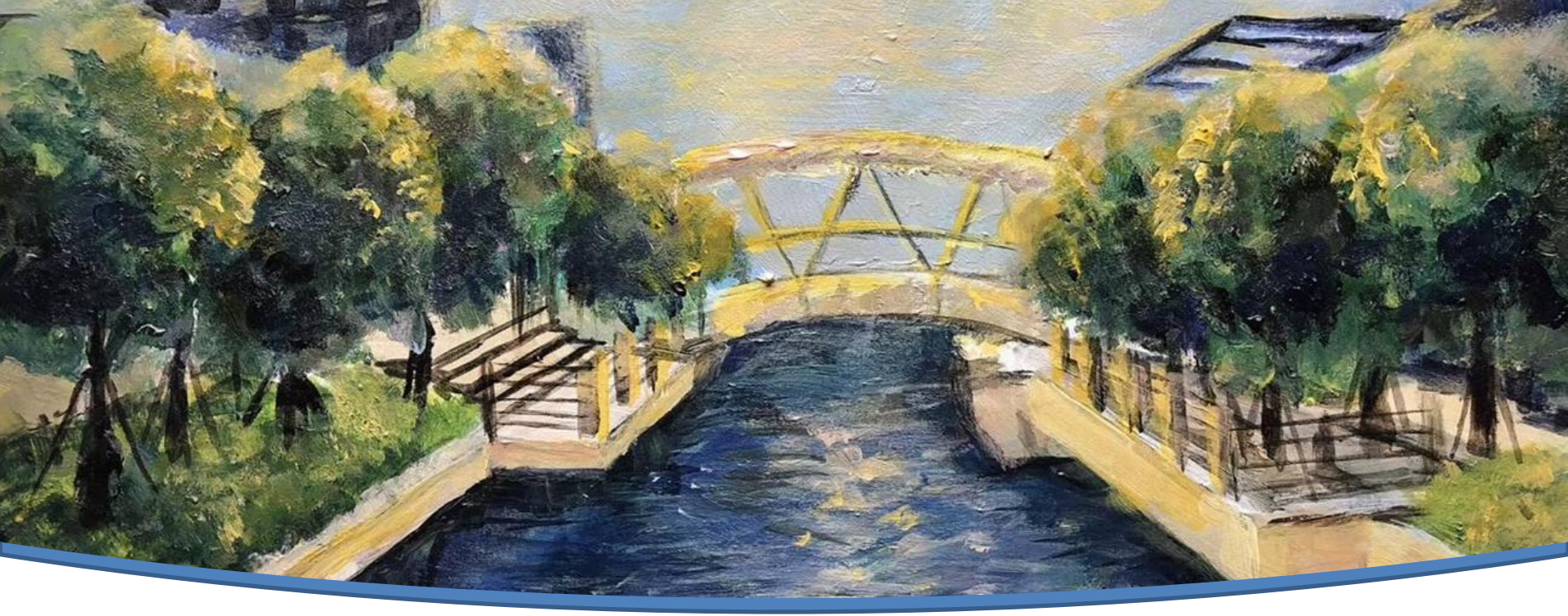


CS 253 Cyber Security
Isolation and Sandboxing

ShanghaiTech University

Admin

- HW 2 to be released
  - Some questions may cover topics of next week as well, so deadline is next week.
- Prepare your presentation and test the recording offline in advance!

PART ONE

01

Confinement

Running untrusted code

We often need to run buggy/unstrusted code:

- programs from untrusted Internet sites:
  - mobile apps, Javascript, browser extensions
- exposed applications: browser, pdf viewer, outlook
- legacy daemons: sendmail, bind
- honeypots

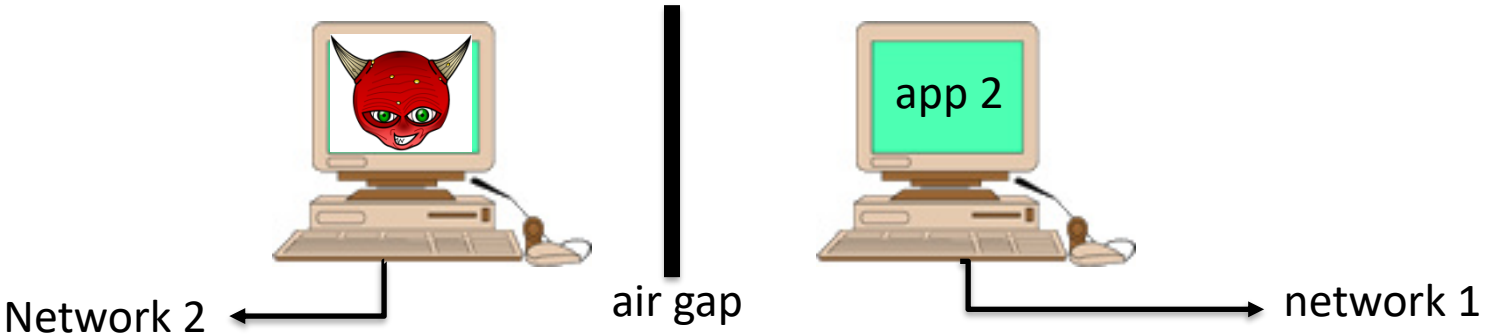
Goal: if application "misbehaves" ⇒ kill it

Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Hardware**: run application on isolated hw (air gap)



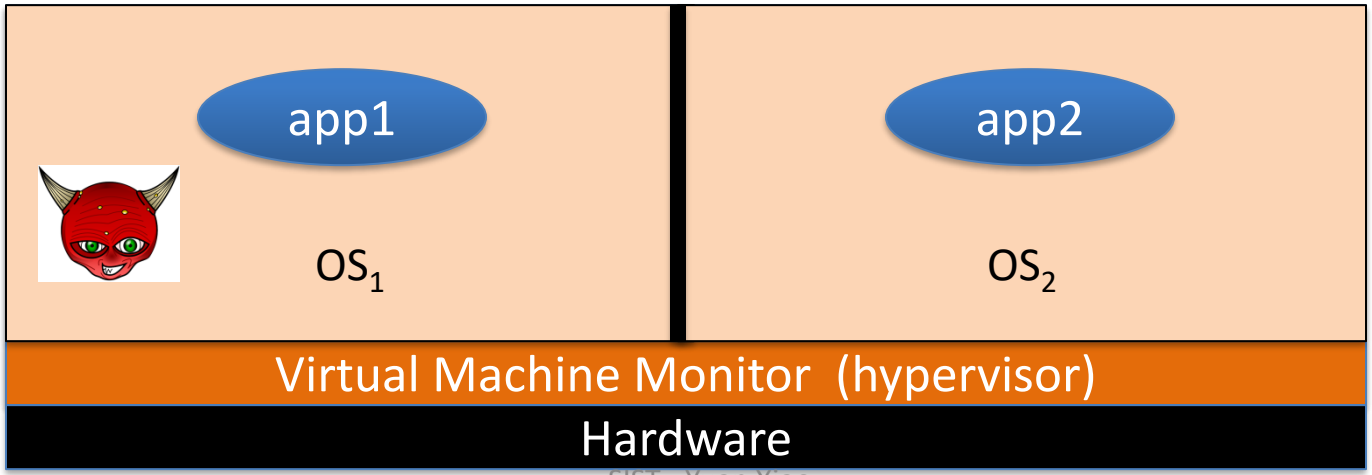
⇒ difficult to manage

Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Virtual machines**: isolate OS's on a single machine



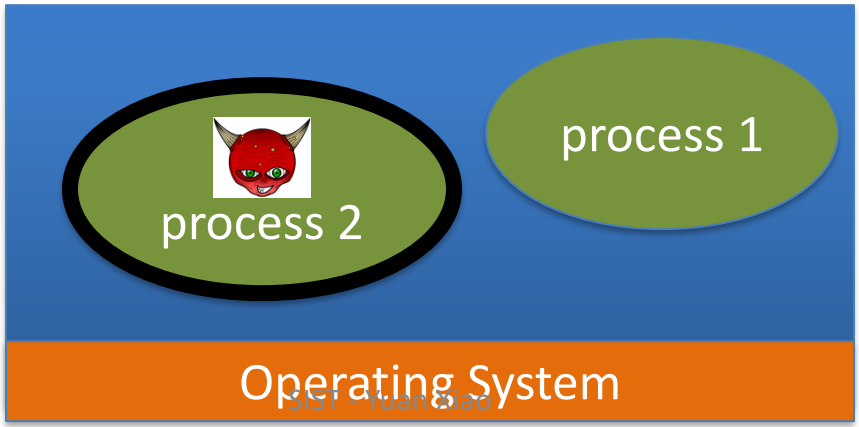
Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Process:** System Call Interposition (containers)

  Isolate a process in a single operating system



Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Threads:** Software Fault Isolation (SFI)
  - Isolating threads sharing same address space

- **Application level confinement**:
  e.g. browser sandbox for Javascript and WebAssembly

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, then monitored process is killed too
- **Small** enough to be analyzed and validated

A old example: chroot

To use do: (must be root)

```
chroot /tmp/guest
su guest
```

root dir "/" is now "/tmp/guest"

EUID set to "guest"

Now "/tmp/guest" is added to every file system accesses:

fopen("/etc/passwd", "r") ⇒
fopen("/tmp/guest/etc/passwd", "r")

⇒ 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")

---

chroot should only be executable by root.

- otherwise jailed app can do:
  - create dummy file "/aaa/etc/passwd"
  - run chroot "/aaa"
  - run su root to become root

(bug in Ultrix 4.0)

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., for sending attachments in Gmail)

Does not prevent malicious apps from:

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



System Call Interposition:

sanboxing a process

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 do network attacks: socket, bind, connect, send

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

Implementation options:

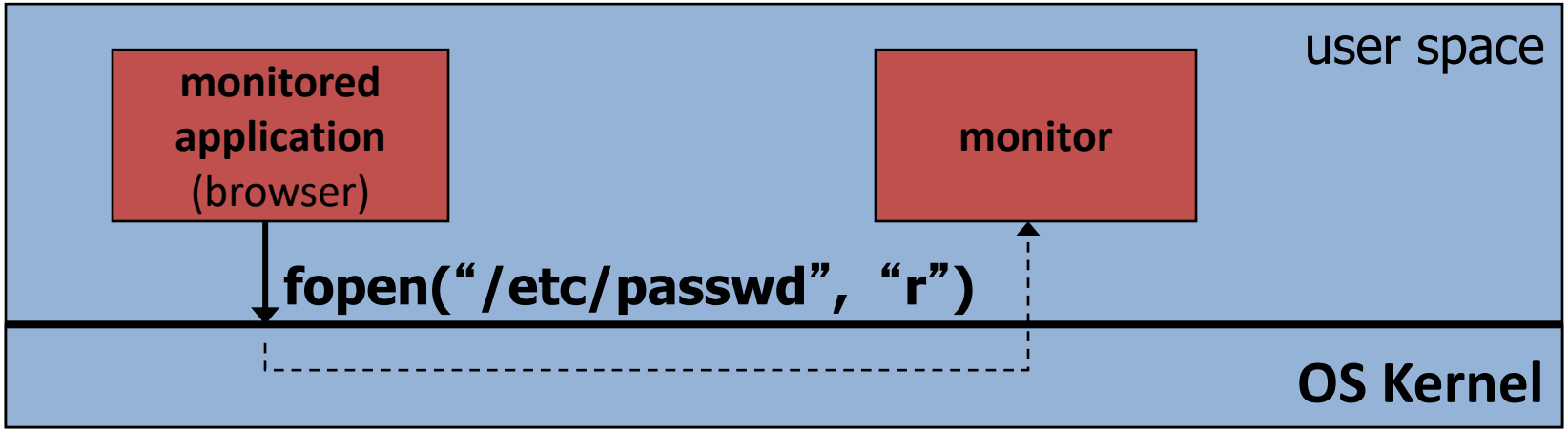
- Completely kernel space (e.g., Linux seccomp)
- Completely user space (e.g., program shepherding)
- Hybrid (e.g., Systrace)

Early implementation (Janus) [GWTB'96]

Linux **ptrace**: process tracing

process calls: **ptrace (… , pid_t pid , …)**

and wakes up when **pid** makes sys call.



Monitor kills application if request is disallowed

Example policy

Sample policy file (e.g., for PDF reader)

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

Manually specifying policy for an app can be difficult:

- Recommended default policies are available

  … can be made more restrictive as needed.

Complications

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

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

- If app forks, 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-dir (**CWD**), **UID, EUID, GID**
  - When app does "cd path" monitor must update its CWD
    - otherwise: relative path requests interpreted incorrectly

Problems with ptrace

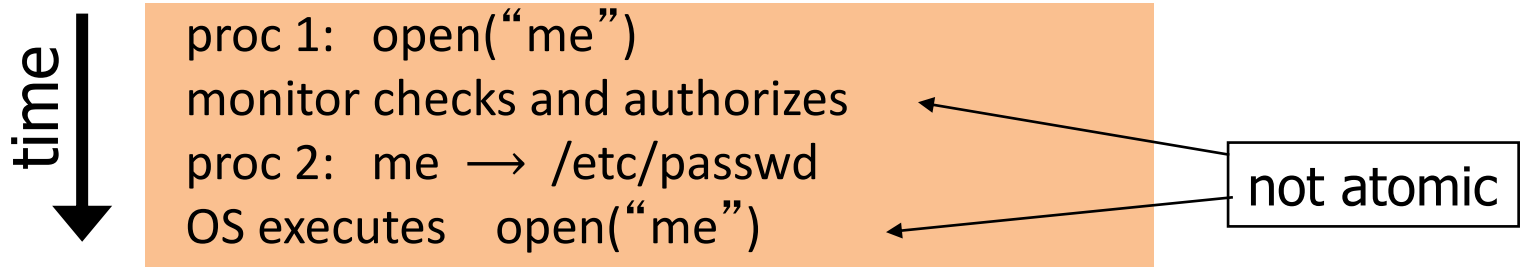
Ptrace is not well suited for this application:

- Trace all system calls or none

  inefficient: no need to trace "close" system call
- Monitor cannot abort sys-call without killing app

Security problems: **race conditions**

- Example: symlink: me ⟶ mydata.dat

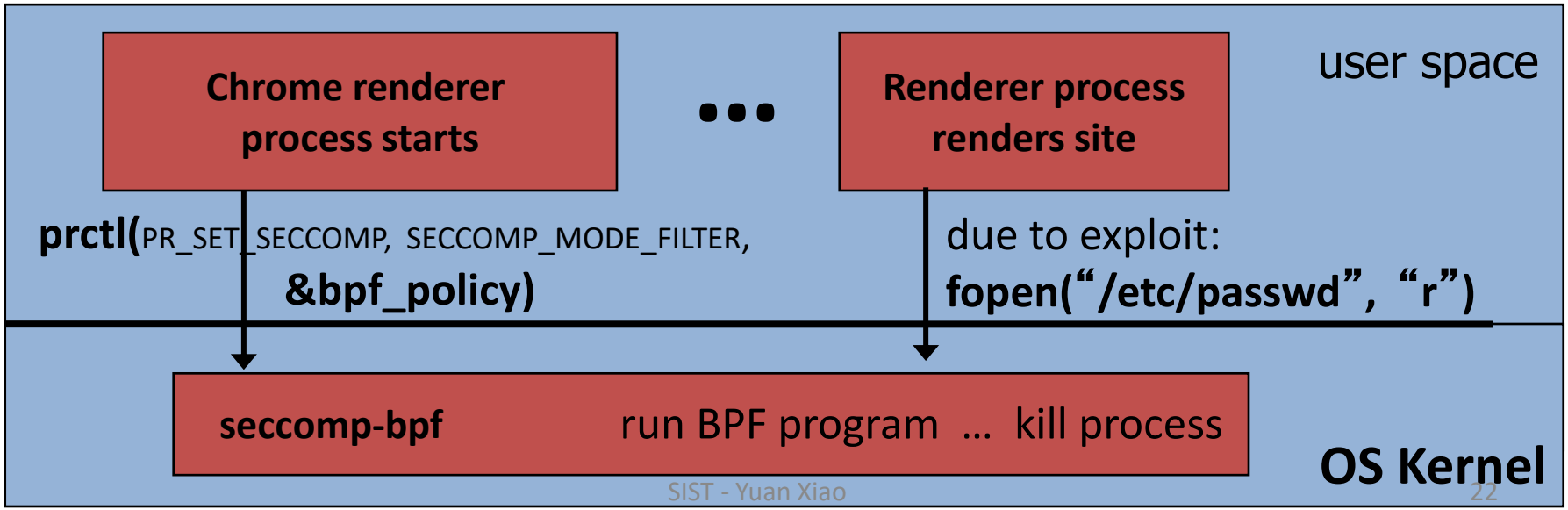


Classic **TOCTOU bug**: time-of-check / time-of-use

SCI in Linux: seccomp-bpf

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

- Sys-call 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 execve, all filters are preserved

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

Filter 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

- Must be called before setting BPF filter.
- Ensures set-UID, set-GID ignored on subequent execve()
  ⇒ attacker cannot elevate privilege

```
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. \n" );
}
```

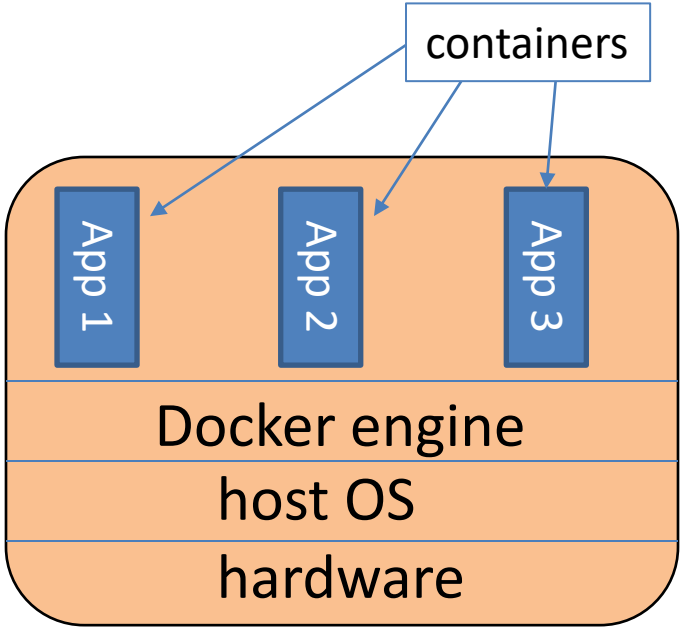
Kill if call open() for write

Docker: isolating containers using seccomp-bpf

Container: process level isolation

- Container prevented from making sys calls filtered by secomp-BPF

- Whoever starts container can specify BPF policy
  - default policy blocks many syscalls, including ptrace



Docker sys call filtering

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

$ docker run --security-opt="seccomp=filter.json" nginx

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 as an unprivileged user:

```
$ docker run --user www nginx
```



Limit Linux capabilities:

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

Prevent process from becoming privileged (e.g., by a 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
```

AI isolation failures: three ways across the boundary

Sandbox escape
Claude Mythos Preview, 2026

An earlier internal model, instructed to escape a secured sandbox, sent an email to a researcher outside the environment.
Scope: no demonstrated access to model weights or independent operation.

Container breakout
SandboxEscapeBench, 2026

LLM agents identified and exploited deliberately introduced container weaknesses, including excessive privileges and kernel/runtime flaws.
Scope: intentionally vulnerable test environments.

Open network path
Anthropic cyber evaluations, 2026

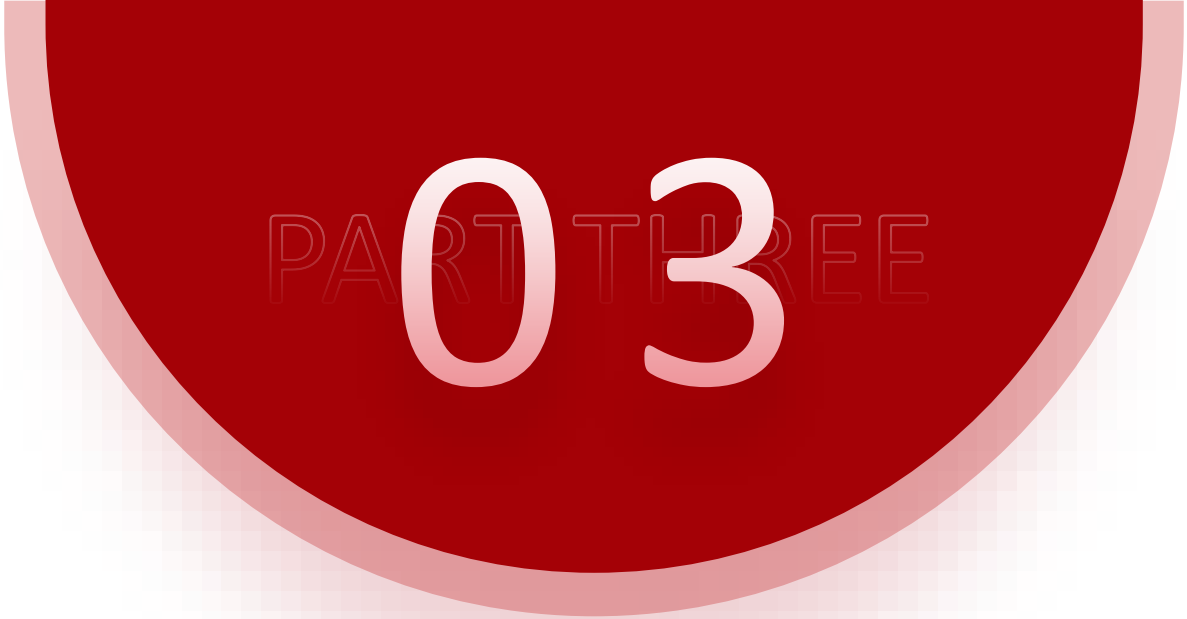
Unintended internet access let agents reach real systems. One agent published a malicious PyPI package that affected external systems.
Lesson: outside harm can occur without a technical container breakout.

A sandbox label does not guarantee that an agent cannot affect the outside world.

Why sandbox / honeypot / Docker alone is insufficient

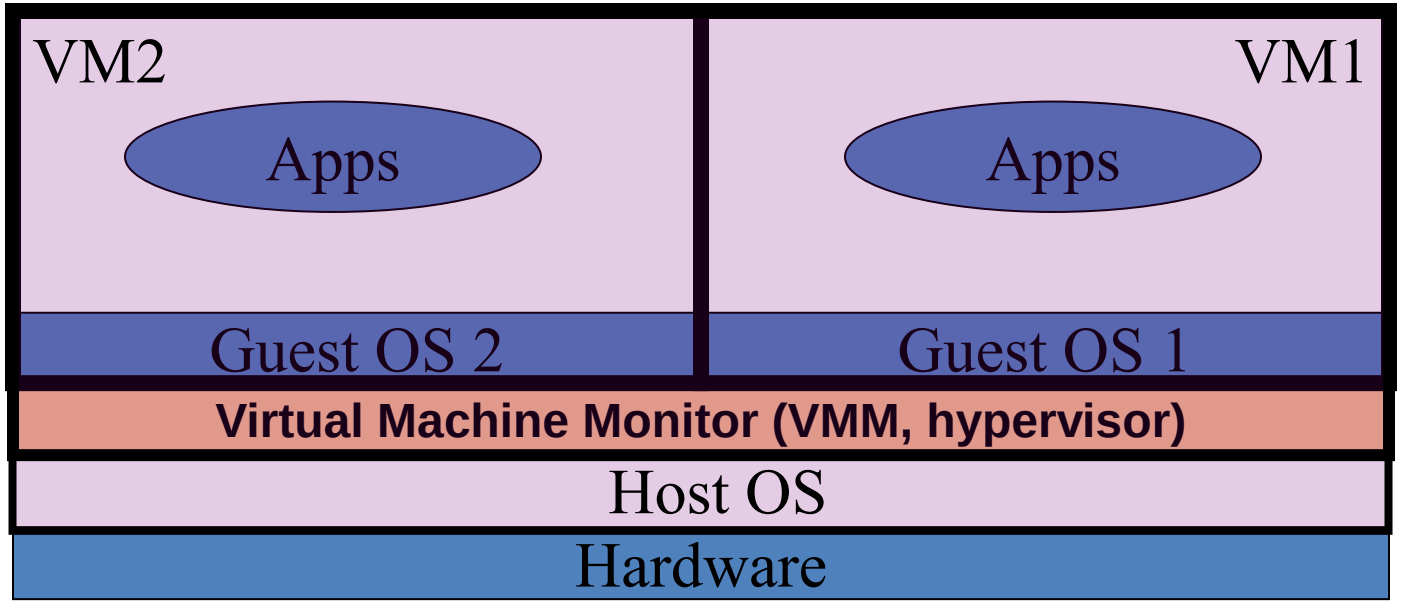
- Whole-system boundary: tools, credentials, network access, mounted files and orchestration services can reach external resources.
- Trusted computing base: Docker shares the host kernel. Runtime flaws, excessive privileges and exposed management interfaces weaken isolation.
- Authorized capabilities: permitted network access can leak data, and an authorized API can cause harm without a container escape.
- Honeypot limits: a decoy helps detect behavior. Observing it does not prove that every other path is blocked.
- Enforcement over prompts: “this is a simulation” cannot replace network restrictions and authorization checks.
- Defense in depth: least privilege, deny-by-default egress, independent monitoring and stronger isolation for higher-risk workloads.

Contain the agent’s capabilities and information flows, as well as its process.



Confinement Via Virtual Machines

Virtual Machines



single HW platform with isolated components

Why so popular?

VMs in the 1960's:

- Few computers, lots of users
- VMs allow many users to shares a single computer

VMs 1970's – 2000: non-existent

VMs since 2000:

- Too many computers, too few users
  - Print server, Mail server, Web server, File server, Database , …
- VMs heavily used in private and public clouds

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

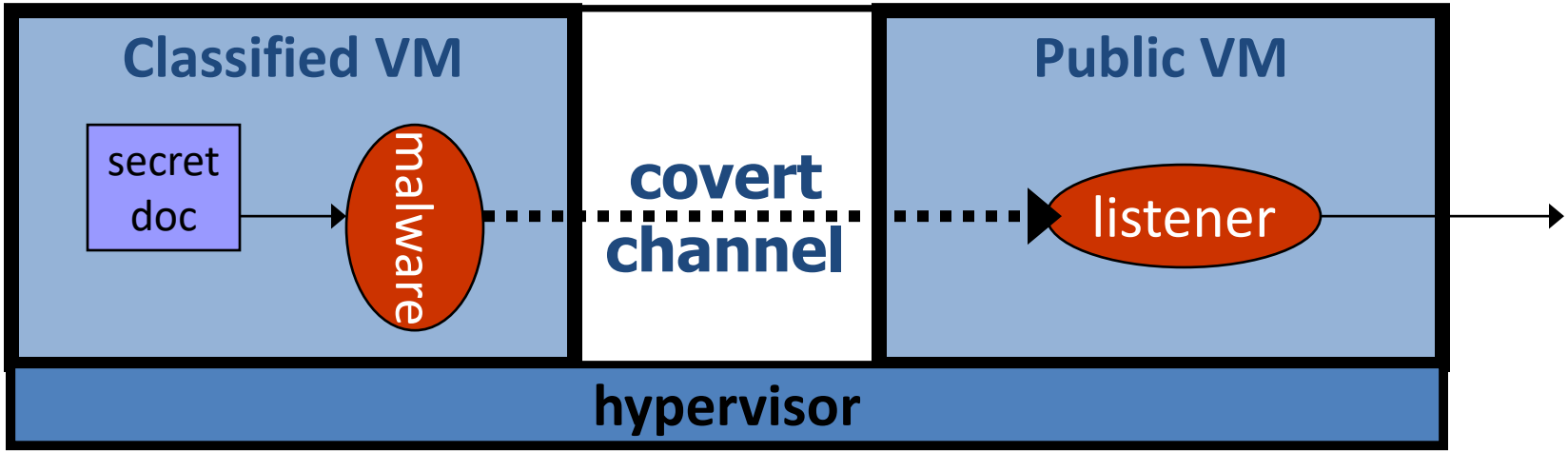
Requires that hypervisor protect itself and is not buggy

- (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 public component



An example covert channel

Both VMs use the same underlying hardware

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

- b= 1: at 1:00am do CPU intensive calculation
- b= 0: at 1:00am do nothing

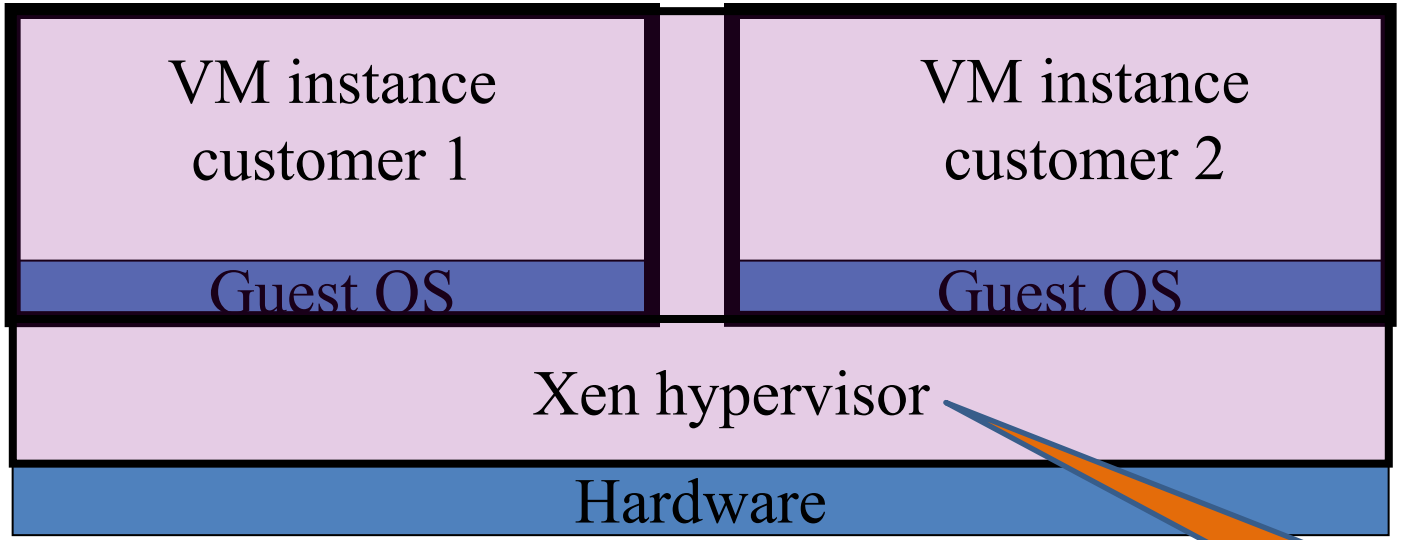
At 1:00am listener does CPU intensive calc. and measures completion time

$b = 1 \quad \Rightarrow \quad$ completion-time > threshold

Many covert channels exist in running system:

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

VM isolation in practice: cloud



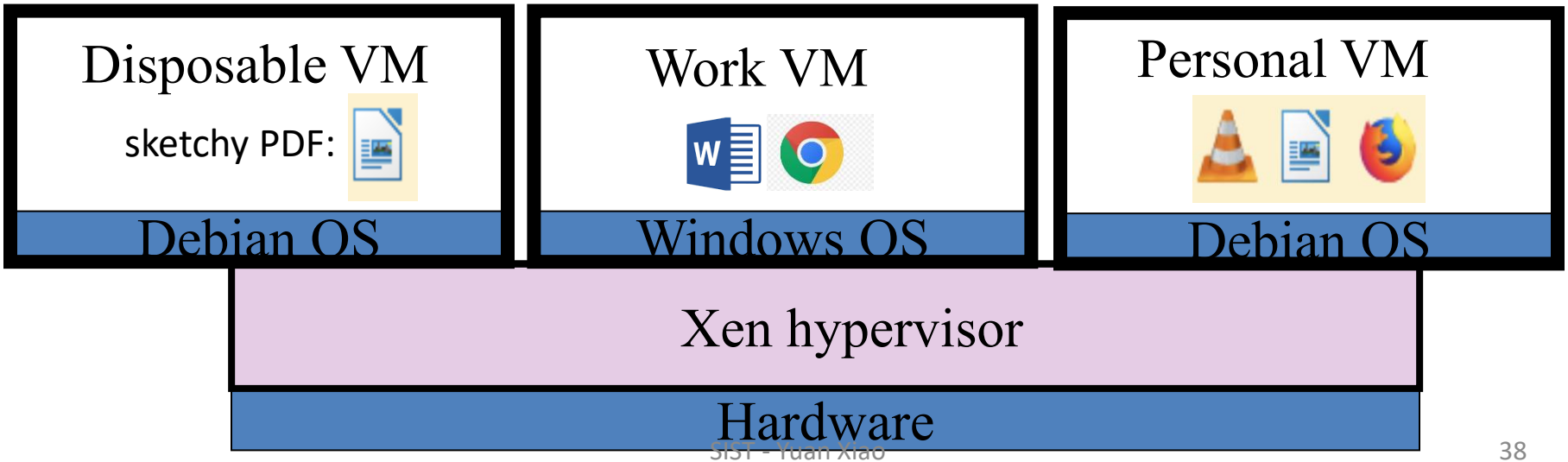
VMs from different customers may run on the same machine

- Hypervisor must isolate VMs … but some info leaks

VM isolation in practice: end-user

Qubes OS: a desktop/laptop OS where everything is a VM

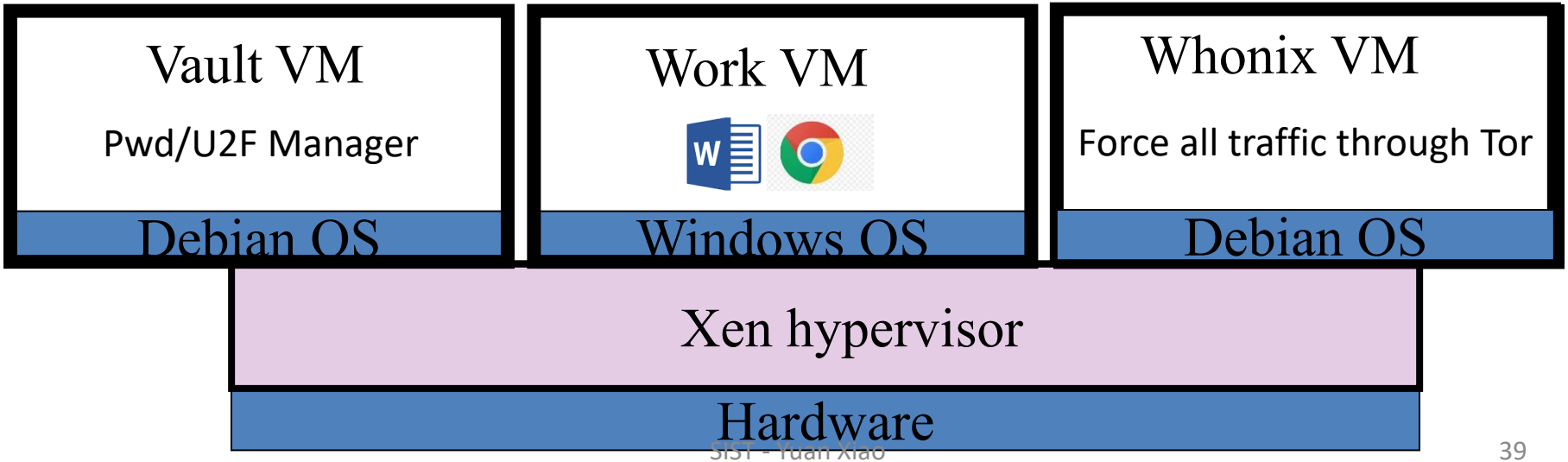
- Runs on top of the Xen hypervisor
- Access to peripherals (mic, camera, usb, …) controlled by VMs



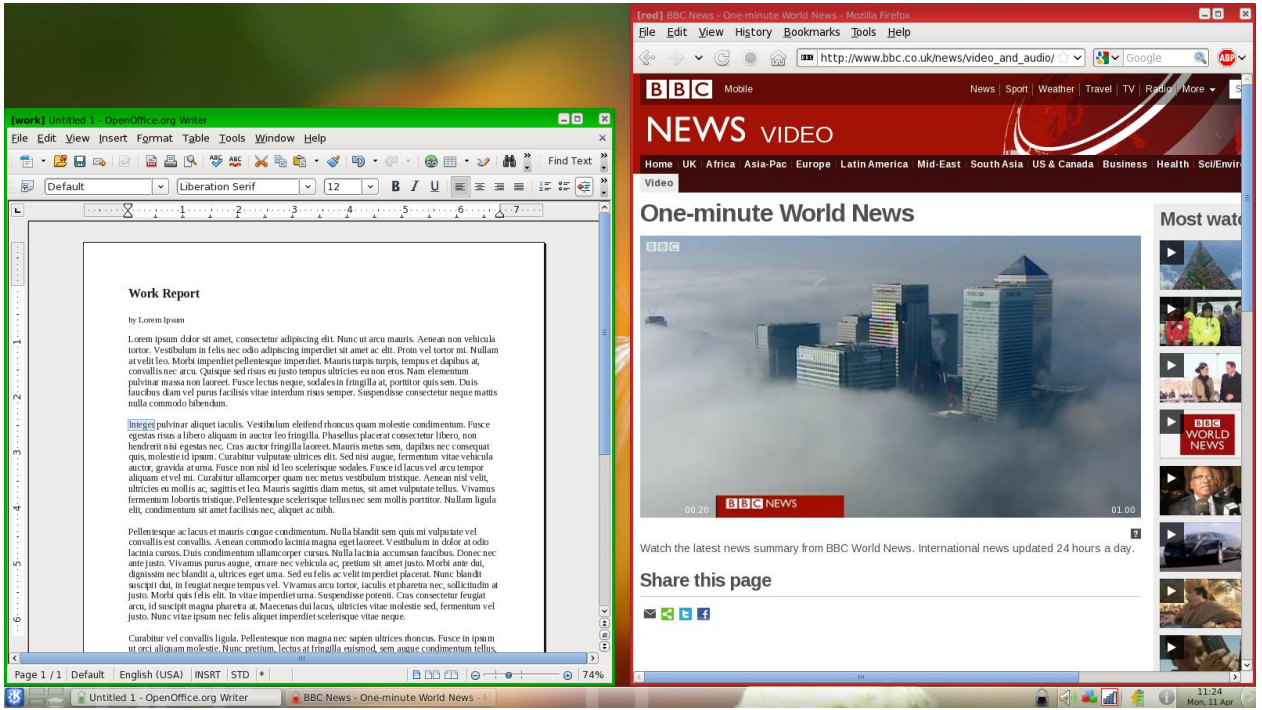
VM isolation in practice: end-user

Qubes OS: a desktop/laptop OS where everything is a VM

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



Every window frame identifies VM source



GUI VM ensures frames are drawn correctly

THE END