CS 489 / 698: Software and Systems Security

Module 3: Operating System Security
compartmentalization / sandboxing

Meng Xu *(University of Waterloo)*

Spring 2023

Outline

OS security: what

Q: What security an operating system needs to provide?

Q: What does an operating system do?

A: Resource sharing — An operating system (OS) allows different "entities" to access different resources in a shared way.

- OS makes resources available to entities **if** required by them and **when** permitted by some policy (and availability).
  - What is a resource?
  - What is an entity?
  - How does an entity request for a resource?
  - How does a policy gets specified?
  - How is the policy enforced?

OS security: why

Q: Why do we need the OS to provide these security features?

A: Because entities DO NOT trust each other.

Q: What if I am the only user on this OS (e.g., my phone)?

A: Even for a single-user OS, protecting the user is a good thing because of user mistakes, malware, or resource over-consumption.

Outline

Memory isolation

Goal: Prevent one program from corrupting other running programs, the operating system, and maybe itself.

Q: How to achieve this?

A: Virtual memory

Memory isolation

- The OS provides a separate, private, and virtual address space for each process — hence the name virtual memory.
- The virtual memory of a process holds the code, data, stack, and heap for the program that is running in that process.
- The running process see only virtual addresses, e.g.,
  - program counter and stack pointer hold virtual addresses of the next instruction and the stack
  - pointers to variables are virtual addresses
  - jump/call instructions refer to virtual addresses
- Each process is isolated in its virtual memory, and cannot access other process' virtual memories.

Address translation

Each virtual memory is mapped to some part of physical memory.

- Since virtual memory is not real, when a process tries to access (load or store) a virtual address, the virtual address is **translated (mapped)** to its corresponding physical address, and the load or store is performed in physical memory.

Q: Who performs address translation?

A: In modern computing architectures, address translation is usually performed by hardware, typically by a component named Memory Managment Unit (MMU) or Memory Protection Unit (MPU), using information provided by the OS.

This includes address translation for program counter (PC), as PC also contains virtual address only, hence, each instruction execution requires at least one address translation.

How is address translation done?

- Direct mapping
- Segmentation
- Paging
- ......

For details, please refer to relevant lectures on virtual memory in course CS 350.

Outline

What is sandboxing?

Sandboxing is a security mechanism for isolating vulnerable / untrusted code from its hosting platform, usually in an effort to confine the potential damage.

What damage?

- Corrupt in-application data
- Corrupt memory
- Corrupt local filesystem
- Corrupt other processes
- Gain root privilege
- Spread into other network-connected computers
- ......

The ladder of separation

- Physical separation
  - *e.g.*, Airgap, RF-shield rooms
- Hardware isolation
  - *e.g.*, AWS dedicated instances
- Whole-system virtualization
  - *e.g.*, Full (VMware ESXi) / Para (Xen)
- Whole-system emulation
  - *e.g.*, QEMU (+ KVM) emulation, Android emulator
- Partial system resources emulation
  - *e.g.*, Docker, Landlock, Jail
- In-process application sandboxes
  - *e.g.*, Chrome Sandbox, capabilities, `seccomp`
- In-thread application sandboxes
  - *e.g.*, hardware-assisted solutions like CHERI

Airgap demonstration

Figure: Airgapped computers in the 2016 DARPA CGC event

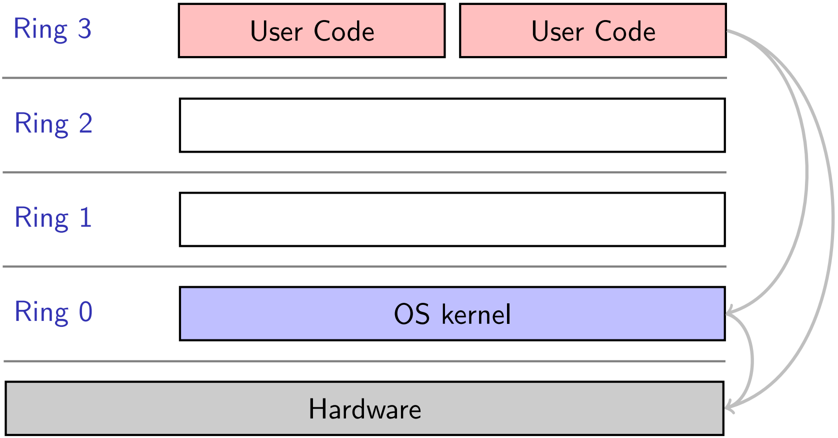
AWS dedicated instances

Based on AWS documentation

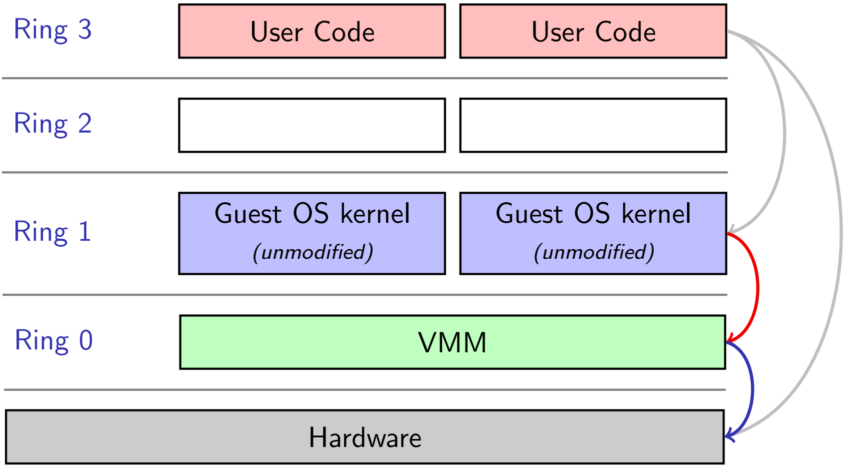
> *Dedicated Instances are Amazon EC2 instances that run in a virtual private cloud (VPC) on hardware that's dedicated to a single customer. Dedicated Instances that belong to different AWS accounts are physically isolated at a hardware level, even if those accounts are linked to a single payer account.*

Outline

x86 privilege levels

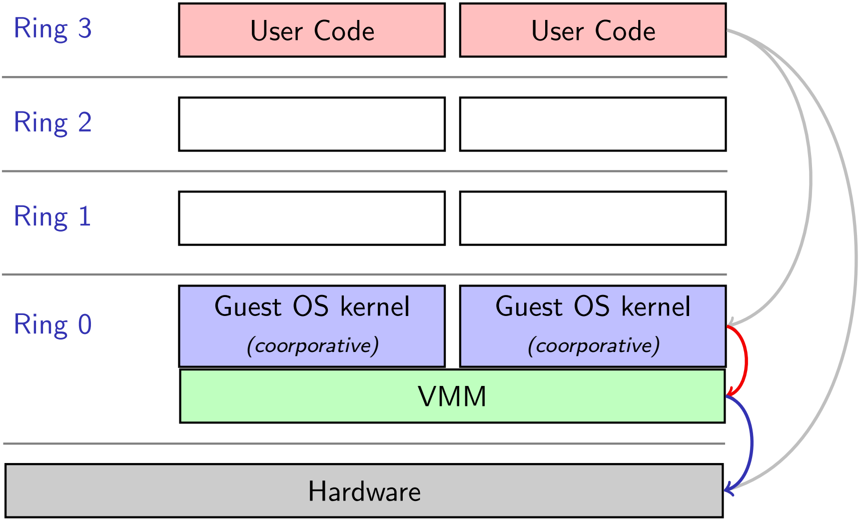


Full virtualization (bare-metal)



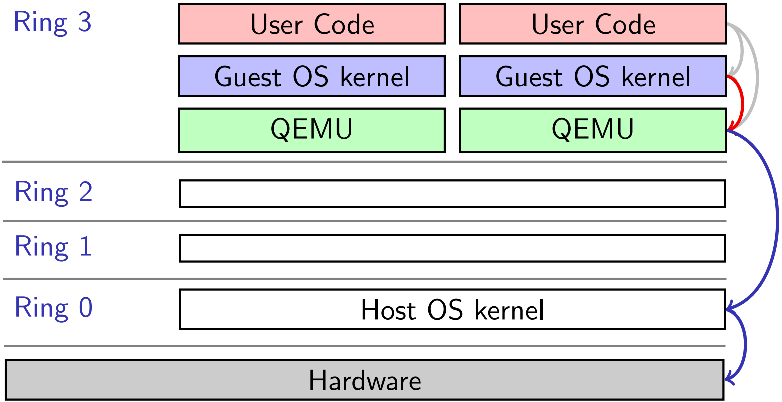
Trap accesses to hardware resources into the Virtual Machine Manager (VMM), possibly via binary translation.

Paravirtualization



Instrument the guest kernel with hypercalls to interact with VMM.

Whole-system emulation



Whole-system emulation attempts to run the entire stack in user-space, including the emulation of hardware devices.

Outline

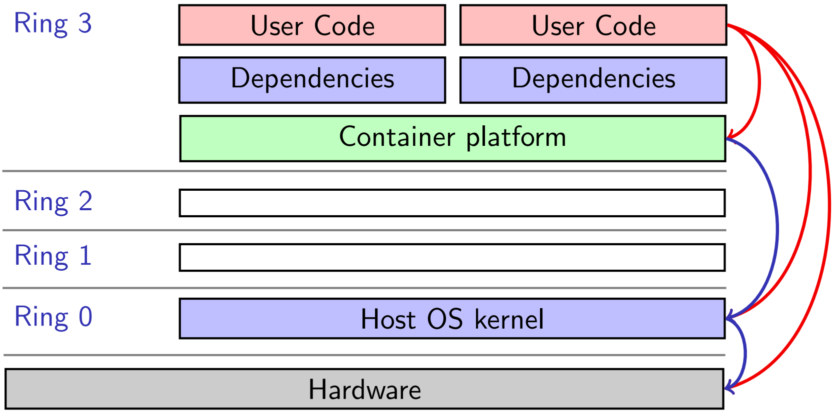
What is a container?

Figure: A cargo ship. Credits / Trademark: Tech Vision

What does a container sees itself?

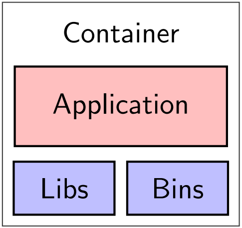
Figure: A single-container ship. Credits / Trademark: MarineTraffic

Containerized application



A containerized application has a delusion that it is the only application running on the platform (other than the dependencies).

From the view of a containerized application



Q: What does a containerized application need to run?

- memory
- filesystem
- networking
- threading / scheduling
- process management
- inter-process communications

Docker on Linux — control groups (cgroups)

Control groups (cgroups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage of a collection of processes. Covered resources include

- memory
- CPU
- block I/O
- network
- *device drivers ...*
- some exotic use cases, such as
  - huge pages (an efficient way of memory allocation)
  - RDMA (for faster memory accesses)
  - ......

Docker on Linux — control groups (cgroups)

cgroups also allows to group processes for batch operations such as:

- freezer (conceptually similar to a mass-SIGSTOP/SIGCONT)
- perf_event (gather performance statistics on these processes)
- cpuset (limit or pin processes to specific CPUs)
- Limit number of pids (i.e., processes) in the group

When a process is created, it is placed in its parent's cgroups

Docker on Linux — namespaces (`ns`)

While `cgroups` limits how much a process can use, `ns` limits what a process can see (and hence make use of).

These namespaces are typically available in modern Linux kernels:

- pid: only "see" processes in the same PID namespace
- net: networking interfaces
- mnt: root fs, private mounts (`/tmp`), masking `/proc`, `/sys`, etc
- uts: hostname
- ipc: ns-specific IPC semaphores, message queues, shared memory
- user: allows UID/GID mapping (*e.g.*, UID 0→99 to 1000→1099)
- time: allows slower/faster clock or an offset to the clock

Each process belongs to one namespace of each type. A new process can re-use none / all / some of the namespaces of its parent

Docker on Linux — namespaces (`ns`)

$ `sudo unshare --uts`

- create new uts namespace while inheriting everything else.

$ `hostname`

> `system76-pc`

$ `hostname cs489`

$ `hostname`

> `cs489`

In another `shell`, check that the hostname remains:

$ `hostname`

> `system76-pc`

Docker on Linux — copy-on-write filesystem (`OverlayFS`)

While Docker generally considers it a mechanism for fast container launch, the overlay filesystem concept itself is a very powerful sandboxing mechanism.

Docker on Linux — copy-on-write filesystem (`OverlayFS`)

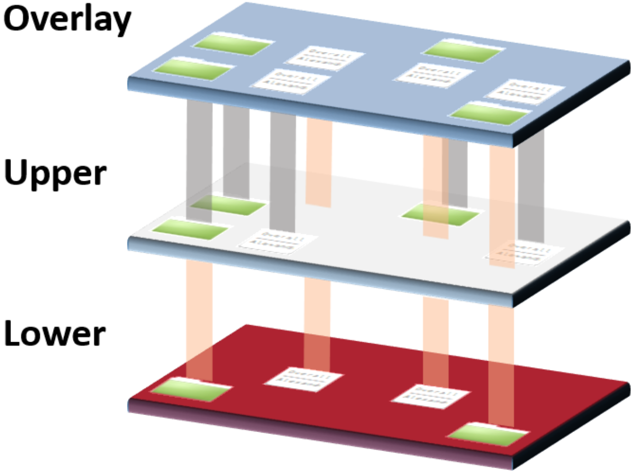


Figure: An illustration of the `OverlayFS`. Credits / Trademark: Datalight

Docker on Linux — copy-on-write filesystem (`OverlayFS`)

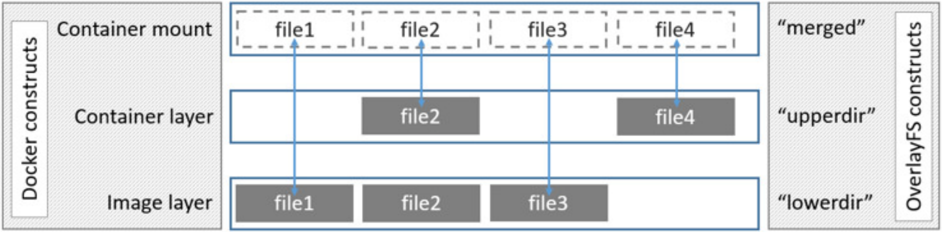


Figure: An illustration of the `OverlayFS`. Credits / Trademark: Docker

Outline

Why would a program sandbox itself?

The most common reason is to deal with untrusted code, *e.g.*,

- Javascript received from website
- Macros carried in documents (*e.g.*, Excel or PDF)
- Language runtime running untrusted application code

Case study: Linux capabilities

In traditional UNIX, many operations are possible when you have UID = 0 (root):

- changing file ownership, accessing all files, ...
- setting up network interfaces, mounting filesystems ...
- binding to a port below 1024 ...
- load and unload kernel modules ...
- ......

But why would a web server has accesses to kernel modules or the ability to mount / unmount filesystems?

Case study: Linux capabilities

Capabilities are per-process flags to allow privileged operations individually (which used to be granted to root as a package).

- `CAP_CHOWN`: arbitrarily change file ownership and permissions.
- `CAP_DAC_OVERRIDE`: arbitrarily bypass file ownership and permissions.
- `CAP_NET_ADMIN`: configure network interfaces, iptables rules, etc.
- `CAP_NET_BIND_SERVICE`: bind a port below 1024.
- `CAP_SYS_MODULE`: load or unload kernel modules.
- ...

See `man capabilities` for the full list and more details.

Case study: Seccomp

Can we have more fine-grained sandboxing?

In a more verbose way, I know exactly what my sub-process should do, can I achieve principle of least privilege?

Enter `seccomp`, that prevent execution of certain system calls by an application, through a customizable filter.

Why system calls?

The Linux kernel exposes a large number of system calls (≈400), while most program only need a small subset to function. — The practicality argument.

In addition, the most common way of making an impact on the host platform is via system calls. — The effectiveness argument.

Seccomp: strict mode

Only permit the following system calls: `read()`, `write()`, `_exit()`, `sigreturn()`. Any other system calls leads to `SIGKILL`.

- NOTE: `open()` not included.

Designed to sandbox untrusted code that is compute-intensive.

Seccomp: BPF filter

Allows filtering based on system call number and argument values *(pointers are not dereferenced)*.

Steps to use BPF filter:

1. Construct filter in BPF rules
2. Install filter using `seccomp()` or `prctl()`
3. `exec()` new program or invoke function in dynamically loaded shared libraries (a.k.a., plug-ins).

Once install, every system call triggers execution of filter.

Seccomp: from BPF to eBPF

Conventional BPF rules are stateless, *i.e.*, the filtering decision is solely based on the current system call being invoked and not based on history of invocations.

eBPF, however, can be stateful. It is in fact a virtual machine in the Linux kernel with its own instruction set and programming model.

In essence, eBPF allows arbitrarily complex checks to be performed quickly and safely.

$\langle$ **End** $\rangle$