

EECS 388



Introduction to Computer Security

Lecture 18:

Access Control and Isolation

Oct. 31, 2024

Prof. Chen



How do we *control access to resources*?

Security Model

System abstraction that enables us to discuss and formulate a policy.

Security Policy

Specifies what actions each subject is allowed to perform on each object.

Security Mechanism

Implementation of the policy. E.g., OS kernel, encryption

What we aim for: **Principle of Least Privilege:** Every program and user should operate using the least *privileges* necessary to complete their job.

- Limits the damage that can result from an accident or error.
- Reduces interactions among privileged programs to minimum for correct operation. Unintentional, unwanted, or improper uses of privilege less likely.
- Minimizes the number of programs that must be audited.
- Example: Classified information concept of “need-to know”.

Models and Policies



Security Model

System abstraction that enables us to discuss and formulate a policy. Consists of:

Subjects (who)

UNIX: users

Android: apps

Web: origins

Objects (what)

UNIX: files, processes

Other: db tables, cookies, device sensors, etc.

Operations

Ways that subjects can operate on (e.g., read, write, call) objects

Security Policy

Defines an **access control matrix** that maps subjects and objects to allowed operations.

		Objects			
		File 1	File 2	File 3	File 4
Subjects	Alice	read	read/ write	no access	no access
	Bob	read	read/ write	no access	no access
	Carol	read	write	read/ write	read/ write
	Wendy	read/ write	read/ write	read	read

Implementing Access Control



What we aim for:

Principle of Complete Mediation

Every access to every object must be checked for authority by a mediator.

- Primary underpinning of access control protection systems.
- Implies that a foolproof method of identifying the source of every request must be devised.
- Be careful of caching checks!
If change in authority occurs, cached results must be updated.

Time-of-check, time-of-use vulnerabilities.



Unix Namespace Security Model



Subjects (Who?)

Users

Objects (What?)

Files, directories,
processes

Access Operations

Read, Write, Execute

Unix Security Model: Users and Groups



Subjects (Who?)

Users

Every user has a unique **user name** and numeric user id

```
user@desktop:~$ whoami; id -u  
user  
1000
```

Objects (What?)

Files, directories,
processes

A user may belong to several **groups**. Provides **role-based access control**

```
user@desktop:~$ groups  
user adm faculty
```

Access Operations

Read, Write, Execute

Superuser (“root”, id 0) has authority to do anything.
Least privilege: *Only assume superuser role when necessary.*

Users in a specific group (adm) can temporarily **elevate** to root privileges

```
user@desktop:~$ sudo whoami  
[sudo] password for user:  
root
```

Unix Security Model: File Permissions



Subjects (Who?)

Users

Objects (What?)

Files, directories,
processes

Access Operations

Read, Write, Execute

Every file and directory has an owner and group

```
user@desktop:~$ ls -l
total 3
d rwx rwx --- 1 user user 4096 Apr  2 15:56 go
- rw- rw- r-- 1 user user  0 Apr 11 04:15 test.py
d rwx rwx --- 1 user faculty 4096 Dec 28 21:09 courses
```

Permission Bits

Owner

Group

Size

Last Modified

Name

File permissions bits specify what role
can do what to the file:

If user == owner, owner permissions;
else if user in group, group permissions;
else other permissions. (If user is root, *override permissions*)

File owner (or root) can change permissions and ownership

rwX **rwX** **rwX**

Owner

Group

Others

Unix Security Model: Processes



Subjects (Who?)

Users

Every process has an **Effective User ID (EUID)** that determines permissions of that process. Processes typically inherit user and group of their parent process, but can be changed by root

Objects (What?)

Files, directories,
processes

```
user@desktop:~$ ps -eo user,comm
USER  COMMAND
root  init
user  bash
user  ps
```

Access Operations

Read, Write, Execute

login and sshd run as root. Authenticate user, then *change* user id and group id to that of user and execute shell (e.g., bash).

Executable files have a **setuid bit**. If set, process has EUID (and privileges) of file's owner, rather than user that executed it.

E.g., passwd command is owned by root and has setuid bit set

Danger: Any vulnerability in passwd means attacker gets root

Traditional thinking:

How to run bad/untrustworthy programs safely?

- Program from untrusted sites
- Apps exposed to adversarial data
- “Honeypots”

New thinking:

Be skeptical of all programs, isolate to achieve least privilege

General goal:

Confinement: ensure misbehaving process cannot harm rest of system

Can be implemented at many levels:

- System call interposition:
(e.g., AppArmor, SELinux)
- Containers:
isolated user space instances
(e.g., Dockers, Kubernetes)
- Virtual machines:
isolate OSes on a single machine
- Physical isolation:
E.g., separate hardware, air gaps

Key component: **reference monitor**

- **Mediates requests** from applications
 - Implements protection policy
 - Enforces isolation and confinement
- 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

Approach: UNIX chroot system call



chroot “jails”: Simple isolation mechanism provided by UNIX kernel

Early use included guest accounts on FTP sites

To use: (must be root)

```
root:/# chroot /tmp/guest      root dir is now /tmp/guest
root:/# su guest               EUID set to guest
guest:/$ ./myapp
```

In this example, myapp is confined such that it can only access files within /tmp/guest for which guest has permission

open("/etc/passwd", "r") actually opens **/tmp/guest/etc/passwd**

Application cannot access files outside of jail because it cannot even name them!

Many ways to evade chroot isolation



- Create device that lets you access raw disk
Major/minor device number namespace is shared
- Send signals to non-chrooted process
PID namespace is shared
- Bind to privileged ports
Network namespace is shared
- Reboot system
Kernel system call interface is shared

Approach: 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., SE Linux)
- Completely user space (e.g., program shepherding)
- Hybrid (e.g., Systrace)

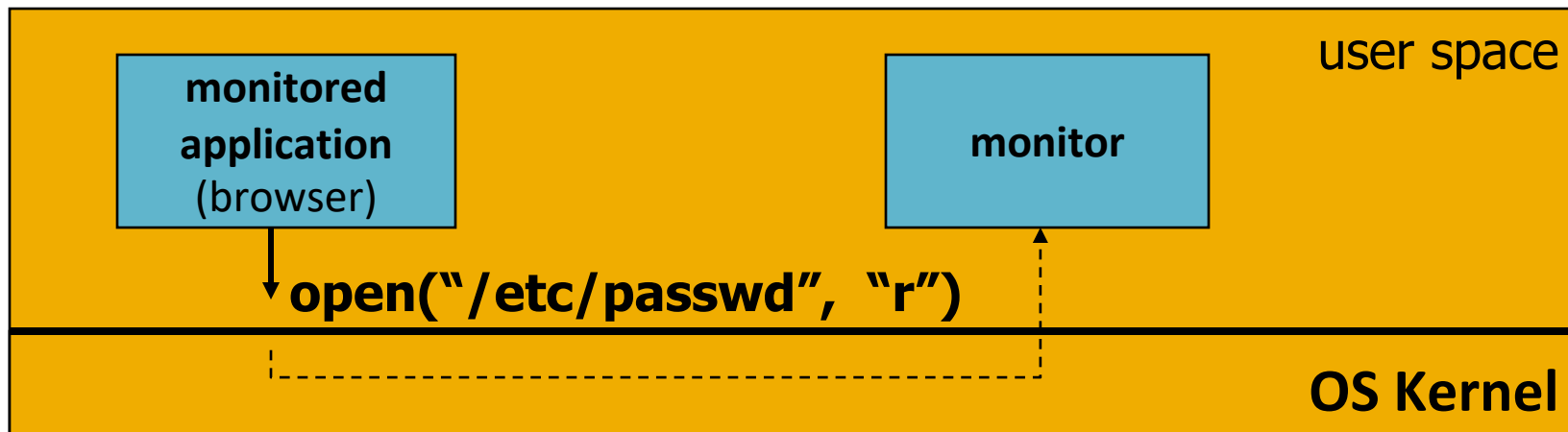
Implementing System Call Interposition



Linux **ptrace**: process tracing

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

and wakes up when **pid** makes a system call.



Monitor checks policy, kills application if request is disallowed.

System Call Interposition Policies



Sample policy file:

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

Manually specifying policy for an app can be difficult:

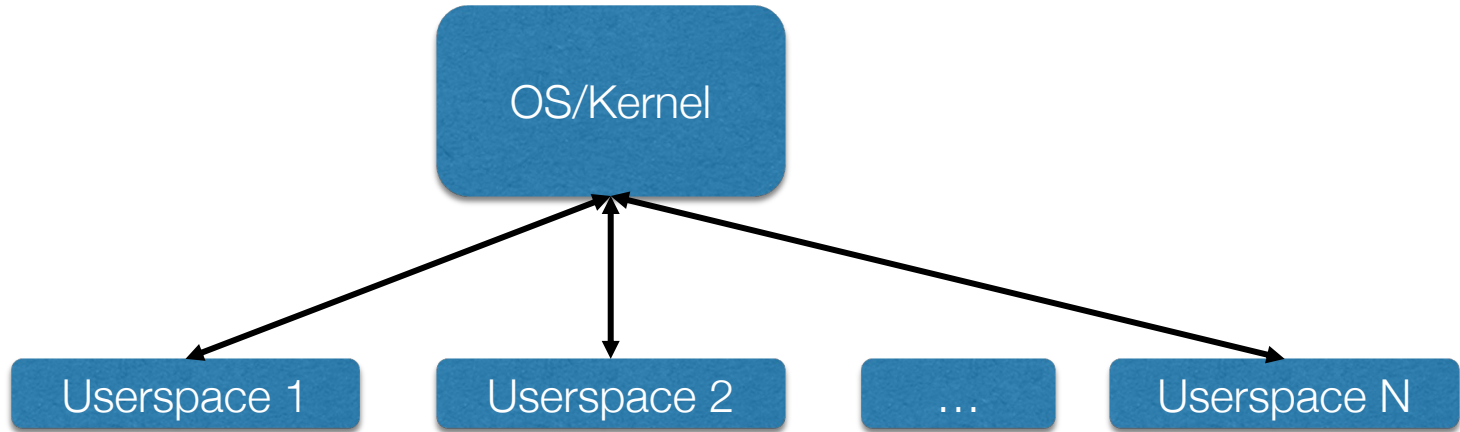
- Can try to auto-generate policy by learning how app behaves on “good” inputs. (But what are “good” inputs?)
- If policy does not cover a specific system call, ask user (But how is the user supposed to decide?)

Difficulty with choosing policy for specific apps (e.g. browser) is the main reason this approach is not widely used

Containers: Further limiting process interference



A virtualization at the level of the operating system which creates multiple isolated userspace instances (e.g. Docker and Kubernetes)



Restrictions: must be same OS/kernel

Cool things: disk quotas, CPU/RAM limits, root privilege isolation, etc...

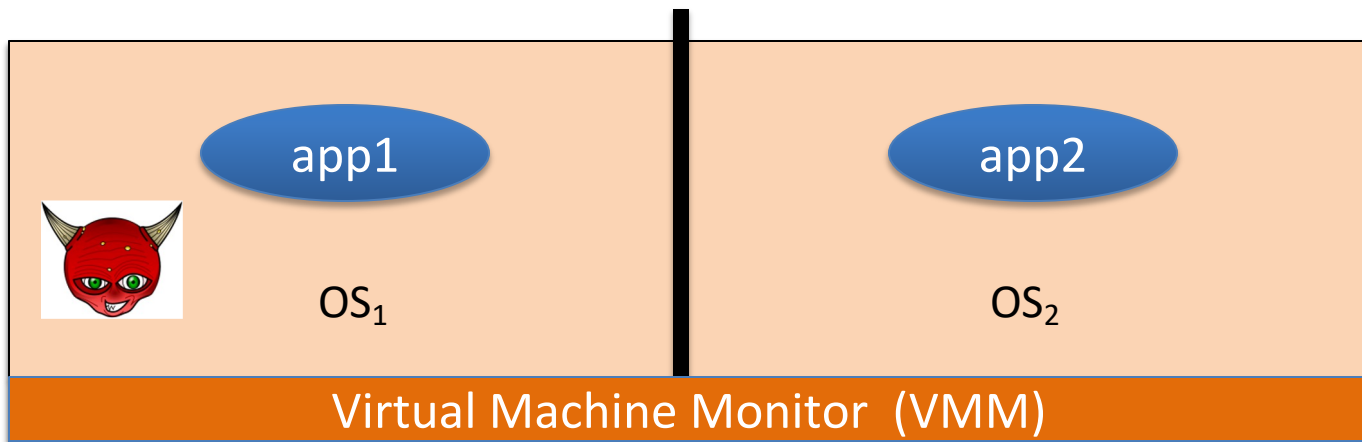
Approach: Virtual Machines



Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

Virtual machines: isolate OSes on a single machine



VMM 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 VMM protect itself and is not buggy

- VMM is much simpler than full OS

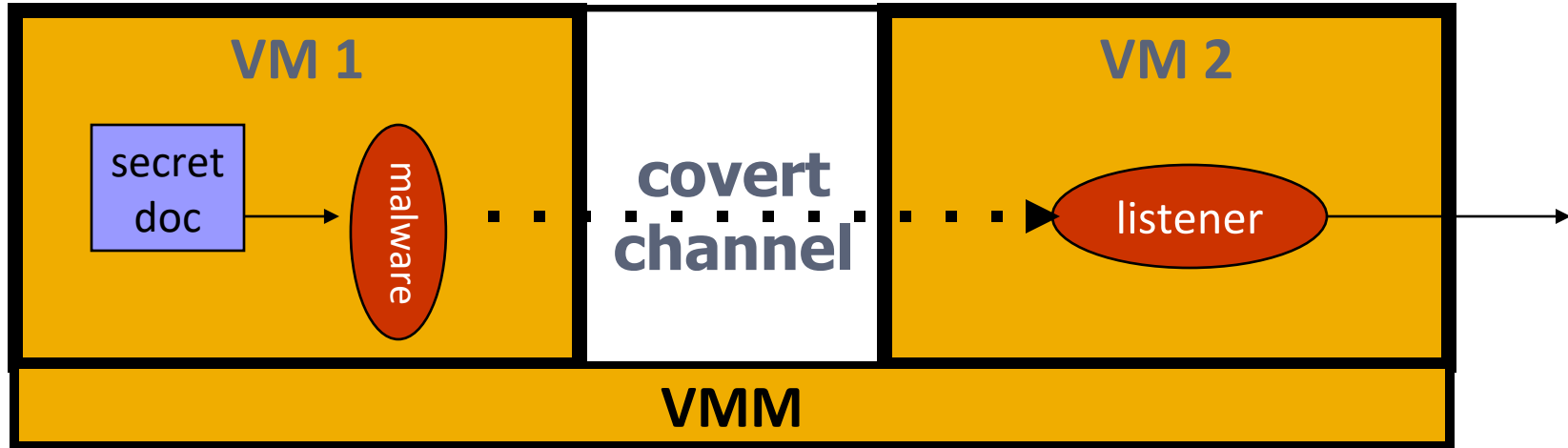
... but device drivers run in Host OS

Problem: Covert Channels



Covert channel: unintended communication channel between isolated components

Can be used to leak classified data from secure component to public component



Covert Channel Example



Both VMs use the same underlying hardware

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

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

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

$b = 1 \Rightarrow \text{completion-time} > \text{threshold}$

Many covert channels exist in running system:

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

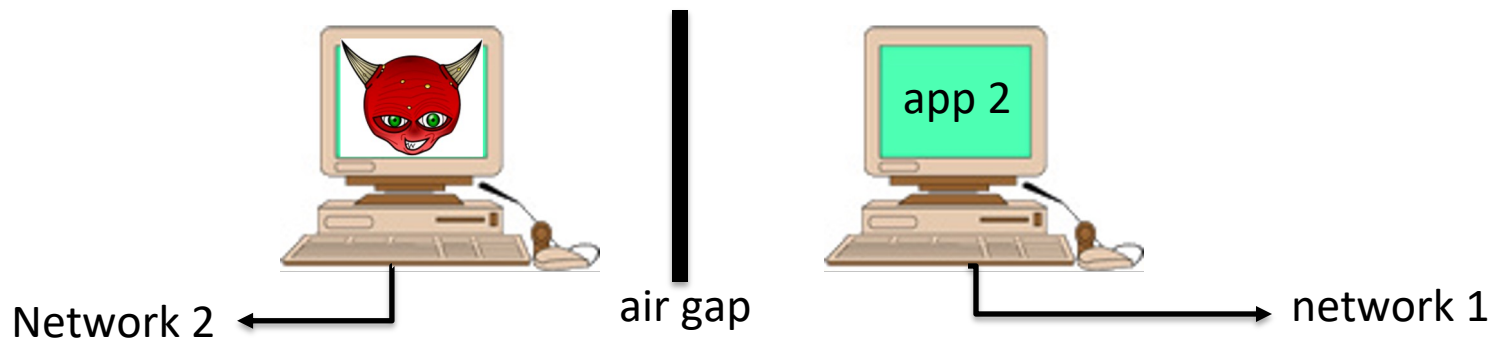
Approach: Physical Air Gap



Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

Hardware: run application on isolated hardware (air gap)



⇒ but restrictive, expensive, difficult to manage

Reminders:

Quiz on Canvas after each lecture

Tuesday November 5
Election security

Coming up

Digital Forensics

Data collection, forensic analysis, anti-forensic techniques

Censorship and Circumvention

Internet censorship, geoblocking, censorship measurement, circumvention