CS 155: Computer Security

Isolation

Logistics

Project 1 part 2 due tomorrow (4/16)

The big picture

Many possible attacks: Malware, supply-chain attacks, phishing attack, …

Computer security goals:

- Defend against attacks
- Limit damage
  ↳ Confinement

Outline

What is confinement?

Types of confinement:

1. System call interposition (sandboxing a process)
2. Virtual machines
3. Software fault isolation

Outline

What is confinement?

Types of confinement:

1. System call interposition (sandboxing a process)
2. Virtual machines
3. Software fault isolation

Running untrusted code

We often need to run buggy or untrusted code:

- Programs from untrusted Internet sites:
  ↳ Mobile apps, Javascript, browser extensions
- Exposed applications:
  ↳ Browser, PDF viewer, outlook
- Legacy applications:
  ↳ sendmail, bind
- Honeypots
- AI agents

Confinement goal

Goal: Ensure misbehaving application cannot harm the rest of the system

- **Non-interference:** Specify some policy that limits how App 1 can affect App 2

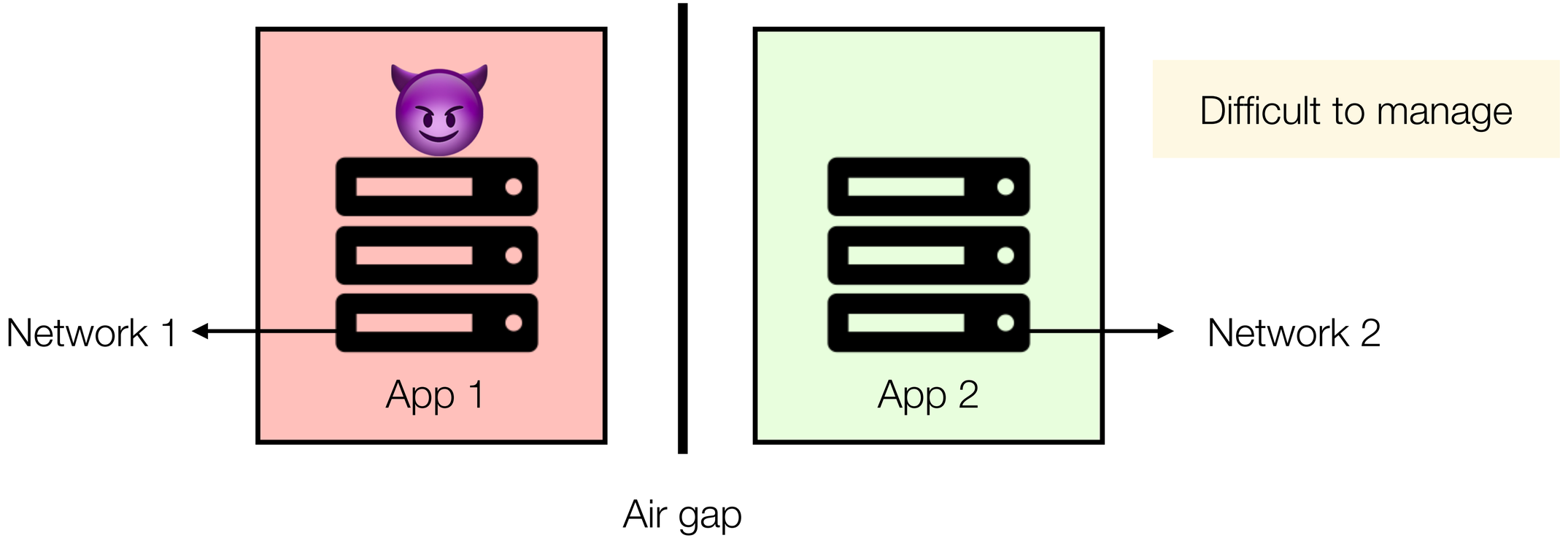
Approach: If application misbehaves, either return error or kill application

Confinement at many levels

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Hardware**: Run application on isolated hardware (air gap)



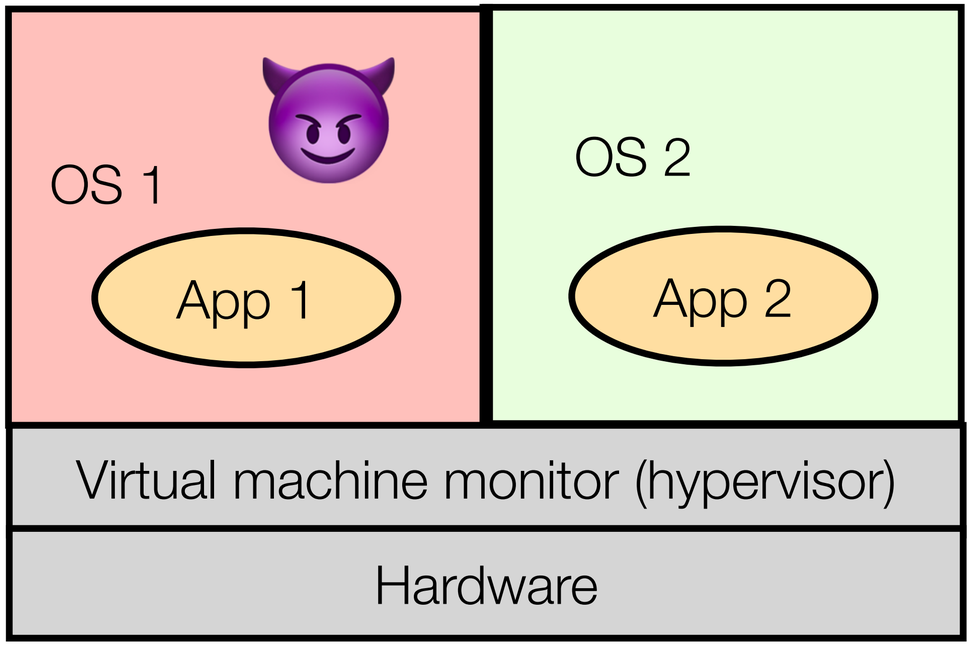
Difficult to manage

Confinement at many levels

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Virtual machines:** Isolate operating systems on a single machine

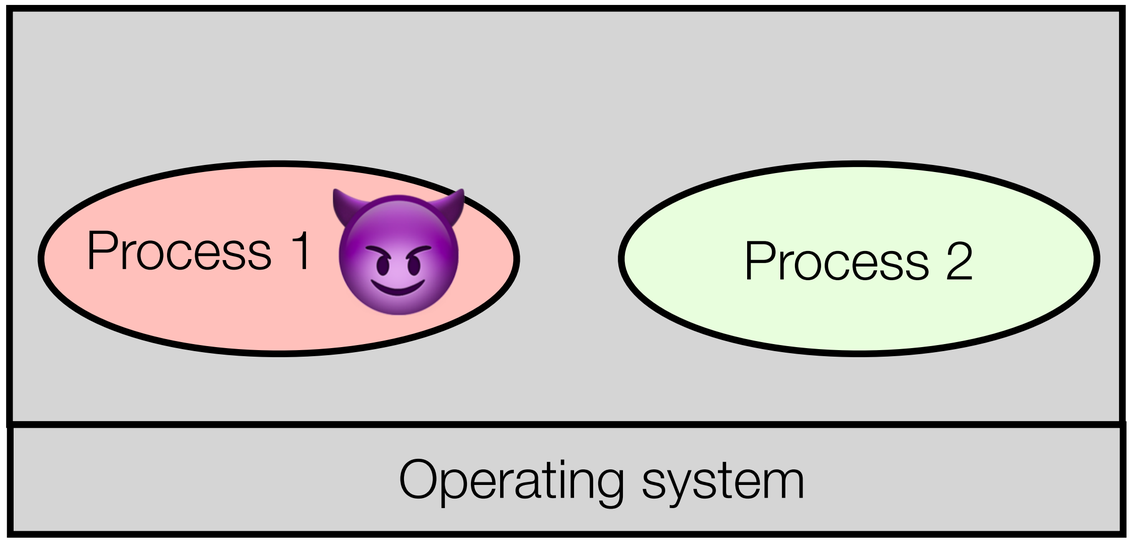


Confinement at many levels

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Process:** System call interposition (containers)
  ↳ Isolate processes in a single OS



Confinement at many levels

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Threads:** Software fault isolation (SFI)
  ↳ Isolate threads sharing the same address space
- **Application-level confinement**
  ↳ EX: browser sandbox for Javascript and WebAssembly

Challenges in implementing confinement

Difficult to achieve

- Adversary and victim share limited resources
  ↳ CPU, RAM, network bandwidth, …

Performance

- We could give app 1 half the resources and app 2 half the resources
- … but this would lead to bad performance
- How to provide strong isolation at a low cost?

Implementing confinement

Key component: **Reference monitor**

- **Mediates requests** from applications
  ↳ Enforces confinement
  ↳ Implements a specified protection policy
- Must **always** be invoked
  ↳ Every application request must be mediated
- **Tamperproof**
  ↳ Reference monitor cannot be killed… or if killed, monitored process is also killed

An old example: `chroot`

To use (must be root):

```
chroot /tmp/guest   // root dir / is now /tmp/guest
su guest            // EUID set to “guest”
```

Now `/tmp/guest` is added to every filesystem access:

```
fopen(“/etc/passwd”, “r”) -> fopen(“/tmp/guest/etc/passwd”, “r”)
```

Goal: Application (e.g., web server) cannot access files outside of jail

Escaping from jails

Early escapes: relative paths

```
fopen(“../../etc/passwd”, “r”) ->
    fopen(“/tmp/guest/../../etc/passwd”, “r”)
```

Escaping from jails

Chroot should only be executable by root. Why?

Jailed app can do:

- Create dummy file `/aaa/etc/passwd`
- Run `chroot /aaa`
- `su root`

Bug in Ultrix 4.0

Many ways to escape jail as root

- Create device that lets you access raw disk
- Send signals to non-chrooted process
- Reboot system
- Bind to privileged ports

Freebsd jail

- Stronger mechanism than simple chroot

To run: `jail jail-path hostname IP-addr cmd`

- Calls hardened chroot (no `../../` escape)
- Can only bind to sockets with specified IP address and authorized ports
- Can only communicate with processes inside jail
- Root is limited (e.g., cannot load kernel modules)

Problems with chroot and jail

Coarse policies:

- All or nothing access to parts of file system
- Inappropriate for apps like a web browser
  ↳ Needs read access to files outside jail (e.g., sending attachments in Gmail)

Does not prevent malicious apps from:

- Accessing network and messing with other machines
- Trying to crash host OS

Outline

What is confinement?

Types of confinement:

1. **System call interposition** (sandboxing a process)
2. Virtual machines
3. Software fault isolation

System call interposition

Observation: To damage host system (e.g., persistent changes), app must make system calls

- To delete/overwrite files: `unlink, open, write`
- To perform network attacks: `socket, bind, connect, send`

Idea: monitor app's system calls and block unauthorized calls

Implementation options:

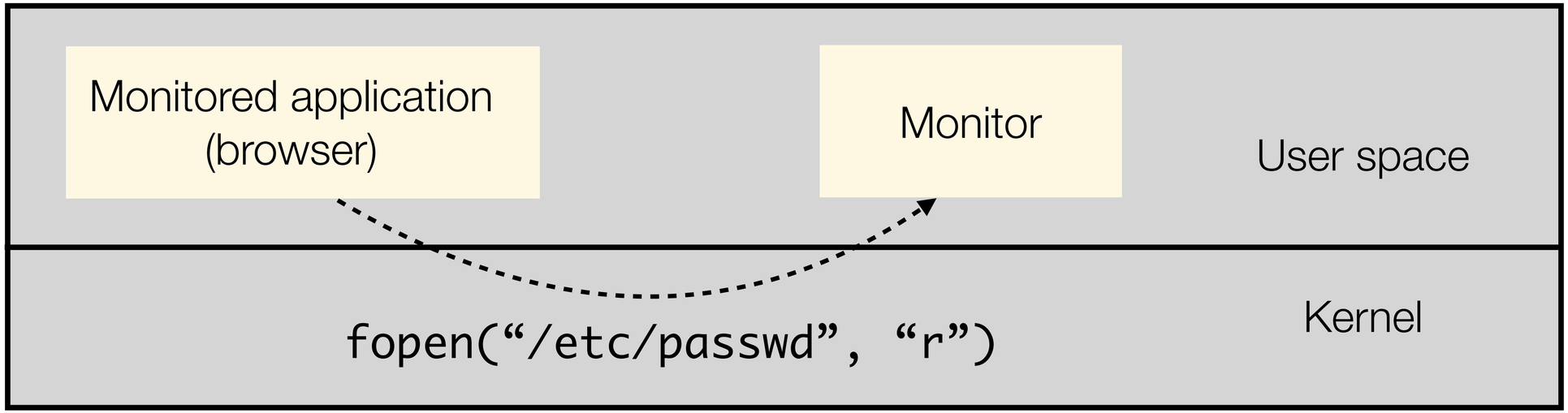
- Completely kernel space (e.g., Linux seccomp)
- Completely user space (e.g., program shepherding)
- Hybrid (e.g., systrace)

Early implementation with ptrace (Janus)

Linux ptrace: process tracing

- Process calls `ptrace(…, pid_t pid, …)`
- Process wakes up when `pid` makes system call

Monitor kills application if request is disallowed



Example policy

An example policy (e.g., for PDF reader):

```
path allow /tmp/*
path deny /etc/passwd
network deny all
```

Manually specifying policy for an application can be tricky:

- Recommended default policies are available
- Can be made more restrictive as needed

Complications

- If app forks, then monitor must also fork
  ↳ Forked monitor monitors forked app
- If monitor crashes, app must be killed
- Monitor must maintain all OS state associated with app
  ↳ current working directory (CWD), UID, EUID, GID
  ↳ When app runs `cd path`, monitor must update its CWD
  (otherwise, relative paths interpreted incorrectly)

```
cd("/tmp")
open("passwd", "r")
```

```
cd("/etc")
open("passwd", "r")
```

Problems with ptrace

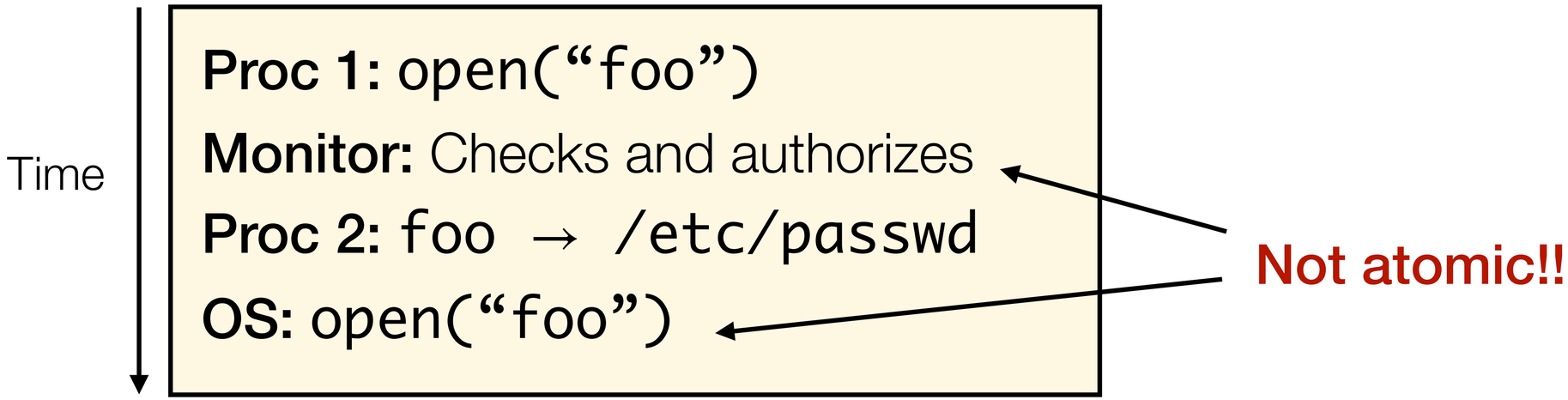
Ptrace is not well-suited for this application:

- Trace all system calls or none
  Inefficient: no need to trace some system calls
- Monitor cannot abort sys call without killing app

Problems with ptrace

Security problem: Race conditions

Example: symlink: `foo → bar`

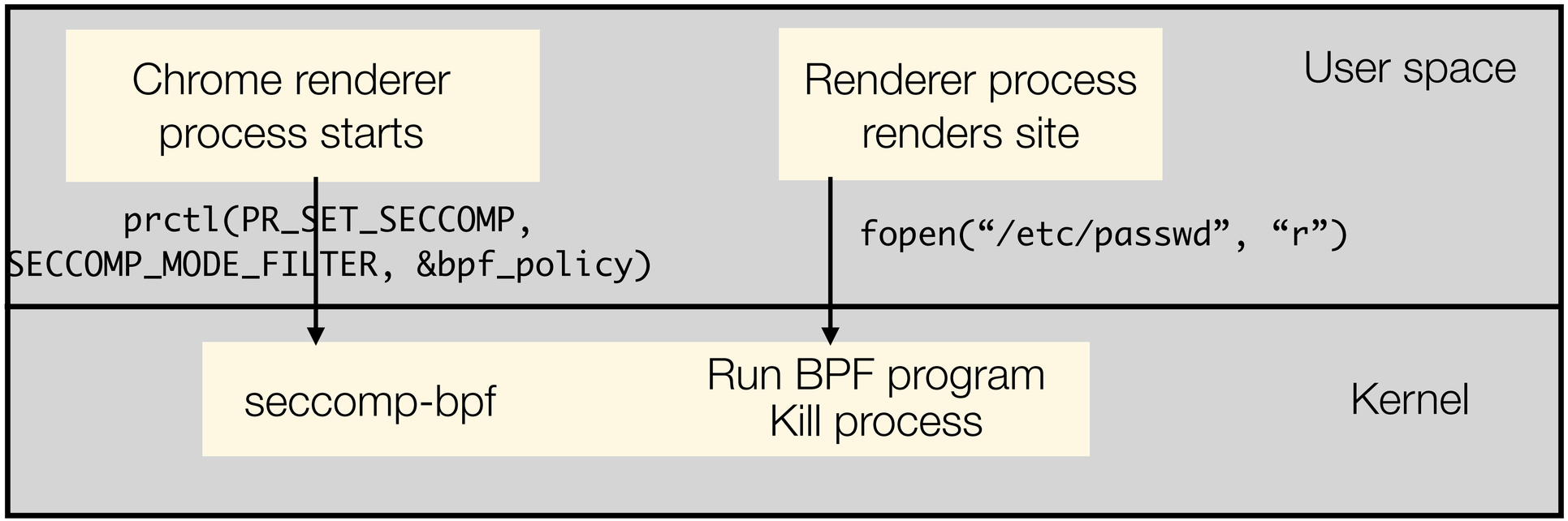


TOCTOU bug: Time-of-check / time-of-use

System call interposition: seccomp-bpf

Seccomp-BPF: Linux kernel facility used to filter process system calls

- Syscall filter written in the BPF language (use BPFC compiler)
- Used in Chromium, Docker containers, …



BPF filters (policy programs)

Process can install multiple BPF filters:

- Once installed, filter cannot be removed (all run on every syscall)
- If program forks, child inherits all filters
- If program calls execute, all filters are preserved

BPF filter input: syscall number, syscall args, arch (x86 or ARM)

Filters returns one of:

- `SECCOMP_RET_KILL`: kill process
- `SECCOMP_RET_ERRNO`: return specified error to caller
- `SECCOMP_RET_ALLOW`: allow syscall

Installing a BPF filter

```
int main (int argc, char **argv) {
    prctl(PR_SET_NO_NEW_PRIVS, 1);
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &bpf_policy)
    fopen(“file.txt”, “w”);
    printf(“will not be printed”);
}
```

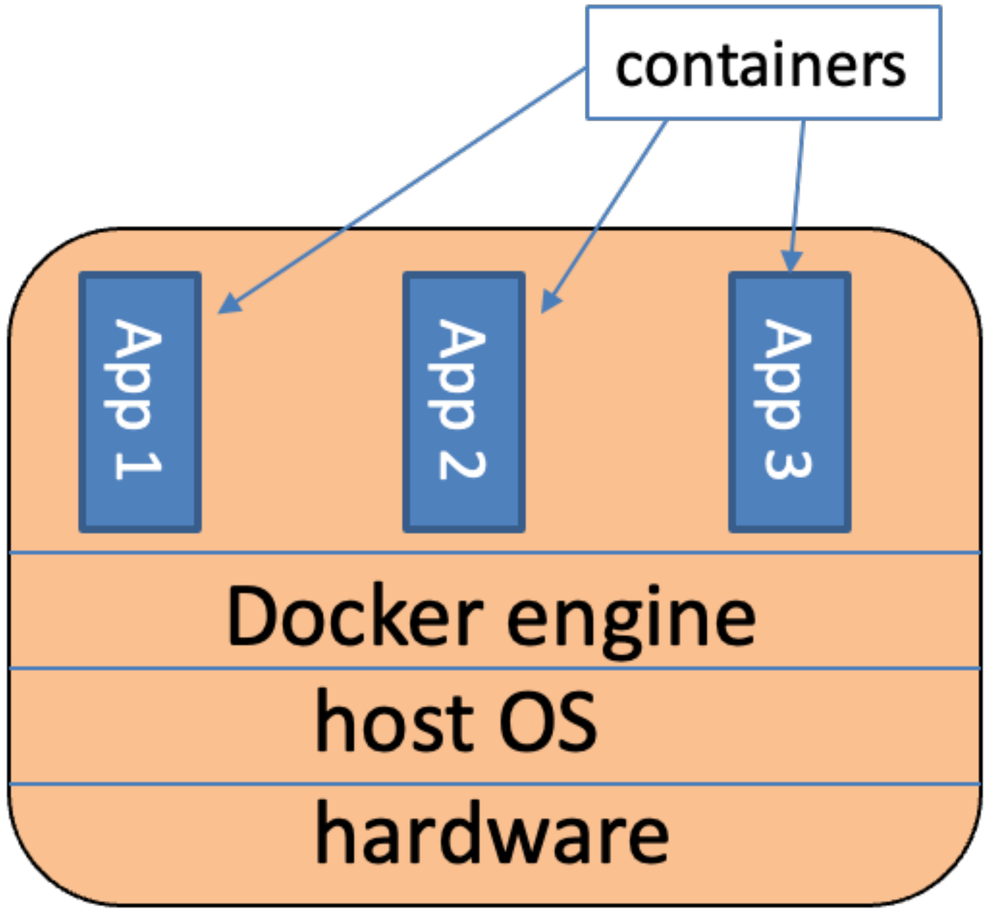
Must be called before setting BPF filter
Ensures set-UID, set-GID ignored on later execve
↳ attacker cannot elevate privilege

Kill if call open() for write

Docker: Isolating containers using seccomp-bpf

Container: Process-level isolation

- Container prevented from making sys calls filtered by seccomp-bpf
- Whoever starts container can specify BPF policy
  ↳ Default policy blocks many syscalls, including ptrace



Docker syscall filtering

Run nginx container with a specific filter called `filter.json`

`docker run –security-opt=“seccomp=filter.json”`

Example filter:

```
“defaultAction”:  “SCMP_ACT_ERRNO”,  //  deny by default
  “syscalls”: [
     {  "names":  ["accept”],   //  sys-call name
        "action":  "SCMP_ACT_ALLOW",  //  allow (whitelist)
        "args": [ ]  } ,  // what args to allow
        …
    ]
```

More Docker confinement flags

Specify run as an unprivileged user:

```
docker run –user www nginx
```

Limit Linux capabilities:

```
docker run –cap-drop all –cap-add NET_BIND_SERVICE nginx
```

Drop all capabilities

Allow to bind to privileged ports

Prevent processes from becoming privileged (e.g., by setUID binary):

```
docker run –security-opt=no-new-privileges:true nginx
```

Limit number of restarts and resources (# open files, # processes):

```
docker run –restart=on-failure:max-retries
  –ulimit nofile=<max-fd> –ulimit nproc=<max-proc> nginx
```

Outline

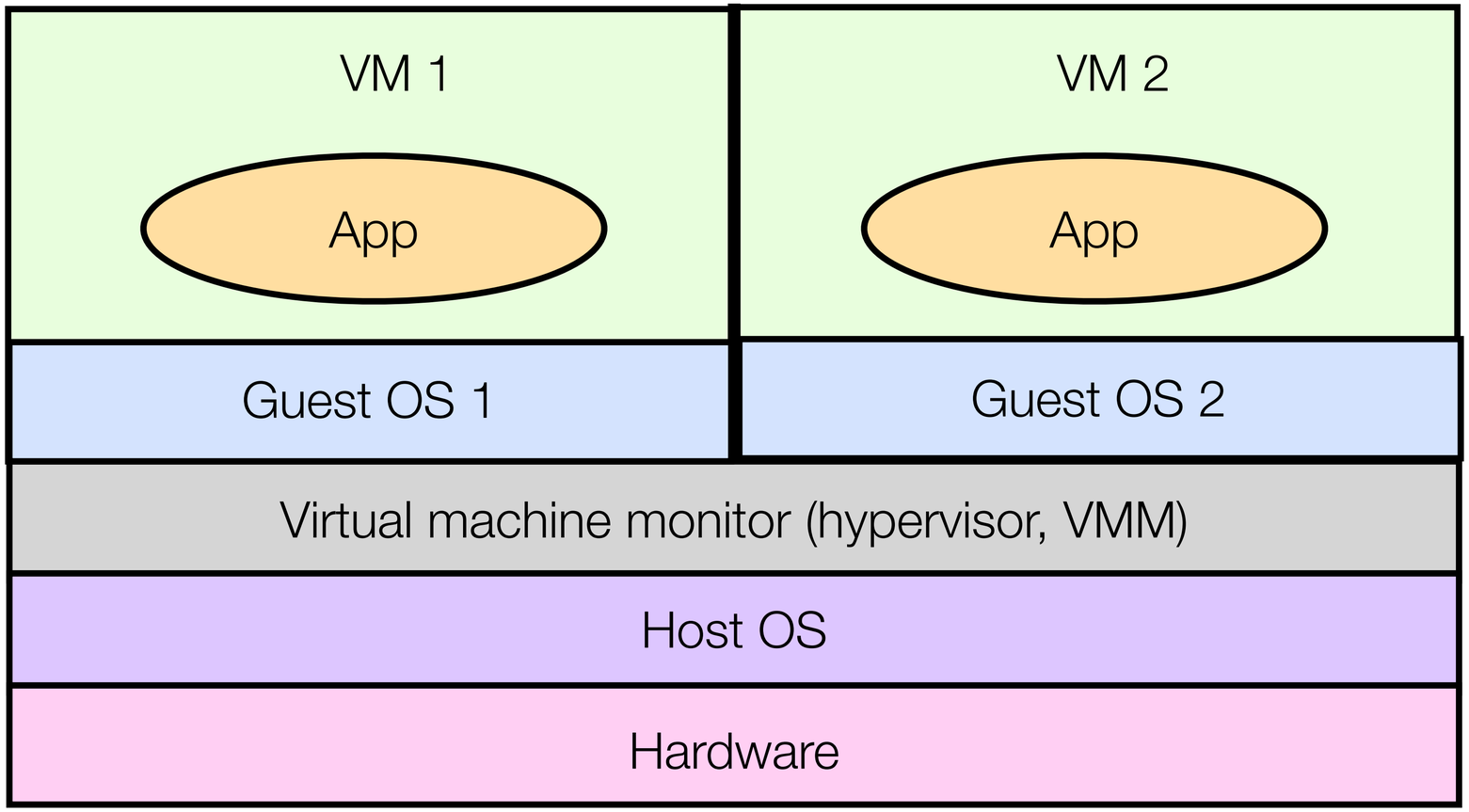
What is confinement?

Types of confinement:

1. System call interposition (sandboxing a process)
2. **Virtual machines**
3. Software fault isolation

Virtual machines

Single HW platform with isolated components



Hypervisor security assumption

Hypervisor security assumption:

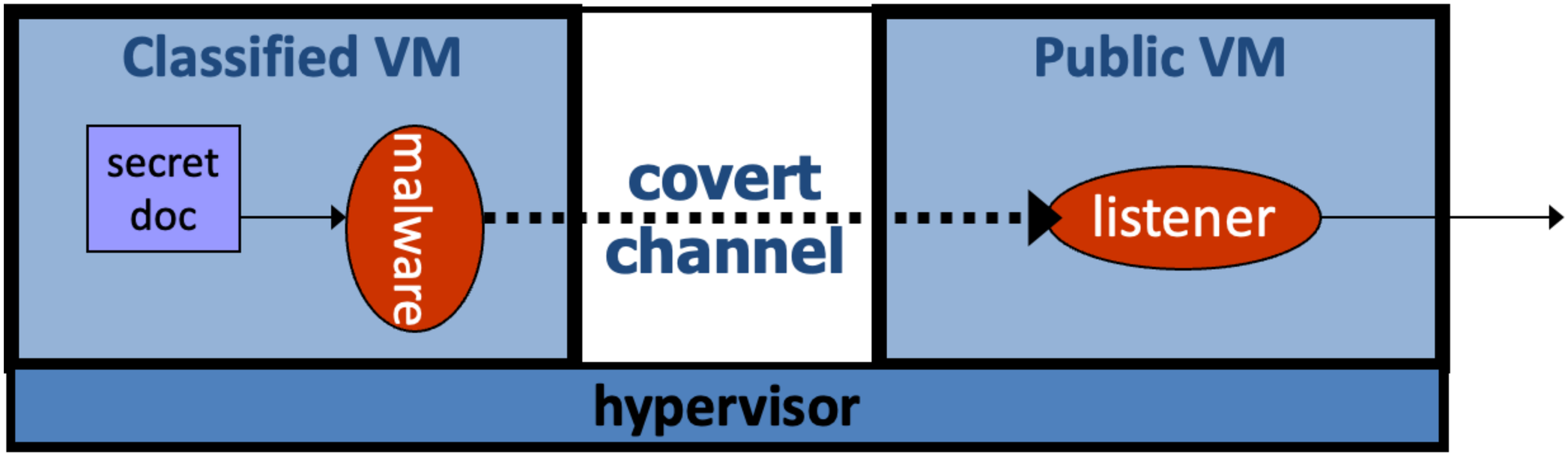
- Malware can infect guest OS and guest apps
- But malware cannot escape from the infected VM
  ↳ Cannot infect host OS
  ↳ Cannot infect other VMs on the same hardware

Requires that hypervisor protect itself and is not buggy
↳ (some) hypervisors are much simpler than a full OS

Problem: Covert channels

Covert channel: Unintended communication channel between isolated components

- Can leak classified data from secure component to a public component



An example covert channel

Both VMs have the same underlying hardware

To send a bit $b \in \{0,1\}$, the malware does:

- $b = 1$: At 1AM, do CPU-intensive calculation
- $b = 0$: At 1AM, do nothing

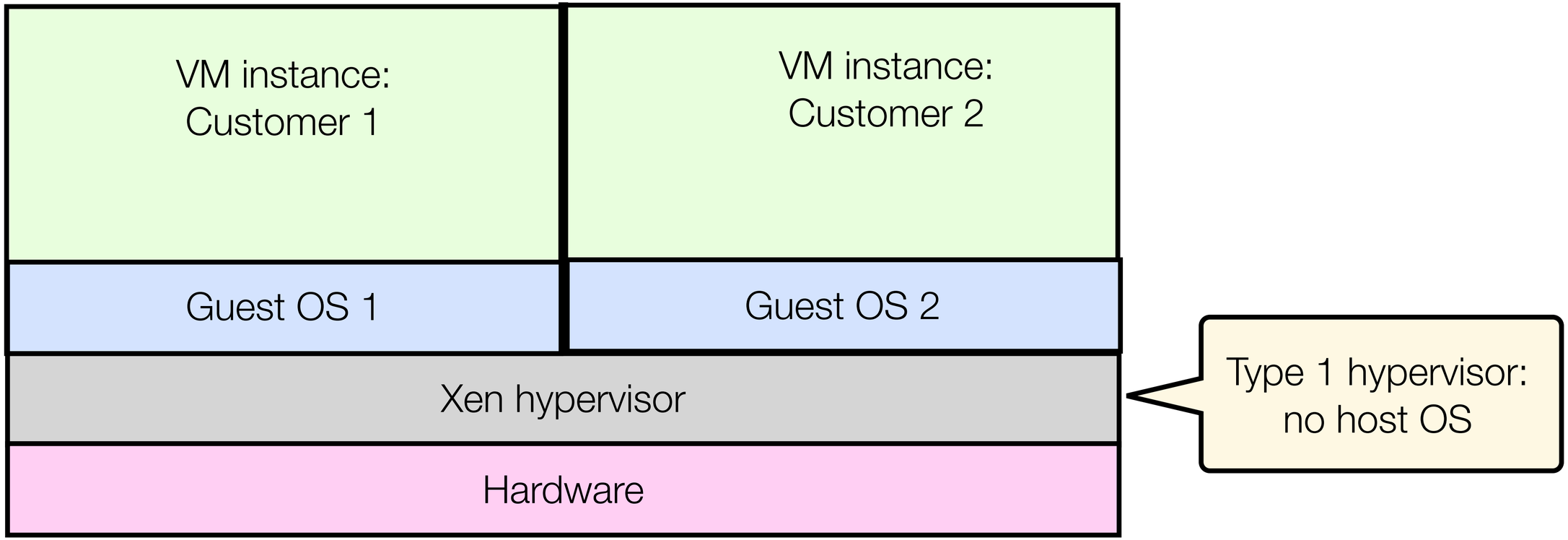
At 1AM, listener does CPU-intensive calculation and measures completion time

- $b = 1 \implies$ completion-time > threshold

Many covert channels exist in running systems:

- File lock status, cache contents, interrupts, …. Difficult to eliminate all!

VM isolation in practice: Cloud deployment



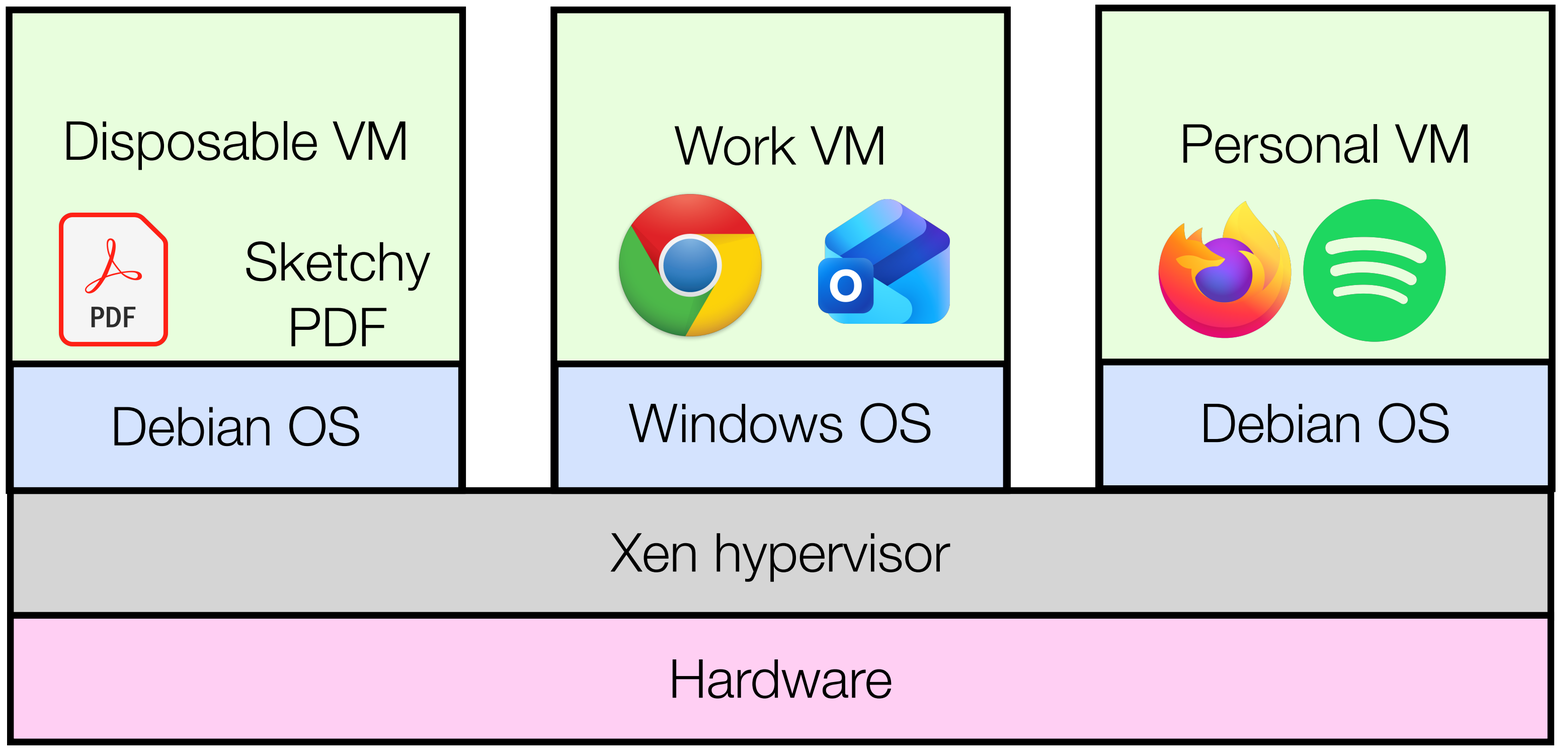
VMs from different customers may run on the same machine

↳ Hypervisors must isolate VMs … but some info leaks

VM isolation in practice: End-user

Quebes OS: A desktop/laptop OS where everything is a VM

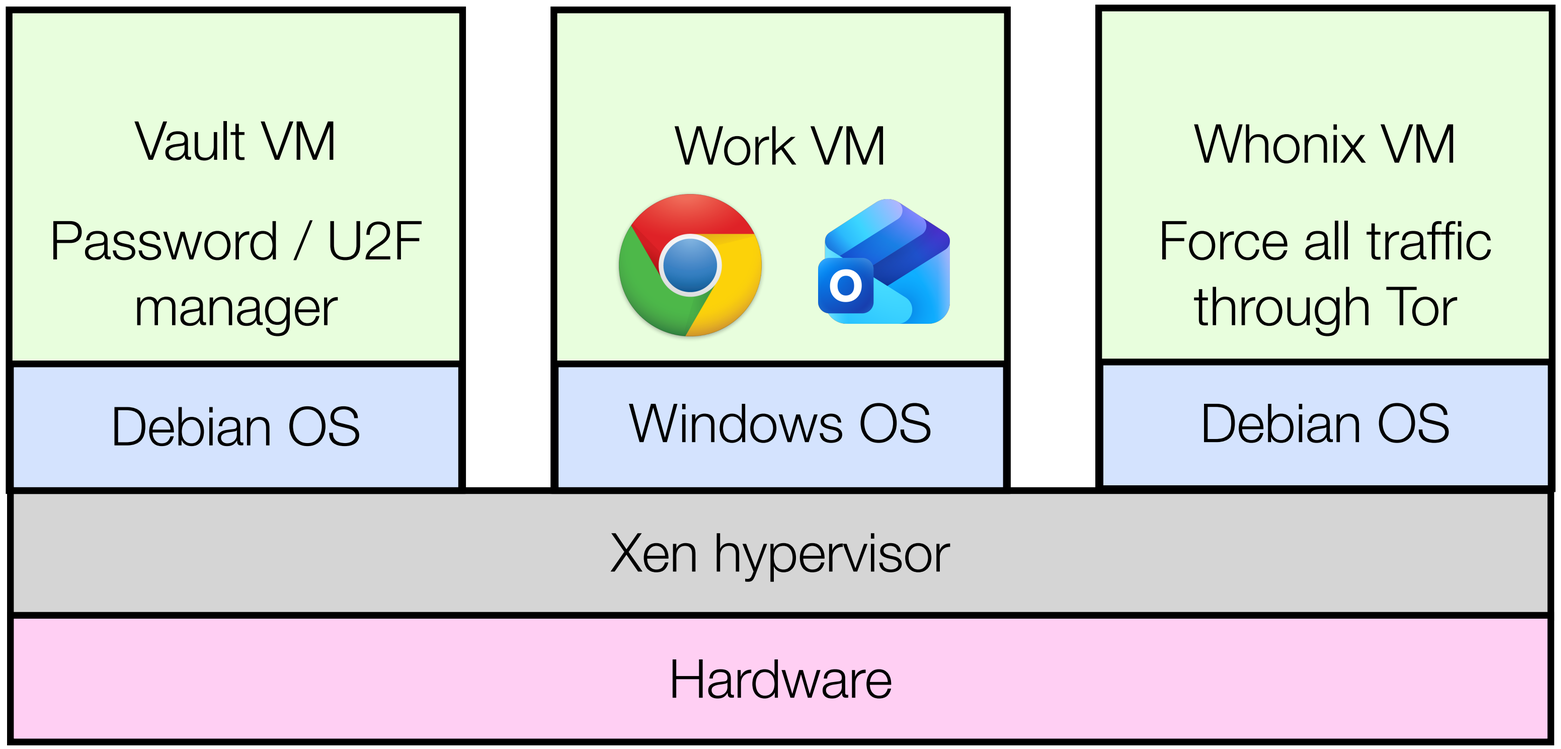
- Runs on top of Xen hypervisor
- Access to peripherals (mic, camera, USB, …) controlled by VMs



VM isolation in practice: End-user

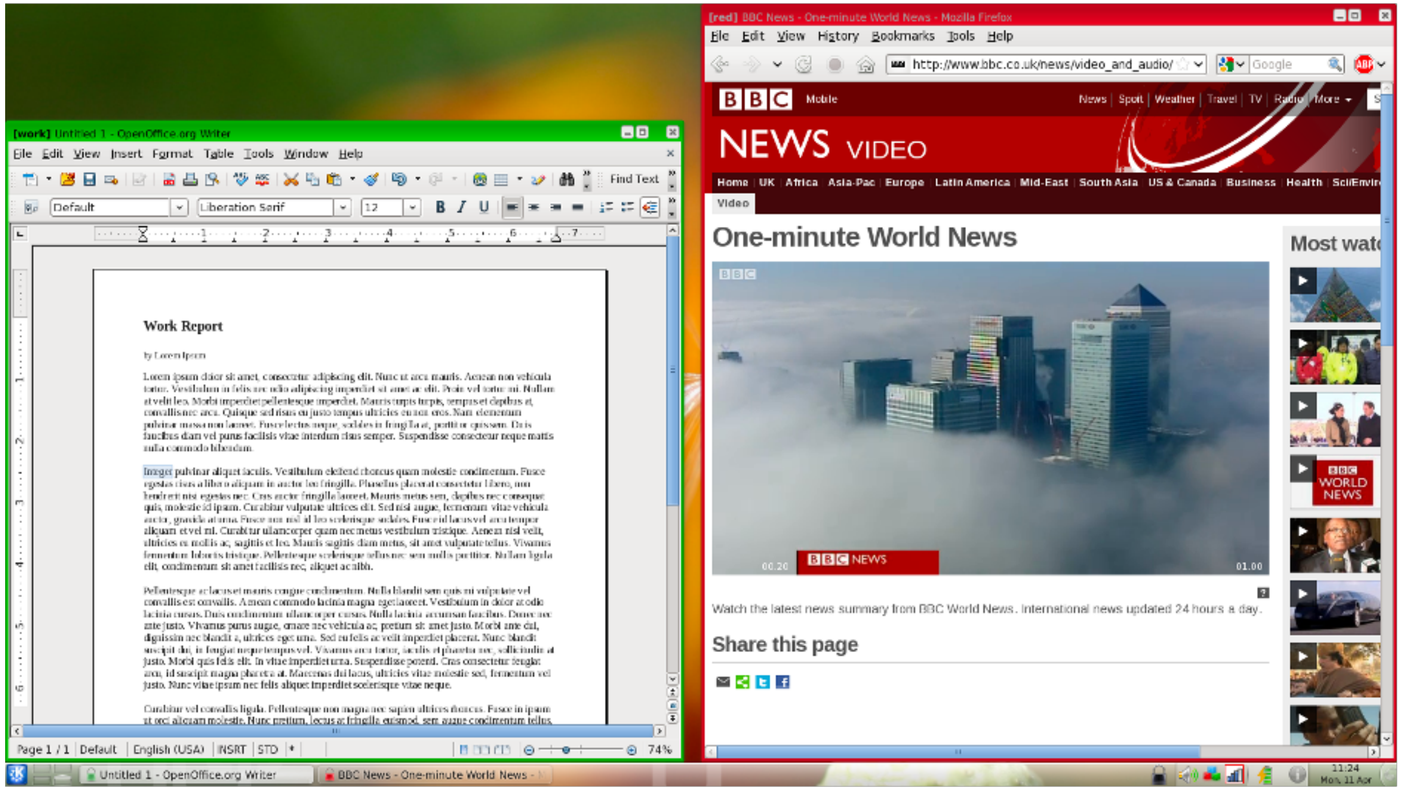
Quebes OS: A desktop/laptop OS where everything is a VM

- Runs on top of Xen hypervisor
- Access to peripherals (mic, camera, USB, …) controlled by VMs

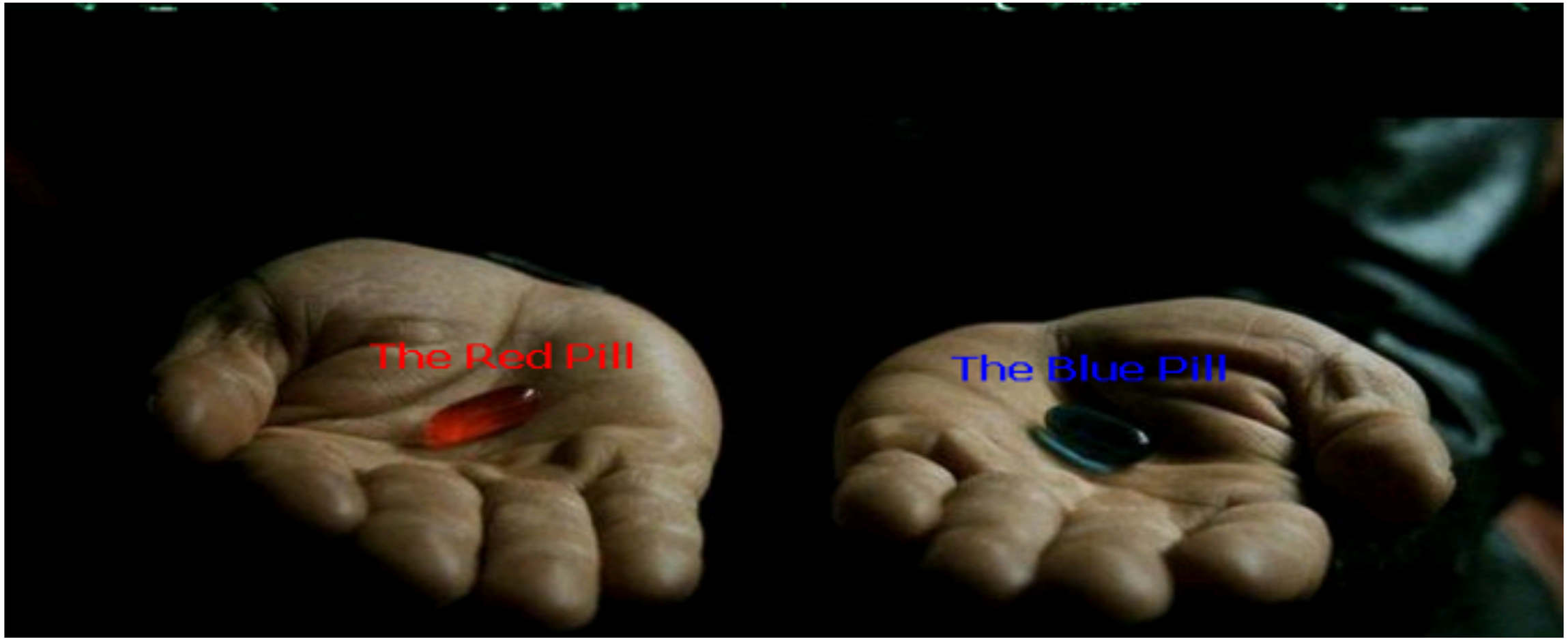


Every window frame identifies VM source

GUI VM ensures frames are drawn correctly



Hypervisor detection



Hypervisor detection (red pill techniques)

- VM platforms often emulate simple hardware
  ↳ VMware emulates an ancient i440bx chipset
    … but reports 64GB RAM, dual CPUs, etc.

- Hypervisor introduces time latency variances
  ↳ Memory cache behavior differs in presence of hypervisor
  ↳ Results in relative time variations for any two operations

- Hypervisor shares the TLB with GuestOS
  ↳ GuestOS can detect reduced TLB size

- … and many more methods [GAWF'07]

Hypervisor detection in the browser [HBBP'14]

Can we identify malware websites?

- Approach:
  1. Crawl web
  2. Load pages in browser running in a VM
  3. Look for pages that damage the VM

- Problem: Web page can detect it is running in a VM
  ↳ How? Using timing variations in writing to screen

- Malware in webpage becomes benign when in a VM
  ↳ Evade detection!

Hypervisor detection

Takeaway: Hypervisors are not fully transparent

- Anomalies reveal existence of hypervisor

Hypervisors today focus on

- Compatibility: ensure off-the-shelf software works
- Performance: minimize virtualization overhead

Outline

What is confinement?

Types of confinement:

1. System call interposition (sandboxing a process)
2. Virtual machines
3. **Software fault isolation**

Software Fault Isolation [Whabe et al., 1993]

Goal: confine apps running in the same address space

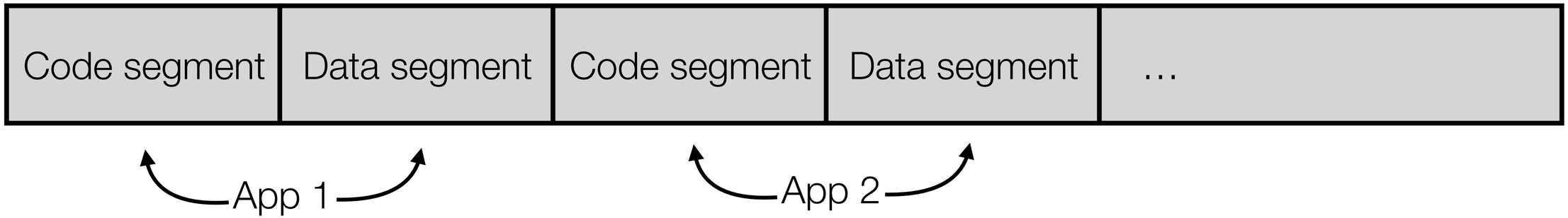
- Kernel modules should not corrupt kernel
- Native libraries should not corrupt JVM

Simple solution: run apps in separate address space

- Problem: slow if apps communicate frequently
  ↳ requires context switch per message

Software Fault Isolation

Approach: Partition process memory into segments



Locate unsafe instructions: `jmp, load, store`

- At compile time, add guards before unsafe instructions
- When loading code, ensure all guards are present

Segment matching technique

- Designed for MIPS processor where many registers available

- `dr1, dr2`: dedicated registers used by binary
  ↳ Compiler pretends these registers don't exist
  ↳ `dr2` contains segment ID

```
R12 <- [R34]
```

Indirect load instruction

```
dr1 <- R34
scratch-reg <- (dr1 >> 20)
if scratch-reg != dr2: trap
R12 <- [dr1]
```

Get segment ID
Validate segment ID
Do load

Guard ensures code does not load data from another segment

Address sandboxing technique

- Designed for MIPS processor where many registers available

- `dr1, dr2`: dedicated registers used by binary
  ↳ Compiler pretends these registers don't exist
  ↳ `dr2` contains segment ID

```
R12 <- [R34]
```

Indirect load instruction

```
dr1 <- R34 & segment-mask
dr1 <- dr1 | dr2
R12 <- [dr1]
```

Zero out seg bits

Set valid seg ID

Do load

Fewer instructions than segment matching
… but does not catch offending instructions
Similar guards placed on all unsafe instructions

SFI jumps

Problem: What if `jmp [addr]` jumps directly into indirect load, bypassing guard?

Solution: Guard for `jmp` instructions

- `jmp` guard ensures `[addr]` does not bypass load guard
- Idea: clear low order bits of `jmp` address

SFI takeaways

Performance is usually good: 4% slowdown on mpeg_play

SFI is harder to implement on x86:

- Variable length instructions: unclear where to put guards
- Fewer registers: Can’t dedicate small number to SFI
- Many instructions affect memory: more guards needed

Confinement takeaways

Many sandboxing techniques:

- Physical air gap
- Virtual air gap (hypervisor)
- System call interposition (SCI)
- Software fault isolation
- Application specific (e.g., Javascript in browser)

Often complete isolation is inappropriate

- Apps need to communicate through regulated interfaces

Hardest aspects of sandboxing

- Specifying policy: what can apps do and not do
- Preventing covert channels

Next time: Processor and microarchitecture security