Lecture 2: Security of Program Execution and System Security
Confidentiality and Confinement Principle

Sandeep K. Shukla

Indian Institute of Technology Kanpur

Acknowledgements

- Dan Boneh (Stanford University)
- John C. Mitchell (Stanford University)
- Nicolai Zeldovich (MIT)
- Jungmin Park (Virginia Tech)
- Patrick Schaumont (Virginia Tech)
- C. Edward Chow
- Arun Hodigere
- Web Resources

Lecture 2: Confidentiality, Isolation, confinement

- Module 2.1: Confidentiality Policies
- Module 2.2: Confinement Principle
- Module 2.3: Detour: Unix User IDs, Process IDs and privileges
- Module 2.4: More on Confinement Techniques
- Module 2.5: System Call Interposition

Module 2.1

Confidentiality Policy

Goals of Confidentiality Policies

- Confidentiality Policies emphasize the protection of confidentiality.
- Confidentiality policy also called information flow policy, prevents unauthorized disclosure of information.
- Example: Privacy Act requires that certain personal data be kept confidential. E.g., income tax return info only available to IT department and legal authority with court order. It limits the distribution of documents/info.

Discretionary Access Control (DAC)

- Mechanism where a user can set access control to allow or deny access to an object
- Also called Identity-based access control (IBAC)
- It is a traditional access control techniques implemented by traditional operating system such as Unix.
  - Based on user identity and ownership
  - Programs run by a user inherits all privileges granted to the user.
  - Program is free to change access to the user's objects
  - Support only two major categories of users:
    - Completely trusted admins
    - Completely untrusted ordinary users

Problems with DAC

- Each users has complete discretion over his objects.
  - What is wrong with that?
    - Difficult to enforce a system-wide security policy, e.g.
      - A user can leak classified documents to a unclassified users.
- Only support coarse-grained privileges
  - Too simple classification of users (How about more than two categories of users?)
- Unbounded privilege escalation

Problems with DAC (2)

- Only based on user's identity and ownership, ignoring security relevant info such as
  - User's role
  - Function of the program
  - Trustworthiness of the program
    - Compromised program can change access to the user's objects
    - Compromised program inherit all the permissions granted to the users (especially the root user)
  - Sensitivity of the data
  - Integrity of the data

Mandatory Access Control (MAC)

- Mechanism where system controls access to an object and a user cannot alter that access.
  - Occasionally called rule-based access control?
- Defined by three major properties:
  - Administratively-defined security policy
  - Control over all subjects (process) and objects (files, sockets, network interfaces)
  - Decisions based on all security-relevant info
- MAC access decisions are based on labels that contains security-relevant info.

What Can MAC Offer?

- Supports a wide variety of categories of users in system.
  - For example, Users with labels: (secret, {EUR, US}) (top secret, {NUC, US}).
  - Here security level is specified by the two-tuple: (clearance, category)
- Strong separation of security domains
- System, application, and data integrity
- Ability to limit program privileges
  - Confine the damage caused by flawed or malicious software
- Authorization limits for legitimate users

Module 2.2

Isolation: Confinement Principle

Isolation

The confinement principle

Running untrusted code

We often need to run buggy/untrusted code:

- programs from untrusted Internet sites:
  - apps, extensions, plug-ins, codecs for media player
- exposed applications: pdf viewers, outlook
- legacy daemons: sendmail, bind
- honeypots

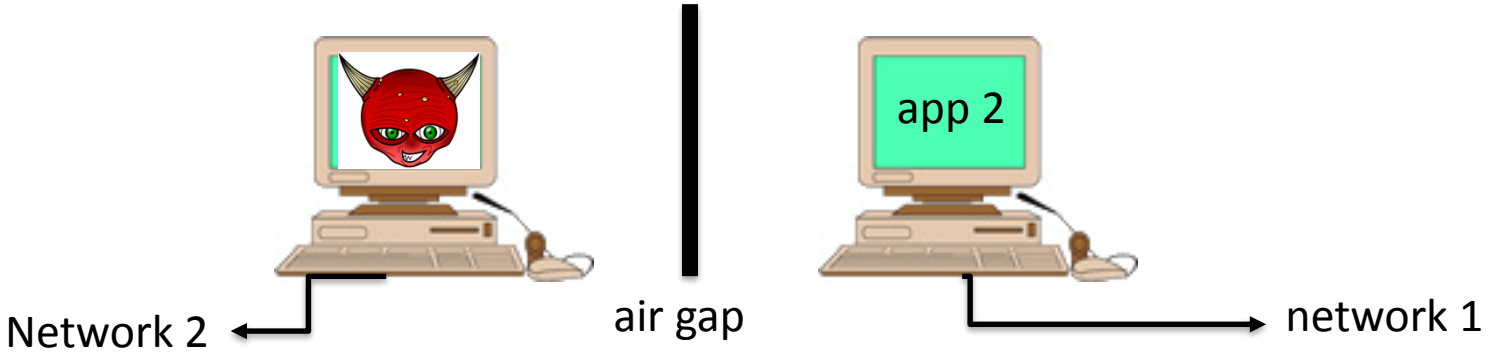
- Goal: if application "misbehaves" ⇒ kill it

Approach:Confinement

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

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



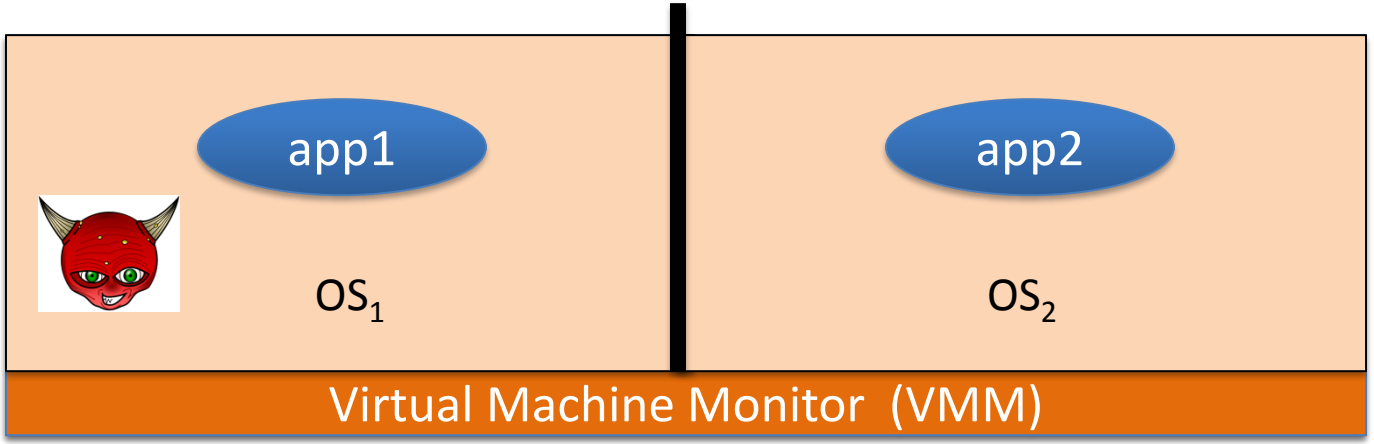
⇒ **Resource Expensive**

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



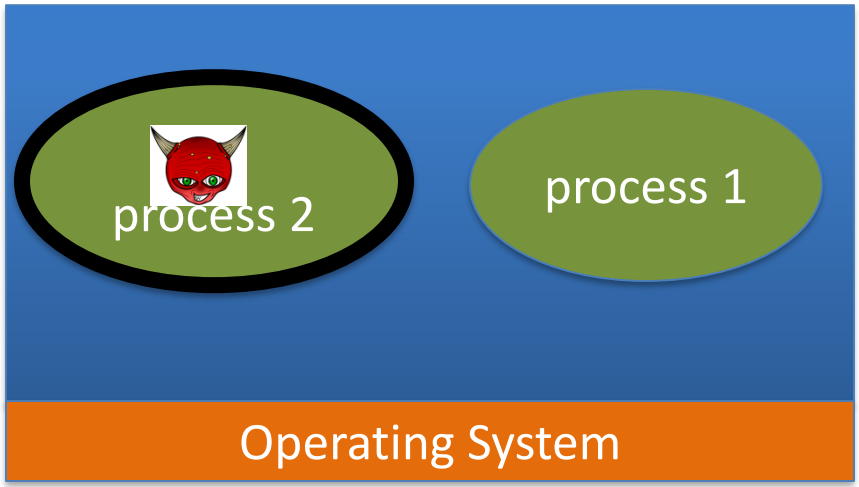
Approach: confinement

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Process:** System Call Interposition

Isolate



User Mode Linux (UML)

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
    - Thread Local Storage (TLS)
- **Application**: e.g. browser-based confinement
  - Discretionary Access Control
    - SOP – Same Origin Policy
    - CSP – Content Security Policy
    - CORS – Cross Origin Resource Sharing
  - Mandatory Access Control
    - COWL – Confinement with Original Web Label

Implementing confinement

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

Module 2.3

Detour into Unix User IDs and IDs of Unix Processes

A Detour

- A few words about Unix User IDs and IDs associated with Unix Processes

Source: CS426 Course at Purdue Univ.

PRINCIPALS AND SUBJECTS

- A subject is a program (application) executing on behalf of some principal(s)
- A principal may at any time be idle, or have one or more subjects executing on its behalf

What are subjects in UNIX?

What are principals in UNIX?

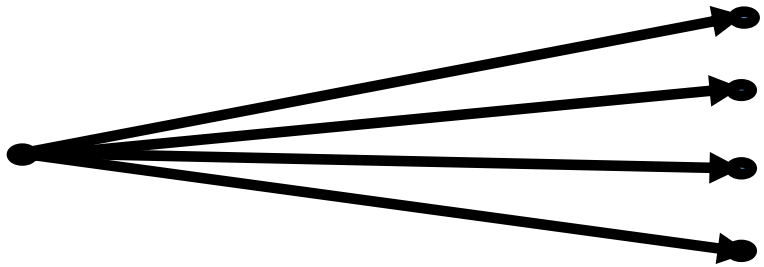
OBJECTS

- An object is anything on which a subject can perform operations (mediated by rights)
- Usually objects are passive, for example:
  - File
  - Directory (or Folder)
  - Memory segment
- But, subjects can also be objects, with operations
  - kill
  - suspend
  - resume

Basic Concepts of UNIX Access Control: Users, Groups, Files, Processes

- Each user account has a unique UID
  - The UID 0 means the super user (system admin)
- A user account belongs to multiple groups
- Subjects are processes
  - associated with uid/gid pairs, e.g., (euid, egid), (ruid, rgid), (suid, sgid)
- Objects are files

USERS AND PRINCIPALS



USERS

Real World User

PRINCIPALS

Unit of Access Control and Authorization

the system authenticates the human user to a particular principal

USERS AND PRINCIPALS

- There should be a one-to-many mapping from users to principals
  - a user may have many principals, but
  - each principal is associated with an unique user

This ensures accountability of a user's actions

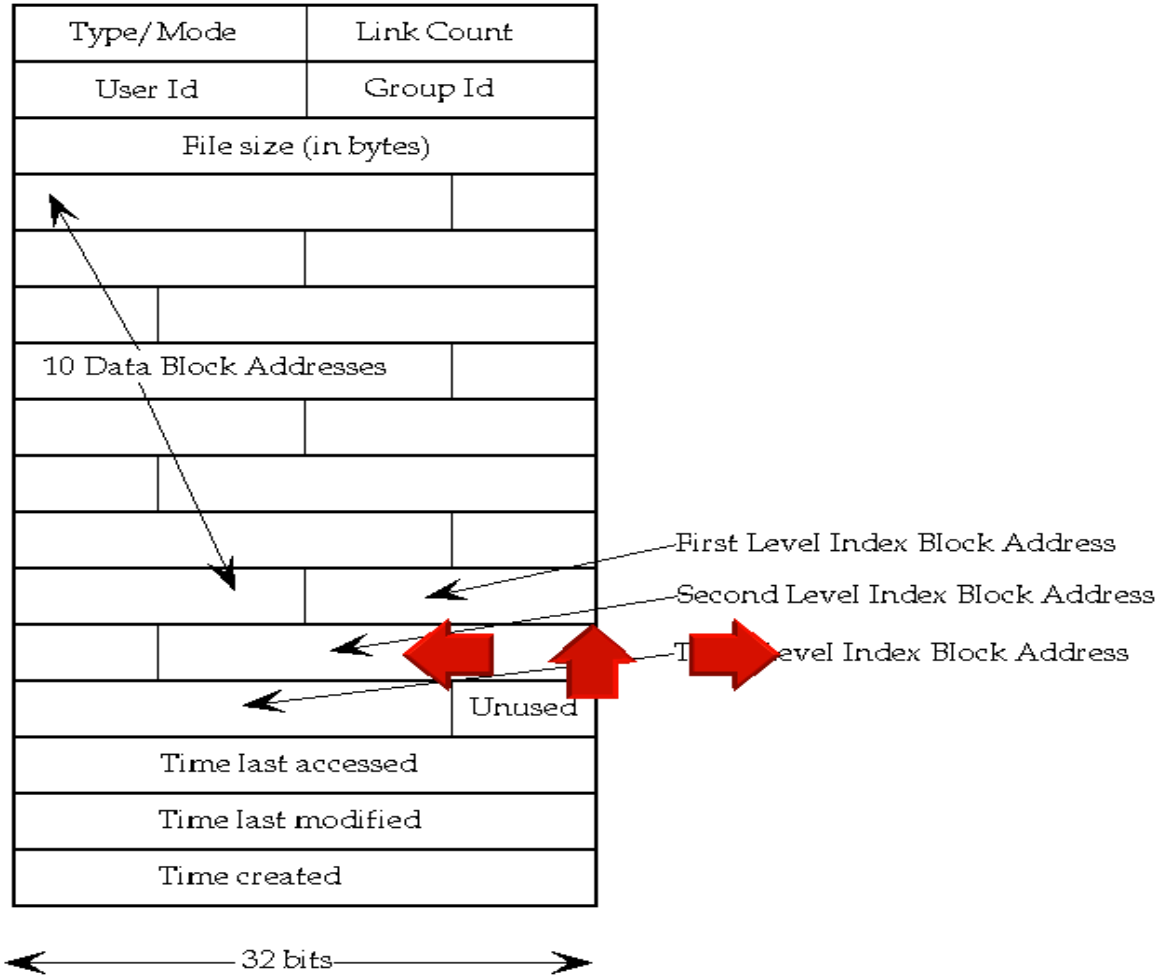
What does the above imply in UNIX?

Organization of Objects

- Almost all objects are modeled as files
  - Files are arranged in a hierarchy
  - Files exist in directories
  - Directories are also one kind of files
- Each object has
  - owner
  - group
  - 12 permission bits
    - rwx for owner, rwx for group, and rwx for others
    - suid, sgid, sticky

UNIX inodes:

Each file corresponds to an inode



Basic Permissions Bits on Files (Non-directories)

- Read controls reading the content of a file
  - i.e., the read system call
- Write controls changing the content of a file
  - i.e., the write system call
- Execute controls loading the file in memory and execute
  - i.e., the execve system call

Execution of a file

- Binary file vs. script file
- Having execute but not read, can one run a binary file?
- Having execute but not read, can one run a script file?
- Having read but not execute, can one run a script file?

Permission Bits on Directories

- Read bit allows one to show file names in a directory
- The execution bit controls traversing a directory
  - does a lookup, allows one to find inode # from file name
  - chdir to a directory requires execution
- Write + execution control creating/deleting files in the directory
  - Deleting a file under a directory requires no permission on the file
- Accessing a file identified by a path name requires execution to all directories along the path

The suid, sgid, sticky bits

	suid	sgid	sticky bit
non-executable files	no effect	affect locking (unimportant for us)	not used anymore
executable files	change euid when executing the file	change egid when executing the file	not used anymore
directories	no effect	new files inherit group of the directory	only the owner of a file can delete

Some Examples

- What permissions are needed to access a file/directory?
  - read a file: /d1/d2/f3
  - write a file: /d1/d2/f3
  - delete a file: /d1/d2/f3
  - rename a file: from /d1/d2/f3 to /d1/d2/f4
  - …
- File/Directory Access Control is by System Calls
  - e.g., open(2), stat(2), read(2), write(2), chmod(2), opendir(2), readdir(2), readlink(2), chdir(2), …

The Three sets of permission bits

- Intuition:
  - if the user is the owner of a file, then the r/w/x bits for owner apply
  - otherwise, if the user belongs to the group the file belongs to, then the r/w/x bits for group apply
  - otherwise, the r/w/x bits for others apply

- Can one implement negative authorization, i.e., only members of a particular group are not allowed to access a file?

Other Issues On Objects in UNIX

- Accesses other than read/write/execute
  - Who can change the permission bits?
    - The owner can
  - Who can change the owner?
    - Only the superuser
- Rights not related to a file
  - Affecting another process
  - Operations such as shutting down the system, mounting a new file system, listening on a low port
    - traditionally reserved for the root user

Subjects vs. Principals

- Access rights are specified for users (accounts)
- Accesses are performed by processes (subjects)
- The OS needs to know on which users' behalf a process is executing

Process User ID Model in Modern UNIX Systems

- Each process has three user IDs
  - real user ID (ruid) owner of the process
  - effective user ID (euid) used in most access control decisions
  - saved user ID (suid)
- and three group IDs
  - real group ID
  - effective group ID
  - saved group ID

Process User ID Model in Modern UNIX Systems

- When a process is created by *fork*
  - it inherits all three users IDs from its parent process
- When a process executes a file by *exec*
  - it keeps its three user IDs unless the set-user-ID bit of the file is set, in which case the effective uid and saved uid are assigned the user ID of the owner of the file
- A process may change the user ids via system calls

The Need for suid/sgid Bits

- Some operations are not modeled as files and require user id = 0
  - halting the system
  - bind/listen on “privileged ports” (TCP/UDP ports below 1024)
  - non-root users need these privileges
- File level access control is not fine-grained enough
- System integrity requires more than controlling who can write, but also how it is written

Security Problems of Programs with suid/sgid

- These programs are typically setuid root
- Violates the least privilege principle
  - every program and every user should operate using the least privilege necessary to complete the job
- Why violating least privilege is bad?
- How would an attacker exploit this problem?
- How to solve this problem?

Changing effective user IDs

- A process that executes a set-uid program can drop its privilege; it can
  - drop privilege permanently
    - removes the privileged user id from all three user IDs
  - drop privilege temporarily
    - removes the privileged user ID from its effective uid but stores it in its saved uid, later the process may restore privilege by restoring privileged user ID in its effective uid

Access Control in Early UNIX

- A process has two user IDs: real uid and effective uid and one system call setuid
- The system call setuid(id)
  - when euid is 0, setuid set both the ruid and the euid to the parameter
  - otherwise, the setuid could only set effective uid to real uid
    - Permanently drops privileges
- A process cannot temporarily drop privilege

Ref: Setuid Demystified, In USENIX Security ‘ 02

System V

- Added saved uid & a new system call
- The system call seteuid
  - if euid is 0, seteuid could set euid to any user ID
  - otherwise, could set euid to ruid or suid
    - Setting to ruid temporarily drops privilege
- The system call setuid is also changed
  - if euid is 0, setuid functions as seteuid
  - otherwise, setuid sets all three user IDs to real uid

BSD

- Uses ruid & euid, change the system call from setuid to setreuid
  - if euid is 0, then the ruid and euid could be set to any user ID
  - otherwise, either the ruid or the euid could be set to value of the other one
    - enables a process to swap ruid & euid

Modern UNIX

- System V & BSD affect each other, both implemented setuid, seteuid, setreuid, with different semantics
  - some modern UNIX introduced setresuid

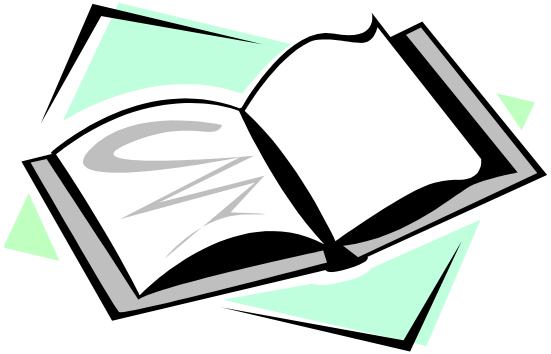
- Things get messy, complicated, inconsistent, and buggy
  - POSIX standard, Solaris, FreeBSD, Linux

Suggested Improved API

- Three method calls
  - drop_priv_temp
  - drop_priv_perm
  - restore_priv
- Lessons from this?
- Psychological acceptability principle
  - “human interface should be designed for ease of use”
  - the user’s mental image of his protection goals should match the mechanism

Readings for This Lecture

- Wiki
  - Filesystem Permissions
- Other readings
  - UNIX File and Directory Permissions and Modes
    - http://www.hccfl.edu/pollock/AUnix1/FilePermissions.htm
  - Unix file permissions
    - http://www.unix.com/tips-tutorials/19060-unix-file-permissions.html



Module 2.4

More on Confinement Techniques

An old example: chroot

Often used for “guest” accounts on ftp sites

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 file system accesses for applications in jail

open(“/etc/passwd”, “r”) => open(“/tmp/guest/etc/passwd”, “r”)

⇒ application cannot access files outside of jail

Jailkit

Problem: all utility progs (ls, ps, vi) must live inside jail

- **jailkit** project: auto builds files, libs, and dirs needed in jail env
  - **jk_init**: creates jail environment
  - **jk_check:** checks jail env for security problems
    - checks for any modified programs,
    - checks for world writable directories, etc.
  - **jk_lsh**: restricted shell to be used inside jail

```
$ wget http://olivier.sessink.nl/jailkit/jailkit-2.19.tar.gz
$ gunzip jailkit-2.19.tar.gz
$ tar xvf jailkit-2.19.tar
$ cd jailkit-2.19
$./configure
$ make
$ make install
```

note: simple chroot jail does not limit network access

Escaping from jails

Early escapes: relative paths

open(“../../etc/passwd”, “r”) ⇒

open(“/tmp/guest/../../etc/passwd”, “r”)

---

chroot should only be executable by root.

- otherwise jailed app can do:
  - create dummy file “/aaa/etc/passwd”
  - echo root::0:0::/:/bin/sh > /aaa/etc/passwd
  - mkdir /aaa/bin
  - cp /bin/sh /aaa/bin
  - **run chroot “/aaa”**
  - run su root to become root

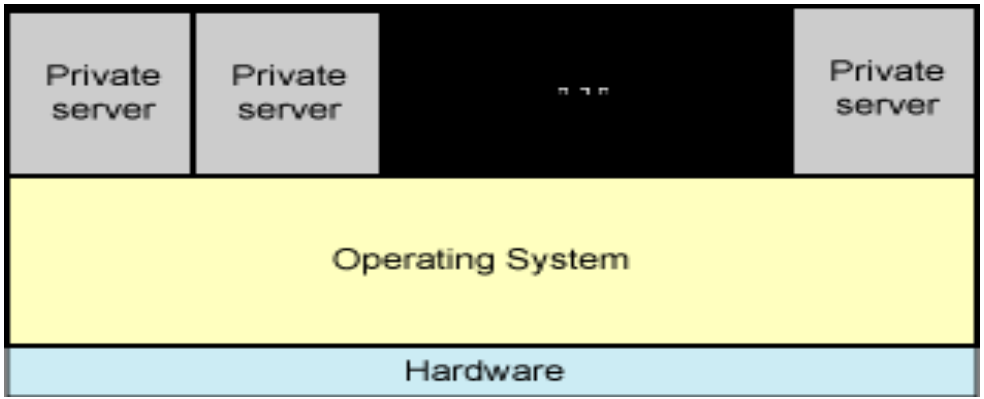
(bug in Ultrix 4.0)

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

- Where chroot jail was weak partitioning, FreeBSD jail is strong partitioning
  - Create virtual machines.
- Popularly used in ISPs. Check out
  http://www.onlamp.com/pub/a/bsd/2003/09/04/jails.html



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

Virtual OS on the Desktop

- VMWare, Parallels, KVM, Xen
- Different approaches
  - Hardware emulation
    - Emulation is where software is used to simulate hardware for a guest operating system to run in. This has been used in the past but is difficult to do and offers low performance.
  - Native virtualization (with hardware)
    - Native virtualization (or full virtualization) is where a type-2 hypervisor is used to partially allow access to the hardware and partially to simulate hardware in order to allow you to load a full operating system. This is used by emulation packages like VMware Server, Workstation, Virtual PC, and Virtual Server, Oracle Virtual Box
  - Paravirtualization
    - Paravirtualization is where the guest operating systems run on the hypervisor, allowing for higher performance and efficiency. For more technical information and videos on this topic, visit VMware's Technology Preview for Transparent Virtualization. Examples of paravirtualization are Microsoft Hyper-V and VMware ESX Server.

Not all programs can run in a jail

Programs that can run in jail:

- audio player
- web server

Programs that cannot:

- web browser
- mail client

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

Module 2.5

System Call Interposition

Isolation

System Call Interposition

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:

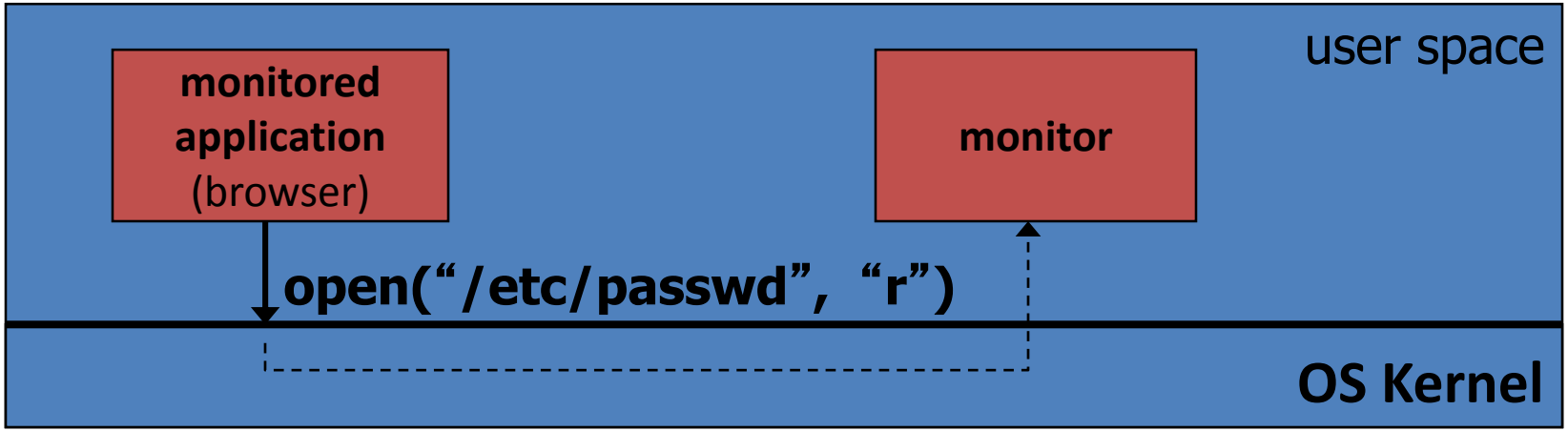
- Completely kernel space (e.g. GSWTK) -- security extension architecture
- Completely user space (e.g. program shepherding)
- Hybrid (e.g. Systrace)

Initial 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

Complications

- 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

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

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

Problems with ptrace

Ptrace is not well suited for this application:

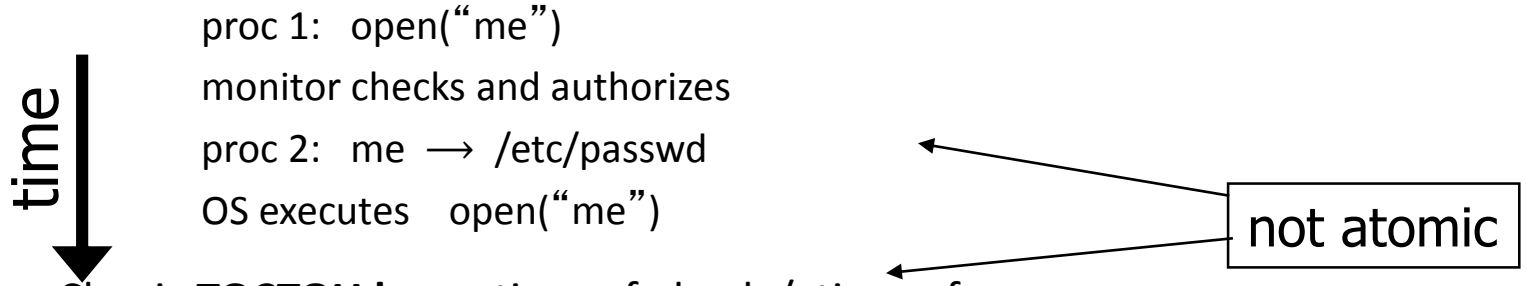
- Trace all system calls or none

  inefficient: no need to trace "close" system call

- Monitor cannot abort sys-call without killing app

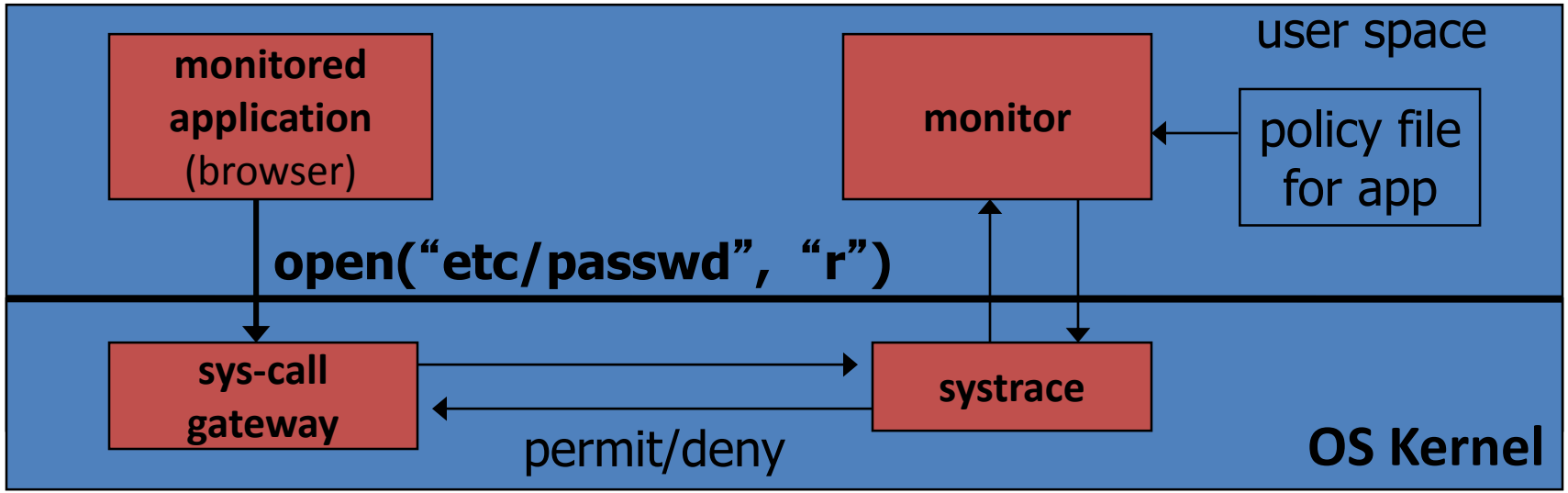
Security problems: **race conditions**

- Example: symlink: me ⟶ mydata.dat



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

Alternate design: systrace [P'02]



- systrace only forwards monitored sys-calls to monitor (efficiency)
- systrace resolves sym-links and replaces sys-call path arguments by full path to target
- When app calls execve, monitor loads new policy file

Systrace Policies

• Systrace executes the application without privileges and only elevates them to the desired level when required.
  • native-socket: sockdom eq "AF_INET" and socktype eq "SOCK_RAW" then permit as root
  • native-bind: sockaddr eq "inet-[0.0.0.0]:22" then permit as root
  • native-fsread: filename eq "/dev/kmem" then permit as :kmem

A simple policy for the *ls* binary. If *ls* attempts to list files in */etc*, Systrace disallows the access and */etc* does not seem to exist. Listing the contents of */tmp* works normally, but trying to *ls /var* generates a warning.

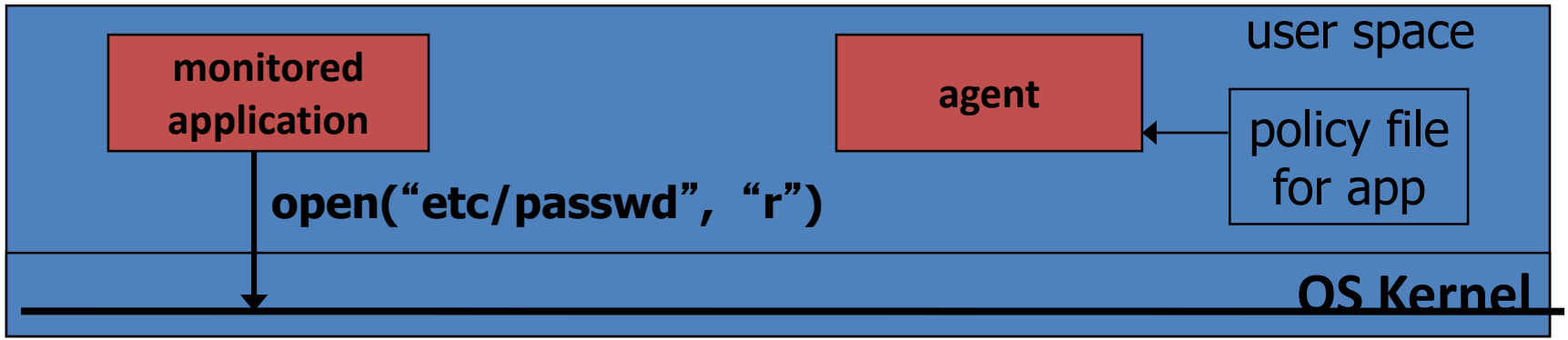
```
Policy: /bin/ls, Emulation: native native-munmap: permit
[...]           native-stat: permit
                native-fsread: filename match "/usr/*" then permit
                native-fsread: filename eq "/tmp" then permit
                native-fsread: filename eq "/etc" then deny[enotdir]
                native-fchdir: permit
                native-fstat: permit native-fcntl: permit
[...]           native-close: permit
                native-write: permit
                native-exit: permit
```

Ostia: a delegation architecture [GPR'04]

Previous designs use filtering:

- Filter examines sys-calls and decides whether to block
- Difficulty with syncing state between app and monitor (CWD, UID, ..)
  - Incorrect syncing results in security vulnerabilities (e.g. disallowed file opened)

A delegation architecture:



Ostia: a delegation architecture [GPR'04]

- Monitored app disallowed from making monitored sys calls
  - Minimal kernel change (… but app can call **close**() itself)
- Sys-call delegated to an agent that decides if call is allowed
  - Can be done without changing app

    (requires an emulation layer in monitored process)
- Incorrect state syncing will not result in policy violation
- What should agent do when app calls **execve?**
  - Process can make the call directly. Agent loads new policy file.

Policy

Sample policy file:

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

Manually specifying policy for an app can be difficult:

- Systrace can auto-generate policy by learning how app behaves on “good” inputs
- If policy does not cover a specific sys-call, ask user

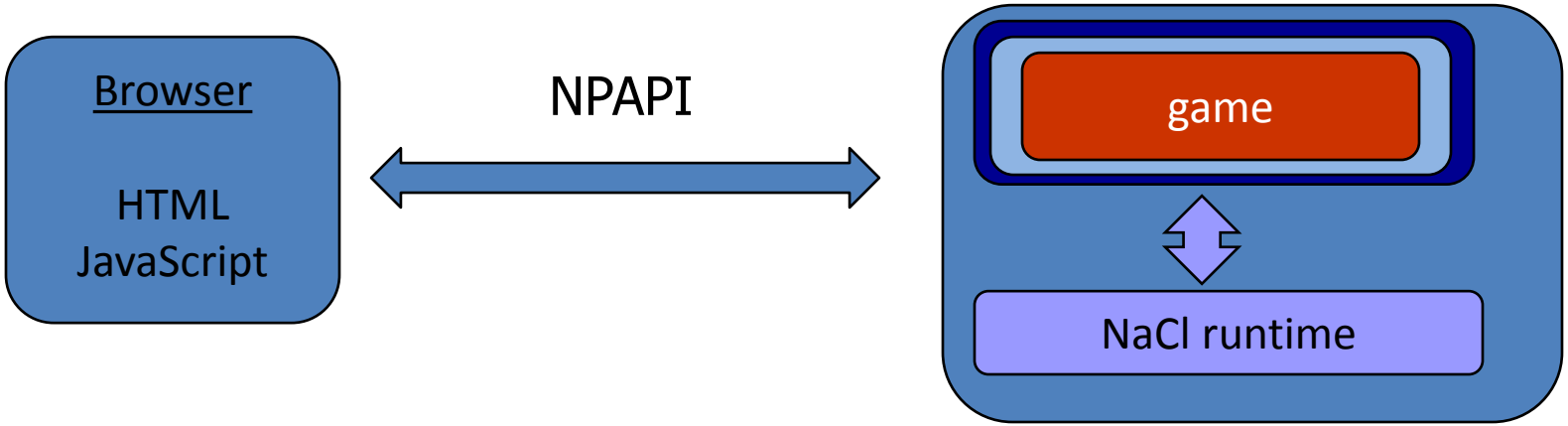
  … but user has no way to decide

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

Program Shepherding

Restricting	Least restrictive			Most restrictive	
Code origins	Any		Dynamically writ- ten code, if self- contained and no system calls	Only code from disk, can be dynamically loaded	Only code from disk, originally loaded
Function returns	Any	Only to after calls	Direct call targeted by only one return	Random `xor` as in StackGhost [14]	Return only from called function
Intra-segment call or jump	Any		Only to function entry points (if have symbol table)		Only to bindings given in an interface list
Inter-segment call or jump	Any		Only to export of tar- get segment	Only to import of source segment	Only to bindings given in an interface list
Indirect calls	Any		Only to address stored in read-only memory	Only within user segment or from library	None
`execve`	Any		Static arguments	Only if the operation can be validated not to cause a problem	None
`open`	Any		Disallow writes to specific files (e.g., `/etc/passwd`)	Only to a subregion of the file system	None

NaCl: a modern day example



- game: untrusted x86 code
- Two sandboxes:
  - outer sandbox: restricts capabilities using system call interposition
  - Inner sandbox: uses x86 memory segmentation to isolate application memory among apps

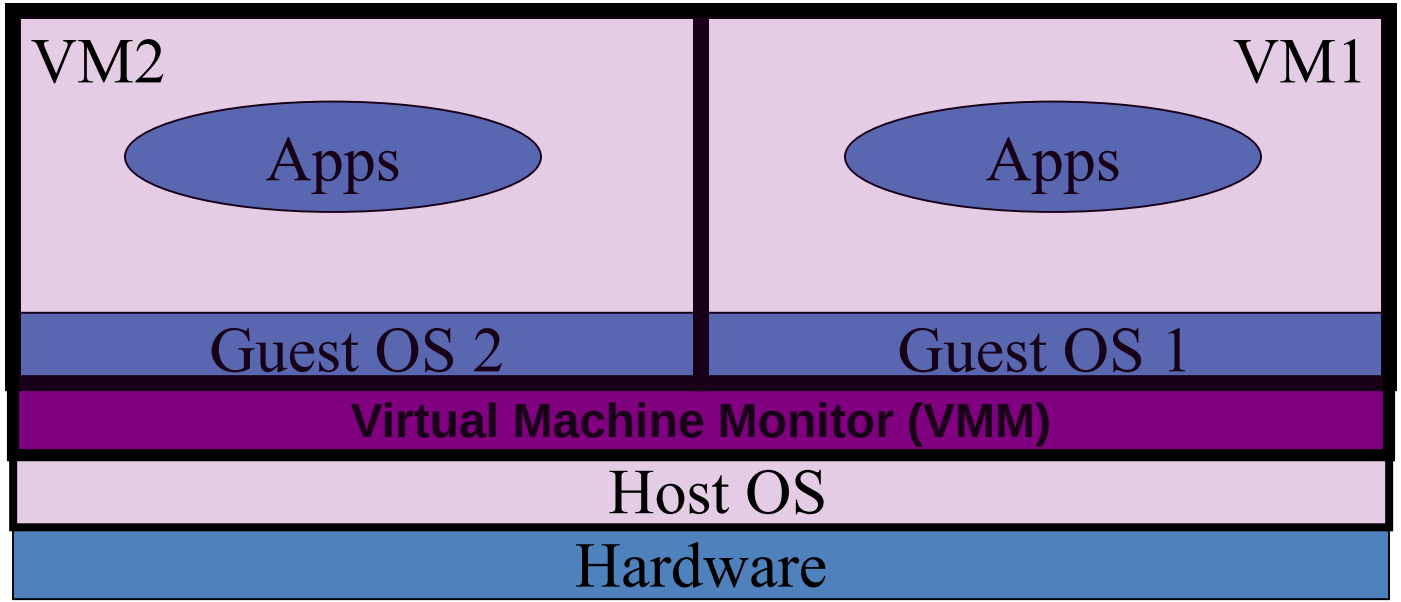
Module 2.6

Isolation via Virtual Machines

Isolation

Isolation via
Virtual Machines

Virtual Machines



Example: **NSA NetTop**

single HW platform used for both classified and unclassified data

Why so popular now?

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 , …
- Wasteful to run each service on different hardware
- More generally: VMs heavily used in cloud computing

VMM security assumption

VMM Security assumption:

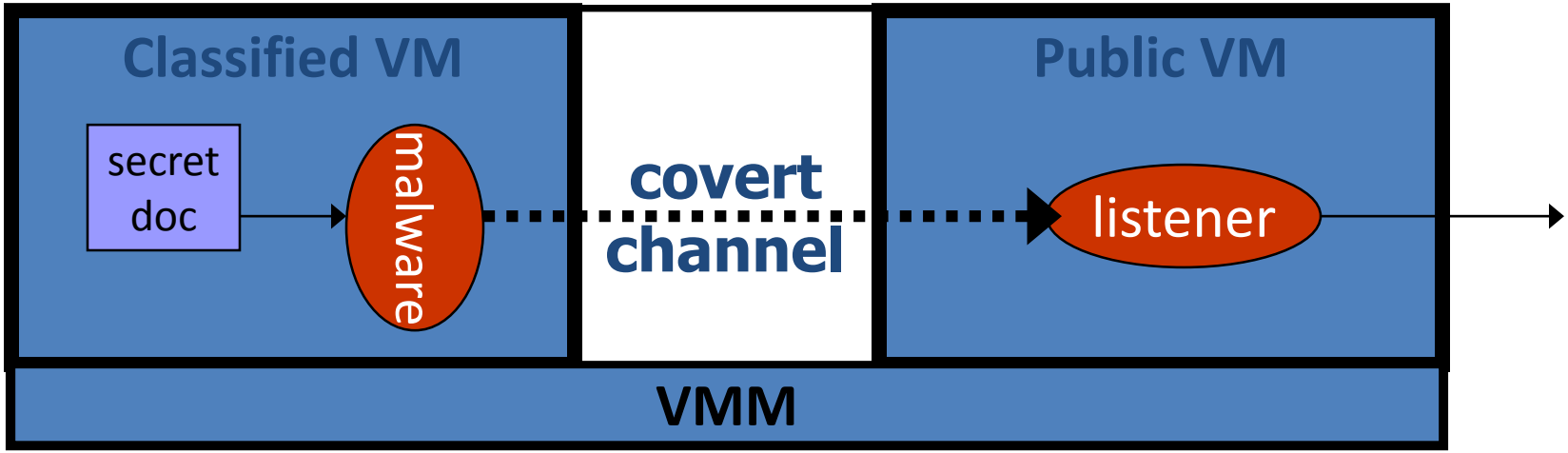
- Malware can infect guest OS and guest apps
- But malware cannot escape from the infected VM
  - Cannot infect host OS
  - Cannot infect other VMs on the same hardware

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



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

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

Many covert channels exist in running system:

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

Suppose the system in question has two CPUs: the classified VM runs on one and the public VM runs on the other.

Is there a covert channel between the VMs?

There are covert channels, for example, based on the time needed to read from main memory

Lecture 2: Summary

- Module 2.1: Confidentiality Policies
- Module 2.2: Confinement Principle
- Module 2.3: Detour: Unix User IDs, Process IDs and privileges
- Module 2.4: More on Confinement Techniques
- Module 2.5: System Call Interposition
- Module 2.6: Isolation via Virtual Machines