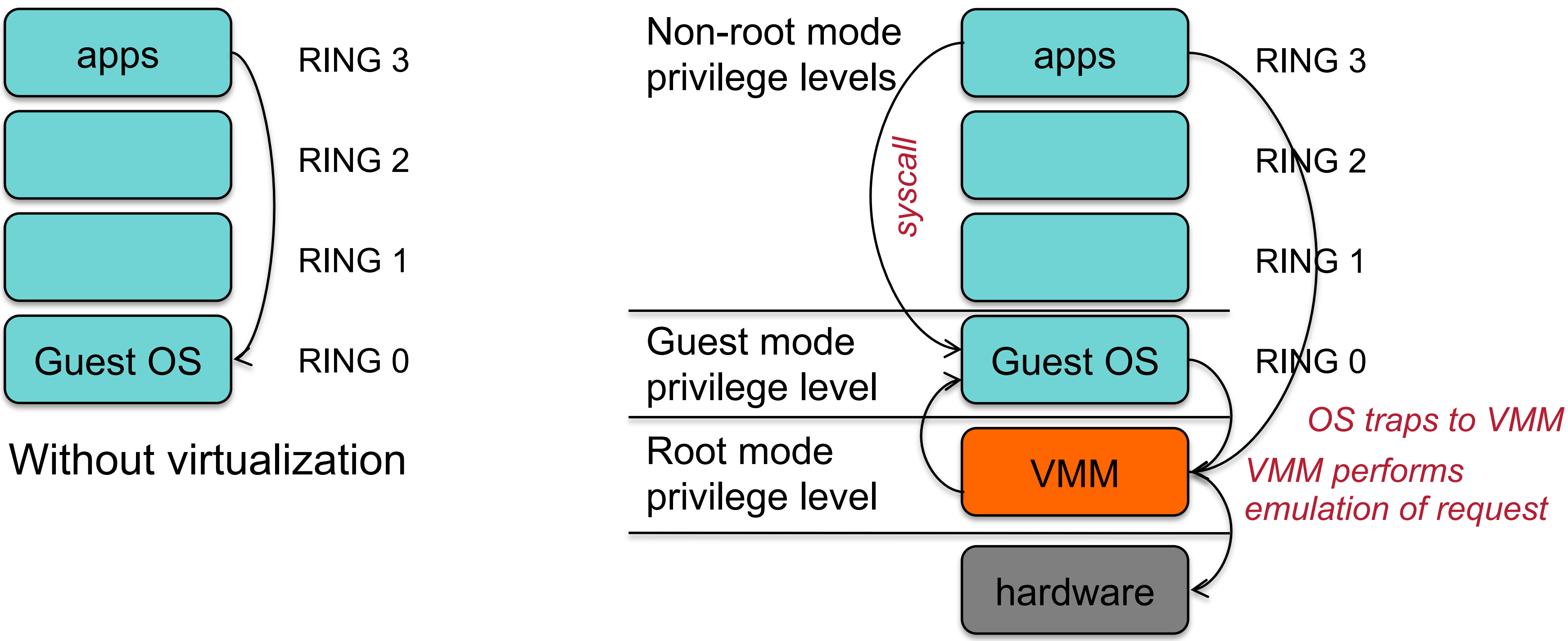
– Privileged instruction traps
– System interrupts
– Exceptions (page faults)
– Explicit call: VMCALL (Intel) or VMMCALL (AMD)



Hardware support for virtualization

Root mode (Intel example)

– Layer of execution more privileged than the kernel



Architectural Support

- Intel Virtual Technology
- AMD Opteron

Guest mode execution: can run privileged instructions directly

- E.g., a system call does not need to go to the VM
- <u>Certain privileged instructions</u> 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
  - Configure what controls VM exits
  - Processor state
    - Saved & restored in guest & host areas
- VM Entry: go from hypervisor to VM
  - Load state from guest area
- VM Exit
  - VM-exit information contains cause of exit
  - Processor state saved in guest area
  - Processor state loaded from host area

Two Approaches to Running VMs

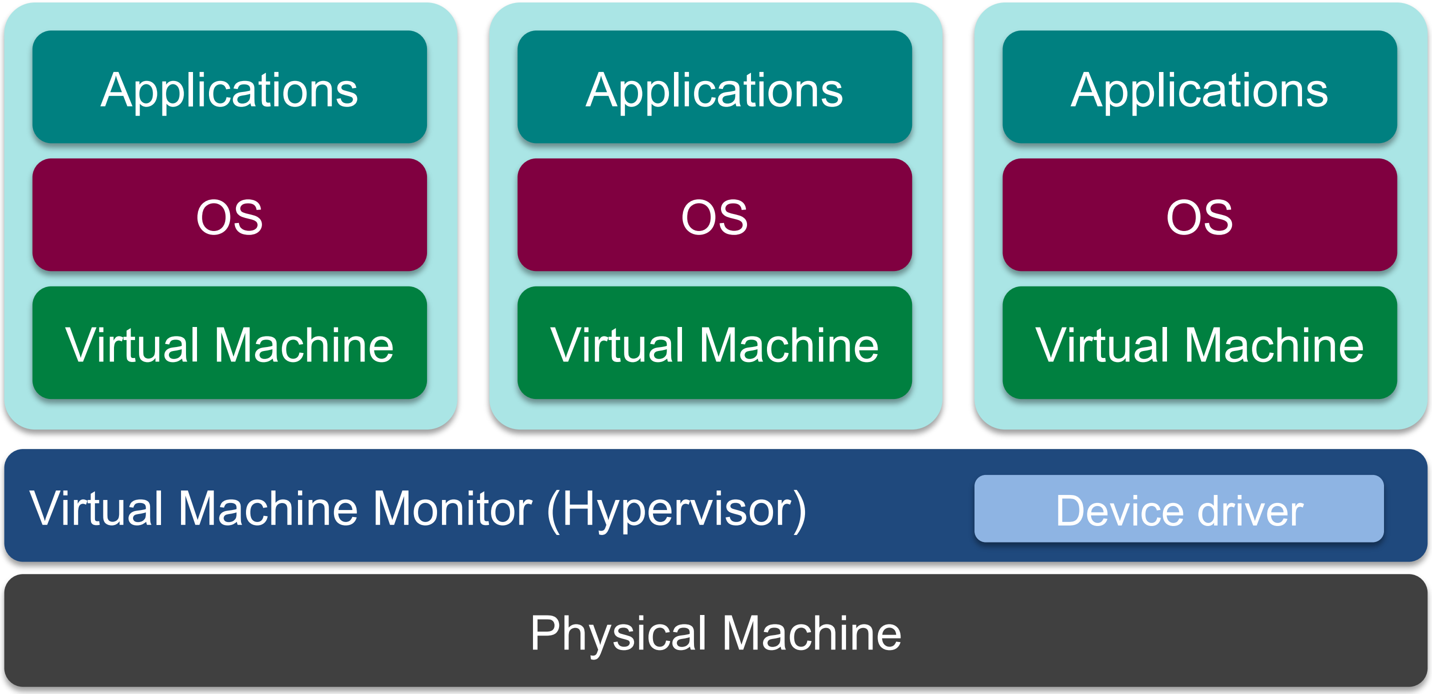
1. Native VM (hypervisor model)
2. Hosted VM

Native Virtual Machine

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

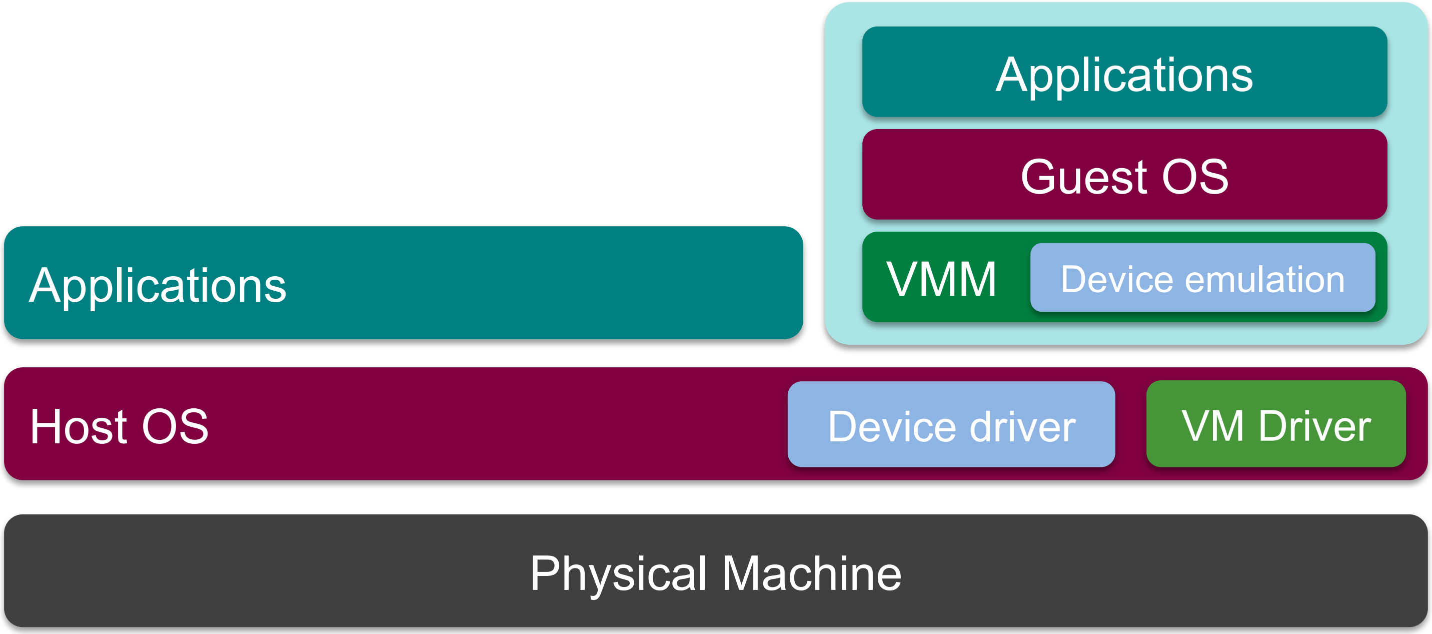


Hosted Virtual Machine

Example: VMware Workstation

Hosted VM

- VMM runs without special privileges
- Primary OS responsible for access to the raw machine
  - Lets you use all the drivers available for that primary OS
- Guest operating systems run under a VMM
- VMM invoked by host OS
  - Serves as a proxy to the host OS for access to devices



Security Benefits

- Virtual machines provide isolation of operating systems
- Attacks & malware can target the guest OS & apps
- Malware cannot escape from the infected VM
  - If a guest OS is compromised or fails
    - the host and other OSes are unaffected
    - The ability of other OSes to access resources is unaffected
    - The performance of other OSes is unaffected
  - Cannot infect the host OS
  - Cannot infect the VMM
  - Cannot infect other VMs on the same computer

Security Benefits

- Recovery from snapshots
  - Easy to revert to a previous version of the system
- Easy to replicate virtual machines
  - Treat the system as a virtual “appliance”
  - If it gets infected with malware, just start another appliance
- Operate as a test environment
  - Great for testing suspicious software
  - See what files have been modified
  - Compare before/after states
  - Restore to pre-installed state

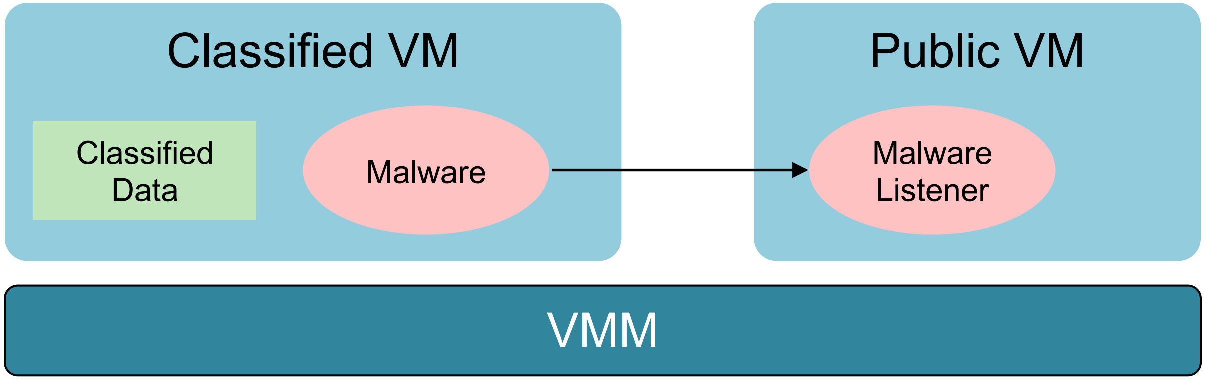
Covert Channels

Covert channel

– Secret communication channel between components that are not allowed to communicate

Side channel attack

– Communication using some aspect of a system's behavior



1. Malware can perform CPU-intensive task at specific times
2. Listener can do CPU-intensive tasks and measure completion times

This allows malware to send a bit pattern:

malware working = 1 = slowdown on listener

Depends on scheduler but there are other mechanisms too… like memory access

Sandboxes

Untrusted applications

- Jail / container / VM solutions
  - Great for running services
- Not really useful for applications
  - These need to be launched by users & interact with their environment

The sandbox

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:** <u>a box that contains sand for children to play in</u>

- A restricted area where code can play in
- Allow users to download and execute untrusted applications with limited risk
- Restrictions can be placed on what an application is allowed to do in its sandbox
- Untrusted applications can execute in a trusted environment

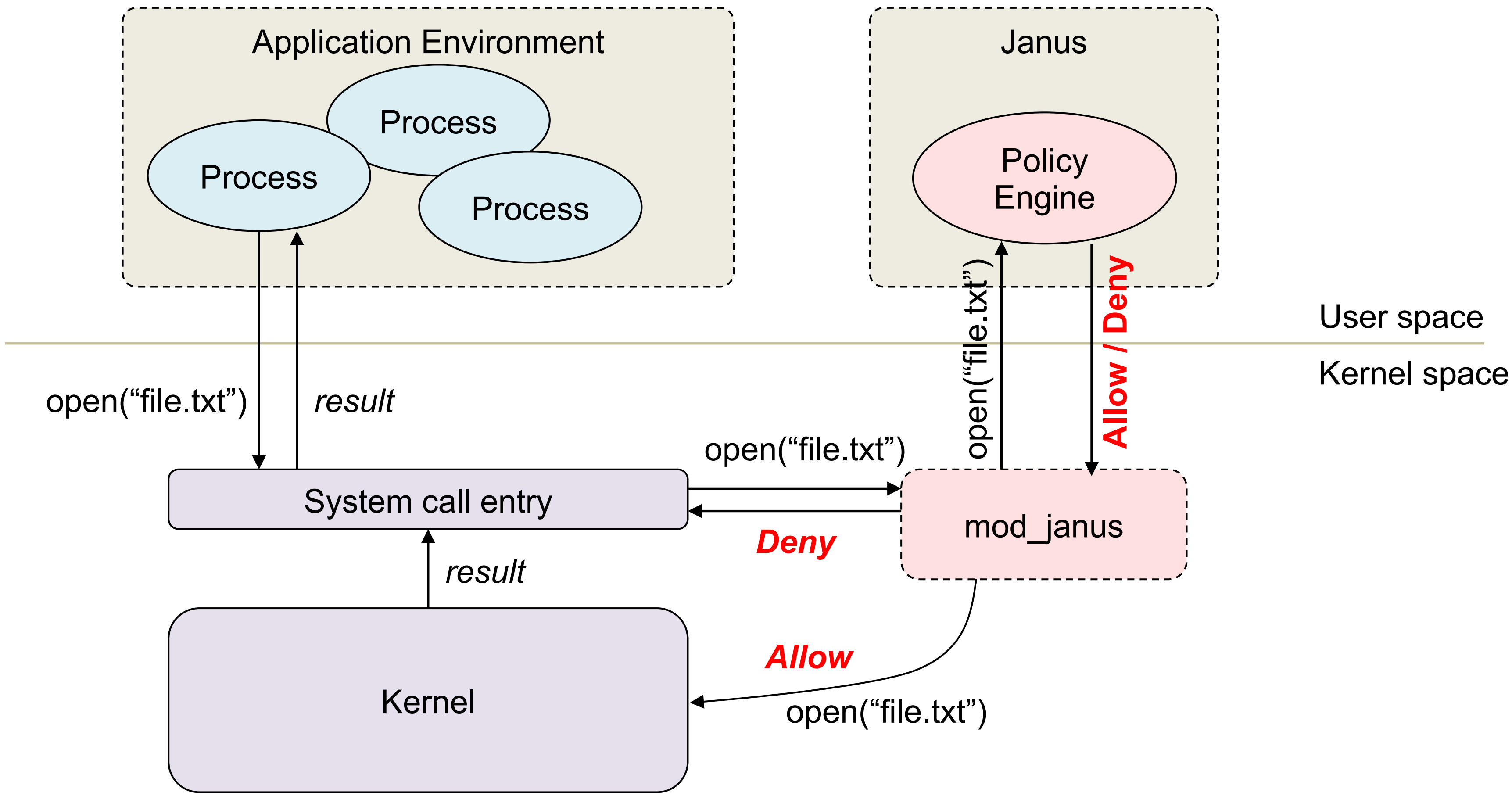
Jails & containers are a form of sandboxing
… but we want to focus on giving users the ability to run apps

System Call Interposition

- System calls interface with 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*
- **Interposition**
  - Intercept & inspect an app's system calls

Example: Janus

App sandboxing tool implemented as a loadable kernel module



Example: Janus

- Policy file defines allowable files and network operations
- Dedicated policy per process
  - Policy engine reads policy file
  - Forks
  - Child process *execs* application
  - All accesses to resources are screened by Janus
- System call entry points contain *hooks*
  - Redirect control to `mod_Janus`
  - Module tells the user-level Janus process that a system call has been requested
    - Process is blocked
    - Janus process queries the module for details about the call
    - Makes a policy decision

Implementation Challenge

Janus has to mirror the state of the operating system!

- If process forks, the Janus monitor must fork
- Keep track of the network protocol
  - socket, bind, connect, read/write, shutdown
- Does not know if certain operations failed
- Gets tricky if file descriptors are duplicated
- Remember filename parsing?
  - We have to figure out the whole dot-dot (..) thing!
  - Have to keep track of changes to the current directory too
- App namespace can change if the process does a *chroot*
- What if file descriptors are passed via Unix domain sockets?
  - *sendmsg, recvmsg*
- Race conditions: **TOCTTOU**

Web plug-ins

- External binaries that add capabilities to a browser
- Loaded when content for them is embedded in a page
- Examples: Adobe Flash, Adobe Reader, Java

Chromium Native Client (NaCl)

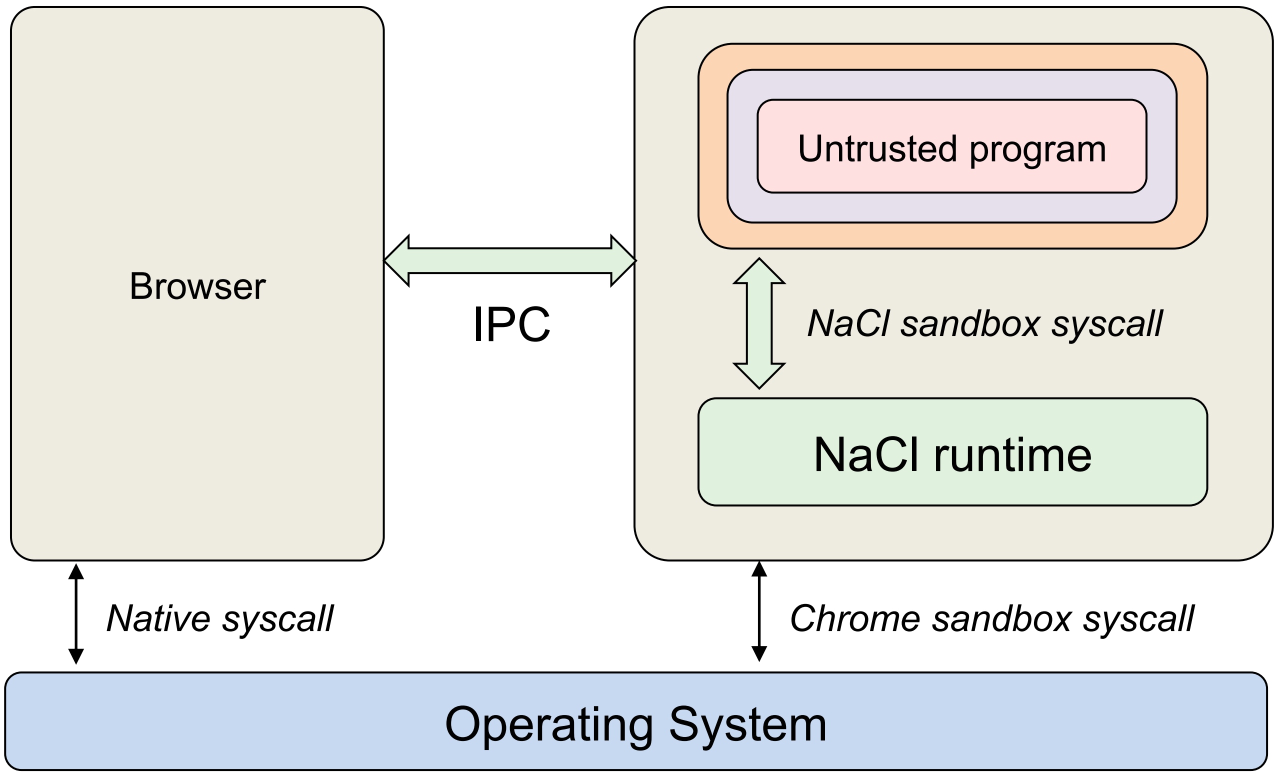
chrome

- Designed for
  - Safe execution of platform-independent untrusted native code in a browser
  - Compute-intensive applications
  - Interactive applications that use resources of a client
- Two types of code: **trusted & untrusted**
  - **Untrusted** has to run in a sandbox
  - Pepper Plugin API (PPAPI): portability for 2D/3D graphics & audio
- Untrusted native code
  - Built using NaCl SDK or any compiler that follows alignment rules and instruction restrictions
    - GNU-based toolchain, custom versions of gcc/binutils/gdb, libraries
    - 32-bit x86 support
  - NaCl statically verifies the code to check for use of privileged instructions

Chromium Native Client (NaCl)

Two sandboxes

- Outer sandbox: restricts capabilities using system call interposition
- Inner sandbox: uses x86 segmentation to isolate memory among apps



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

JVM Security

- Complex process
- ~20 years of bugs … hope the big ones have been found!
- Buffer overflows found in the C support library
  - C support library buggy in general
- Generally, the JVM is considered insecure
  - But Java in general is pretty secure
    - Array bounds checking, memory management
    - Security manager with access controls
  - Use of native methods allows you to bypass security checks

OS-Level Sandboxes

Example: the 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

- sandbox-exec command & sandbox_init function
  - sandbox-exec: calls *sandbox_init()* before *fork()* and *exec()*
  - `sandbox_init(kSBXProfileNoWrite, SANDBOX_NAMED, errbuf);`

Apple sandbox setup & operation

sandbox_init:

– Convert human-readable policies into a binary format for the kernel
– Policies passed to the kernel to the TrustedBSD subsystem
– TrustedBSD subsystem passes rules to the kernel extension
– Kernel extension installs sandbox profile rules for the current process

Operation: intercept system calls

– System calls hooked by the TrustedBSD layer will pass through **`Sandbox.kext`** for policy enforcement
– The extension will consult the list of rules for the current process
– Some rules require pattern matching (e.g., filename pattern)

Apple sandbox policies

Some 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

Virtual Machines

Virtual CPUs (sort of)

What time-sharing operating systems give us

- Each process feels like it has its own CPU & memory
  - But cannot execute privileged CPU instructions (e.g., modify the MMU or the interval timer, halt the processor, access I/O)
- Illusion created by OS preemption, scheduler, and MMU
- User software has to “ask the OS” to do system-related functions
- Containers, BSD Jails, namespaces give us operating system-level virtualization

Process Virtual Machines

CPU interpreter running as a process

- Pseudo-machine with interpreted instructions
  - 1966: O-code for BCPL
  - 1973: P-code for Pascal
  - 1995: Java Virtual Machine (JIT compilation added)
  - 2002: Microsoft .NET CLR (pre-compilation)
  - 2003: QEMU (dynamic binary translation)
  - 2008: Dalvik VM for Android
  - 2014: Android Runtime (ART) – ahead of time compilation
- Advantage: run anywhere, sandboxing capability
- No ability to even pretend to access the system hardware
  - Just function calls to access system functions
  - Or “generic” hardware

The end