

CS 628A: Computer Systems Security

MODULE 2: SECURITY OF PROGRAM EXECUTION AND SYSTEM SECURITY

Isolation and Confinement

Pramod Subramanyan

Acknowledgements

- Sandeep Shukla (IIT Kanpur)
- 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

Three Types of Security Policies

- Confidentiality
- Integrity
- Availability

Information flow policies

Sometimes called the CIA triad

[Anderson, 1972] [Saltzer and Schroeder, 1975]


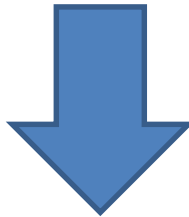
Goals of Confidentiality Policies

Prevent unauthorized disclosure of info

- 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.
- Other examples from US: FERPA, HIPPA

How does access control work?

```
int *ptr = 0x12345678;  
*ptr = 12345678;
```



```
mov rbx, 0x12345678    // get addr  
mov [rbx], 0xcafef00d // store
```

Processor starts executing page fault handler now

How does access control work?

- TLB lookup fails
 - Processor walks page table
- Page walk also fails to return valid translation
 - Processor starts executing page fault handler (OS)
- OS again walks the page table (why?)
 - Finds page is not mapped to application
 - Two things can happen: map page, or signal app
 - Signal can be handled or application killed

Discretionary Access Control (DAC)

- Mechanism where **user** can set controls to allow/deny access to an object
- Also called Identity-based access control (IBAC)
- Traditional access control technique implemented by traditional operating systems such as Unix.
 - Based on user identity and ownership
 - Programs run by user inherit all privileges granted to user
 - Programs is free to change access to user's objects
 - Support only two major categories of users: completely trusted admins and completely untrusted ordinary users

Problems with DAC (1/3)

- Each users has complete discretion over his objects.
 - What is wrong with that?
 - Difficult to enforce a system-wide security policy
 - A user can leak classified documents to a unclassified users.

Problems with DAC (2/3)

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

Problems with DAC (3/3)

- Only support coarse-grained privileges
- Unbounded privilege escalation
- Too simple classification of users (How about more than two categories of users?)

Mandatory Access Control (MAC)

- Mechanism where system control access to an object and a user cannot alter that access.
- 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

Example: SELinux

What Can MAC Offer?

- Supports wide variety of categories of users
- Strong separation of security domains
- System, application, and data integrity
- Ability to limit program privileges
 - Confine damage due to flawed/malicious software
- Authorization limits for legitimate users

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
- honeypots

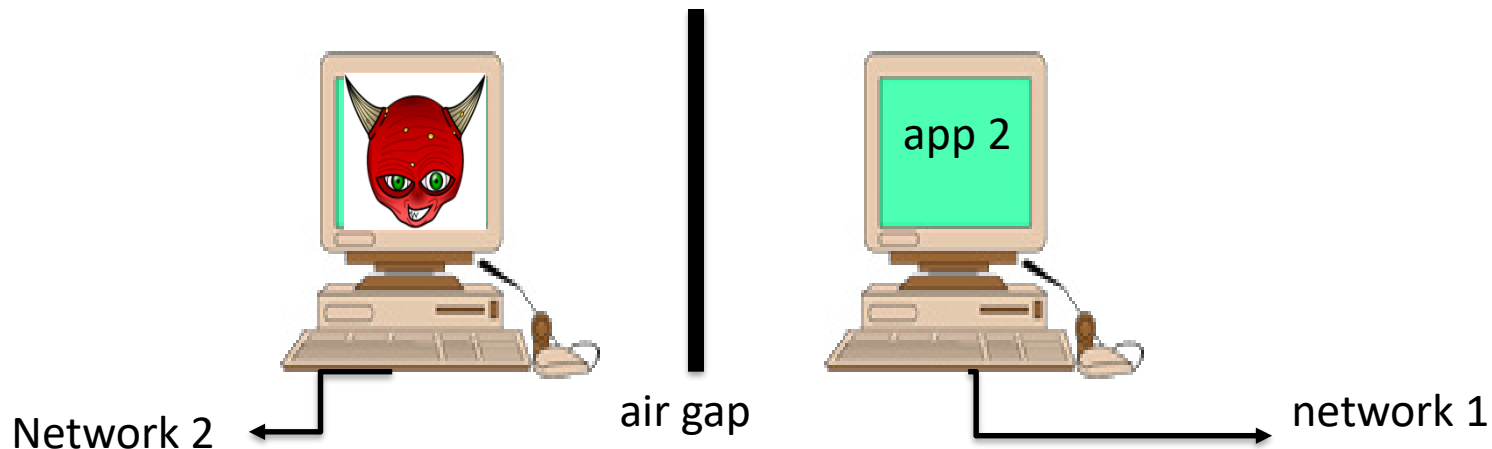
Goal: if application “misbehaves” \Rightarrow **kill it**

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)



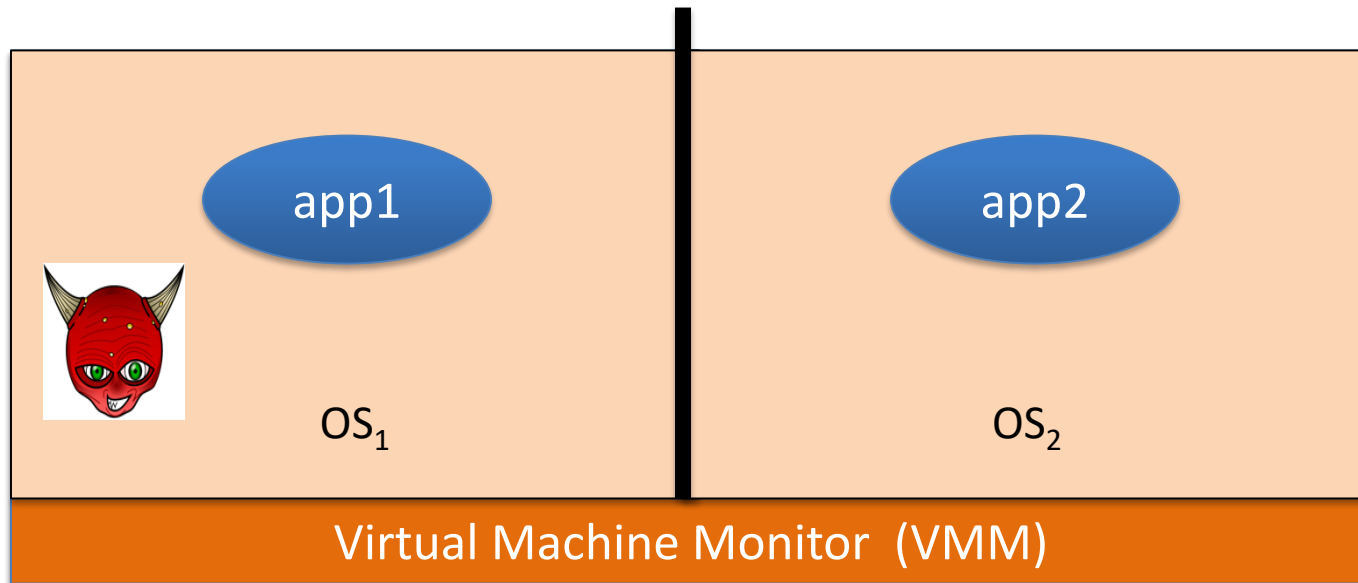
Resource Expensive

Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

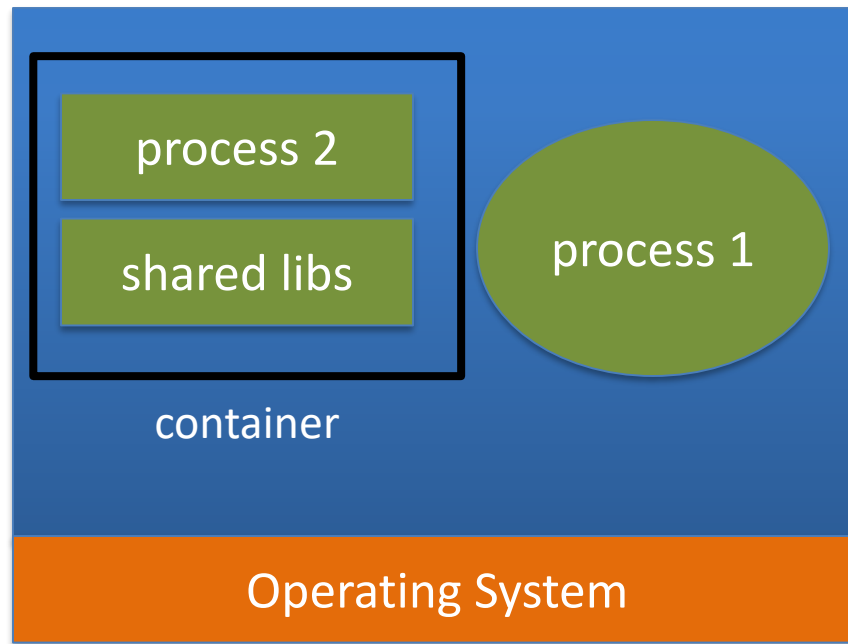
- **Virtual machines**: isolate OS's on a single machine



Approach: confinement

Confinement: Create a **lightweight container** for each application

- **Container:** Has process + all libraries needed for it
- Users inside container may have more rights than outside

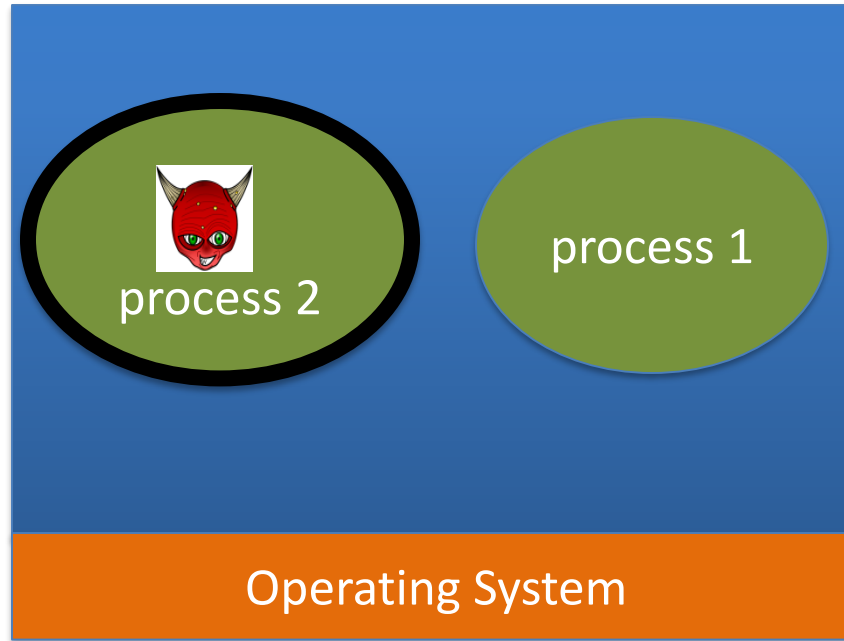


Approach: confinement

Confinement: Ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Process:** System Call Interposition
Isolate a process in a single operating system



Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

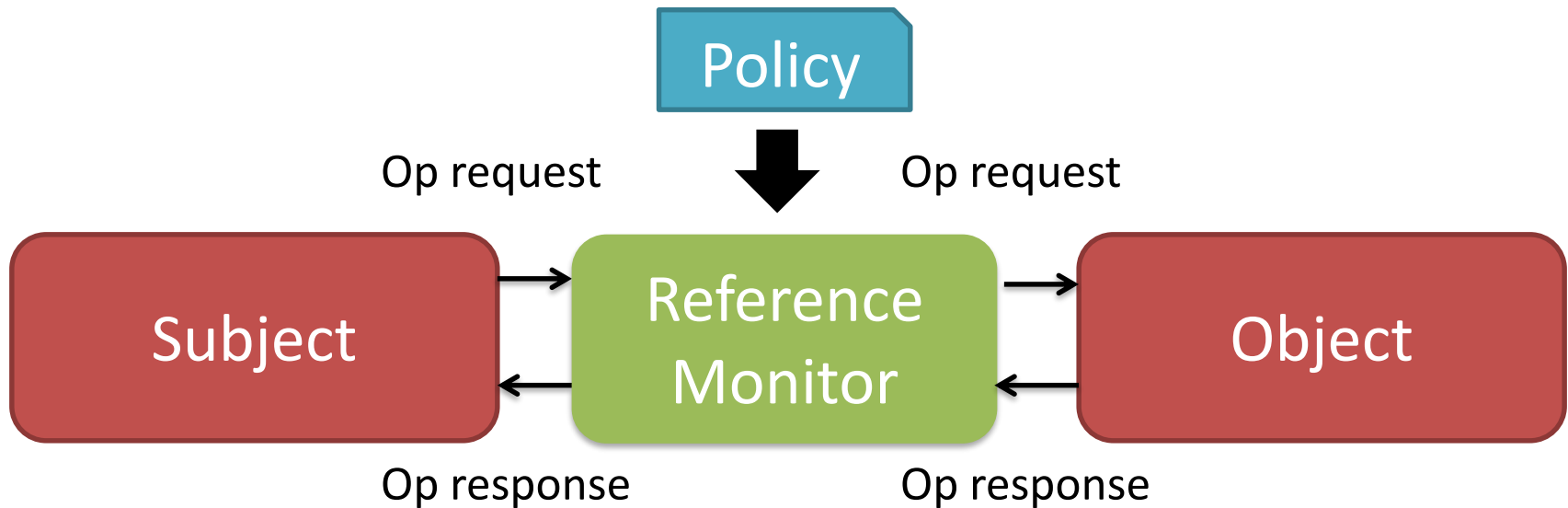
Can be implemented at many levels:

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



Principles:

1. **Complete Mediation:** monitor must always be invoked
2. **Tamper-proof:** monitor cannot be changed by adversary
3. **Verifiable:** reference monitor is small enough to thoroughly understand, test, and ultimately, verify.

Reference Monitors in Practice

- Who is mediating what?
 - For isolation through VMs
 - For isolation through containers
 - For isolation through system call interposition
- How do we ensure complete mediation?
- How do we ensure tamper proofing?

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?

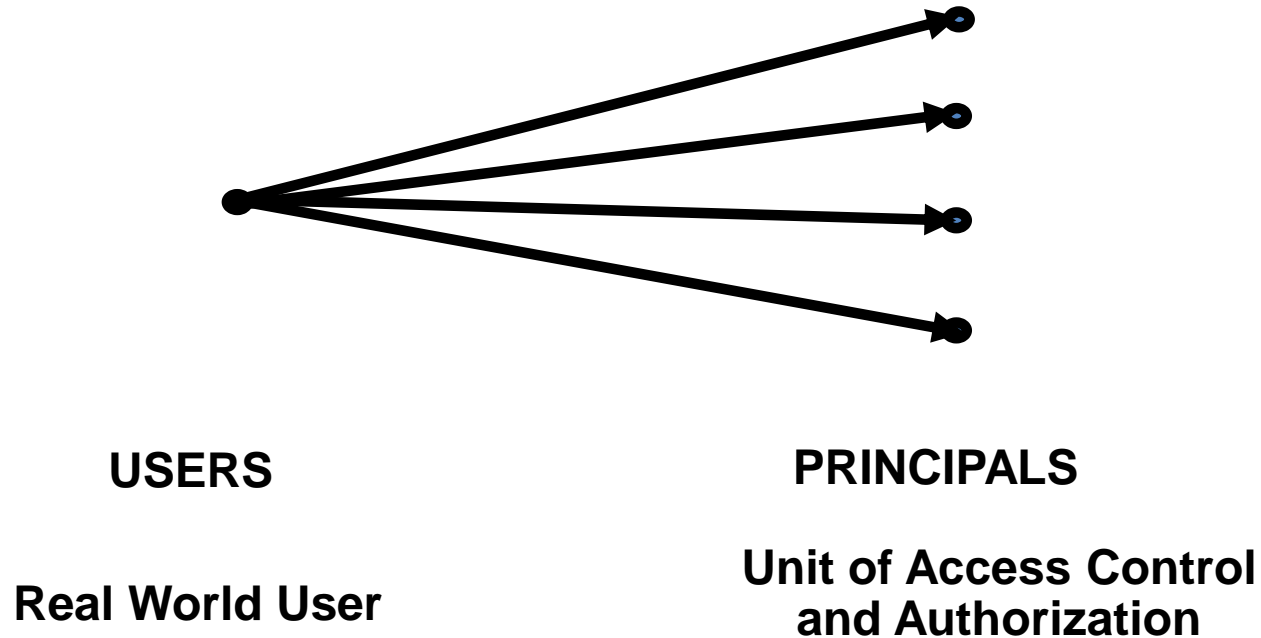
Objects

- 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



the system authenticates the human user to a particular principal

Users and Principals

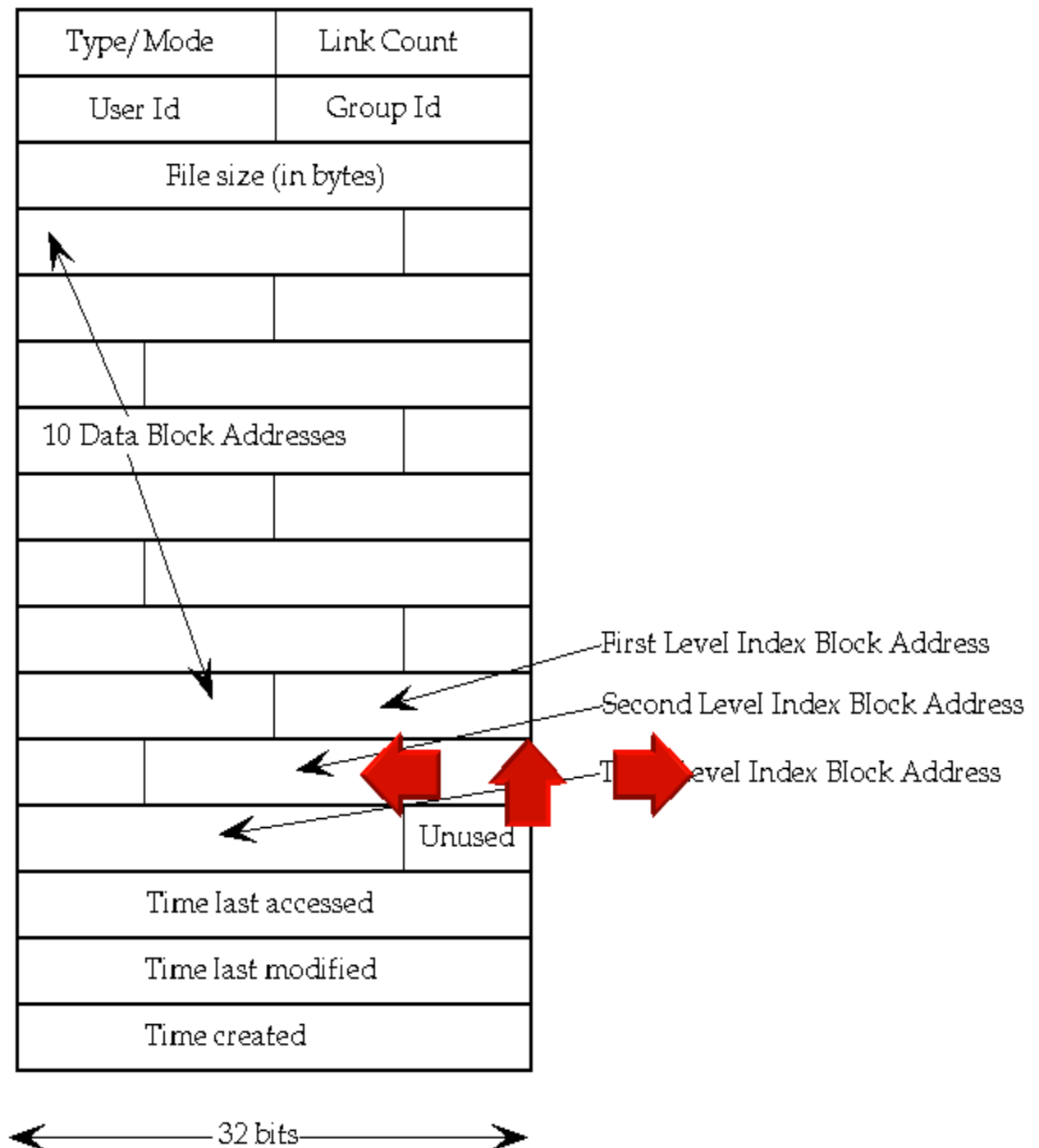
- There should be a one-to-many mapping from users to principals
 - a user may have many principals, but
 - each principal is associated with an unique user
- This ensures accountability of a user's actions

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

- 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 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 (why?)
- 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 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 UIDs 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 user ids via system calls

The Need for suid/sgid Bits

- Some operations are not files and require uid=0
 - halting the system
 - bind/listen on privileged ports (TCP/UDP ports <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

What's the deal with passwords?

- Does the kernel store passwords? **NO!**
 - Where are they stored?
 - /etc/passwd and /etc/shadow
 - Who checks them?
 - An app with setuid bit set checks them
 - It's just a hash a with salt
- ```
$ mkpasswd -msha-512 <passwd> <salt>
```

# What does sudo do?

- Runs a command as root or a specified user
- How does it work?
  - Setuid bit is set on `/usr/bin/sudo`
  - Runs `fork()` and then `execve()`
- Why does sudo ask for the password?
- Will it work without setuid bit?
- Why is it better than logging in as root?
- Impact of buffer overflow in sudo?

# Demo Question 1

- What is the output of
  - \$ echo \$USER
  - # echo \$USER (running as root)
  - \$ sudo echo \$USER
  - \$ bash -c 'echo \$USER'
  - \$ sudo bash -c 'echo \$USER'

# Security Problems of suid/sgid

- These programs are typically setuid root
- Violates the least privilege principle
  - every program and every user should operate using least privilege necessary to complete the job
- Why is violating least privilege 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 `eid` is 0, `setuid` set both the `ruid` and the `eid` to the parameter
    - otherwise, the `setuid` could only set effective uid to real uid
      - Permanently drops privileges
  - A process cannot temporarily drop privilege
- 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

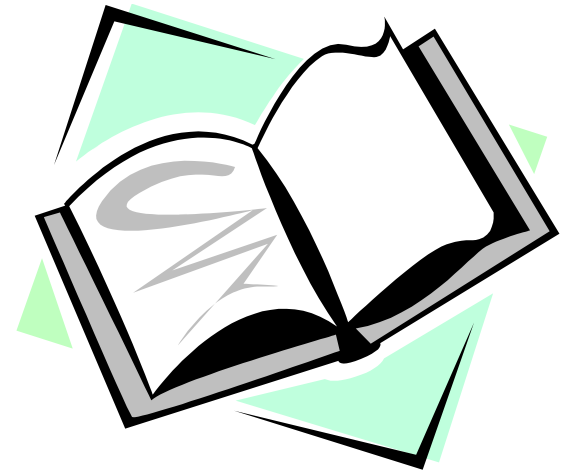
- Uses ruid & euid, change the system call from setuid to setreuid
  - if euid is 0, then the ruid and euid could be set to any user ID
  - otherwise, either the ruid or the euid could be set to value of the other one
    - enables a process to swap ruid & euid

# Modern UNIX

- System V & BSD affect each other, both implemented `setuid`, `seteuid`, `setreuid`, with different semantics
  - some modern UNIX introduced `setresuid`
- Messy, complicated, inconsistent, and buggy
  - POSIX standard, Solaris, FreeBSD, Linux
  - Linux introduced an `fsuid` as well
  - Bug in Linux led to `fsuid = 0` even after dropping other privileges

# Readings for This Lecture

- Wiki
  - [Filesystem Permissions](#)
- Other readings
  - Chapter 4 of Advanced Programming in the Unix Environment
  - Unix file permissions
    - <http://www.unix.com/tips-tutorials/19060-unix-file-permissions.html>



# **PRIVILEGE SEPARATION**

# Principle of Least Privilege (PoLP)

- Restrict privileges to only those required for the operation being performed
- Enforcing PoLP requires **privilege separation**
- Example in logmsg:
  - We only need to have `eid` privileges for `open`
  - Can (and must) give them up after opening file
- Syscall: `setreuid(ruid, eid)`
  - `ruid` and `eid` can be set to either real/eff uid

# 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
- **note**: simple chroot jail does not limit network access

```
$ wget http://olivier.sessink.nl/jailkit/jailkit-2.20.tar.gz
```

```
$ tar zxvf jailkit-2.20.tar
```

```
$ cd jailkit-2.20
```

```
$ sudo ./debian/rules/binary
```

```
$ dpkg -i jailkit_2.20-1_amd64.deb
```

<https://olivier.sessink.nl/jailkit/>

# Escaping from jails

Early escapes: relative paths

`open( "../etc/passwd", "r" ) ⇒`

`open( "/tmp/guest/../../etc/passwd", "r" )`

**chroot** should only be executable by root.

---

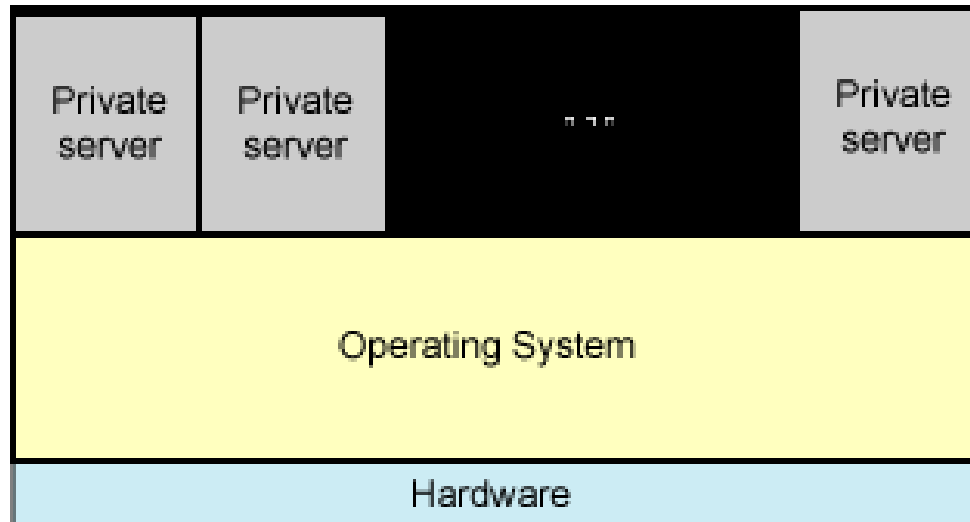
– otherwise jailed app can do:

- 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)



# FreeBSD Jail

- Where chroot jail was weak partitioning, FreeBSD jail is strong partitioning
  - Create virtual machines.
- Popularly used in ISPs.



# 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. for sending attachments in Gmail)

## Does not prevent malicious apps from:

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

# What have we learned so far? (1/2)

- Processor + OS can implement access control
  - Using TLBs + page tables
  - This is the main hardware primitive we need
- Used to implement isolation/confinement
  - Useful primitive for running untrusted code
  - Browser JS, hosted apps on cloud, honeypots, etc.

# What have we learned so far? (2/2)

- Traditional way of implementing isolation
  - Process isolation provided by Unix/Linux
- Relies on four links in the chain
  - Many operations restricted to superuser/root
  - Users are identified by UID (UID=0 means root)
  - Kernel tracks current process UID and enforces permissions for various objects (files, sockets, etc.)
  - Change of uid can happen in only a few ways

# Ways to Change UID

- `setreuid/setresuid` family of syscalls
  - Routinely used by login etc. to reduce privileges
  - Can also restore privilege after reducing it (not a good idea: turns into a bug magnet)
- `setuid` bit set on executable
  - Kernel escalates effective uid upon `execve`

# A lot of this is ancient technology

- Unix its permissions, uids, etc. date to the 70s
- It has obvious problems
  - Coarse grained user differentiation  
(only two levels: root/not root)
  - Unbounded privilege escalation  
(once you are root, you have won)
  - Coarse grained permissions  
(can either read or not read, write or not write)
  - Coarse grained system control  
(can't precisely control what syscalls are allowed)

# In the rest of this module

- Newer (better) ways of implementing isolation
- Based on
  - Containers in modern Linux
  - System call interposition
  - Software Fault Isolation
  - Virtual Machines

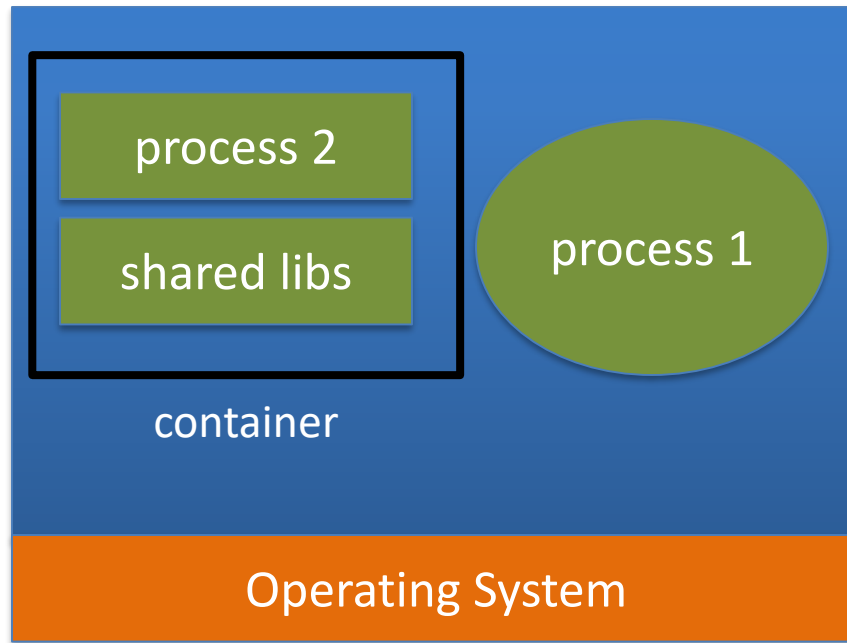


# CONTAINERS

# Approach: confinement

**Confinement:** Create a **lightweight container** for each application

- **Container:** Has process + all libraries needed for it
- Users inside container may have more rights than outside



# Containers in Modern Linux

- Implemented via two important features
  - cgroups: control how much CPU or network bandwidth a set of processes can use
  - namespaces: create a virtual view of system
    - Isolate view of mounted drives
    - Isolate hostname, network, filesystem
    - Isolate users, processes

# Three system calls

- clone
  - Create a new process in a new namespace
- unshare
  - Take an old process into a new namespace
- setns
  - Put an old process inside an existing namespace

# Where do the namespaces live?

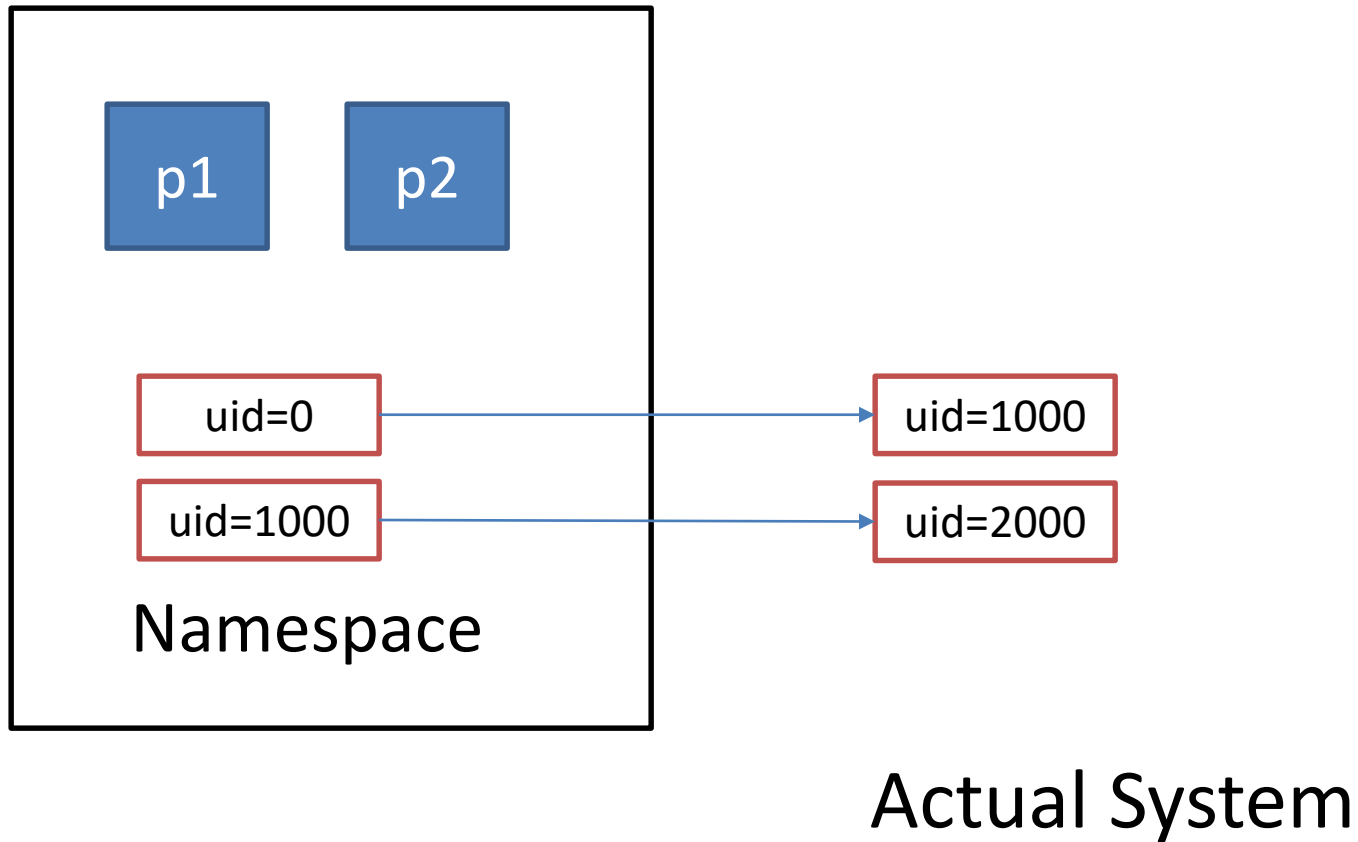
## /proc/PID/ns directory

```
$ ls -l /proc/self/ns
total 0
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 net -> net:[4026531993]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 pid -> pid:[4026531836]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 pid_for_children -> pid:[4026531836]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 user -> user:[4026531837]
lrwxrwxrwx 1 spramod spramod 0 Sep 9 10:19 uts -> uts:[4026531838]
```

# So how to use namespaces

- Use system calls to create new namespaces
- This can isolate:
  - Mounted drives
  - Hostname
  - Network
  - Users and groups
  - Etc.
- Even if program is compromised, it doesn't have access to the entire system

# Isolation via namespaces



# **NAMESPACE API DEMO**



# What do Docker etc. do?

- Provide nice command interface and API
- Internally call Linux namespace/cgroups API
- Provide features for packaging applications
  - This is mainly a business feature
  - Claim is that docker makes it easier to deploy apps

Moral of the story: companies don't want to pay for security but will pay for convenience



# 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, seccomp)

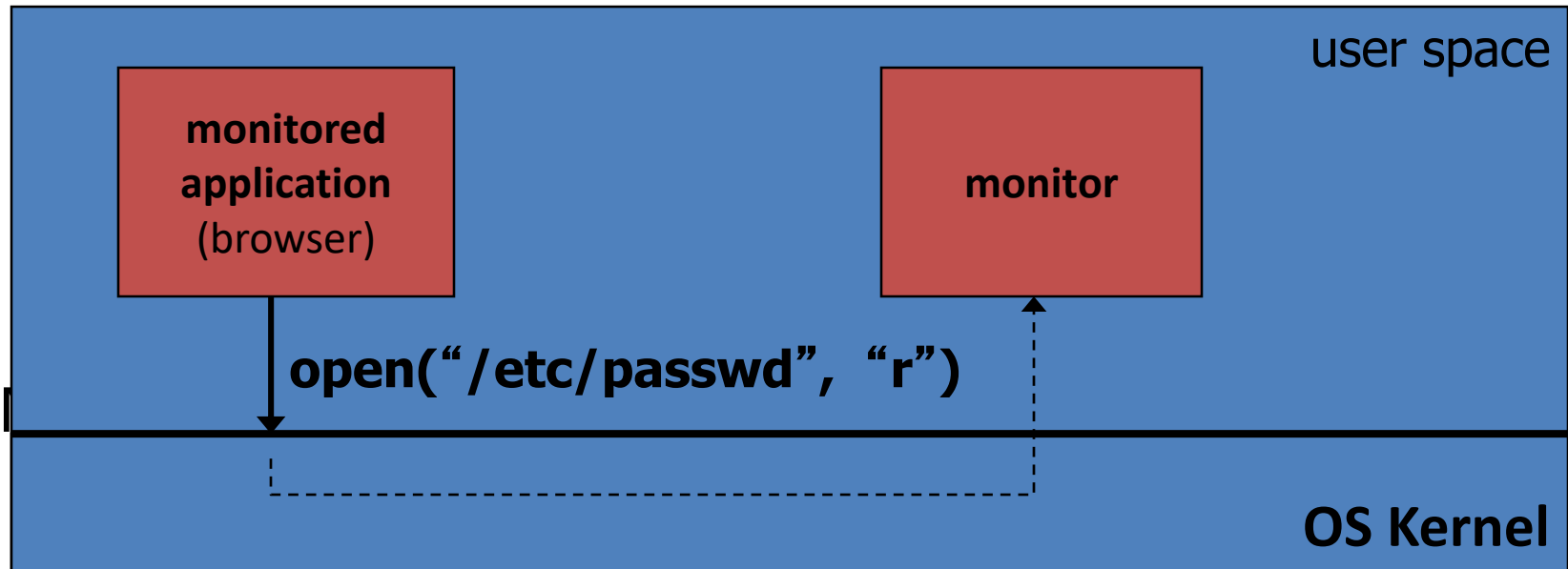
# Initial implementation (Janus)

[GWTB'96]

Linux **ptrace**: process tracing

process calls: **ptrace (... , pid\_t pid , ...)**

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



# 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")
```

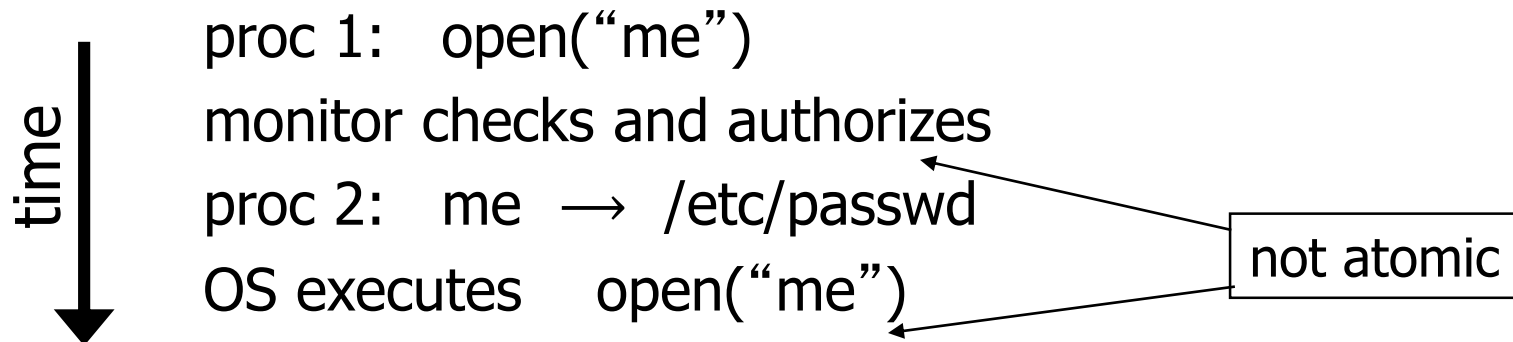
# Problems with ptrace

**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

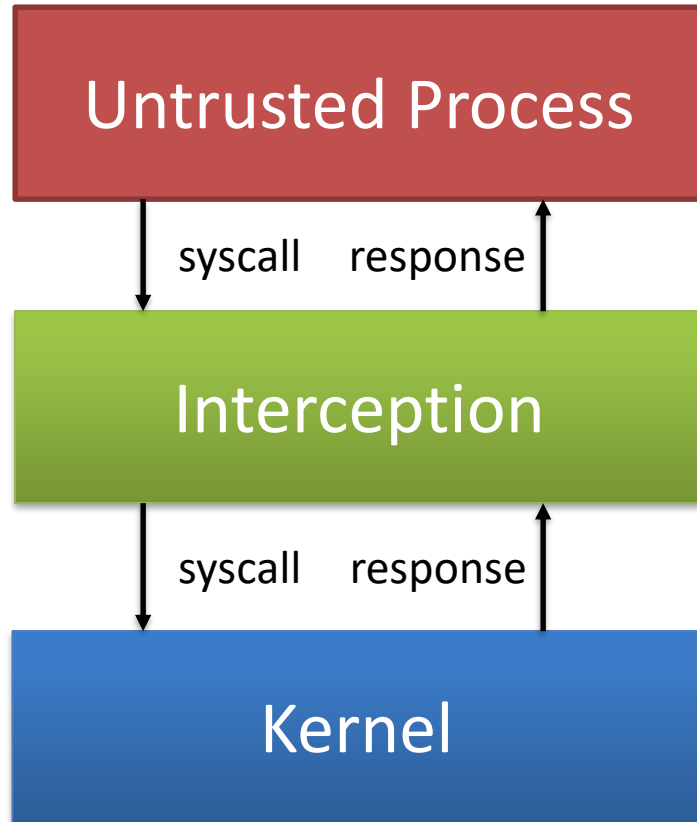
Security problems: **race conditions**

- Example: symlink: me → mydata.dat



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

# Syscall Interposition: Design Choices

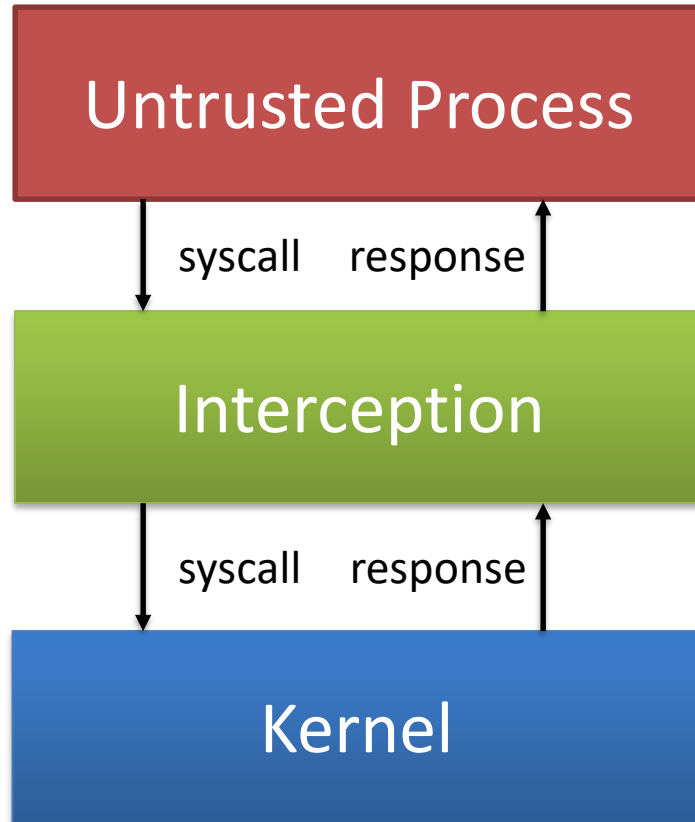


Do we trace all syscalls or only some?

Interception in kernel or userland

Policy decision in kernel or userland?

# Syscall Interposition: Ptrace



Do we trace all syscalls or only some? **ALL**

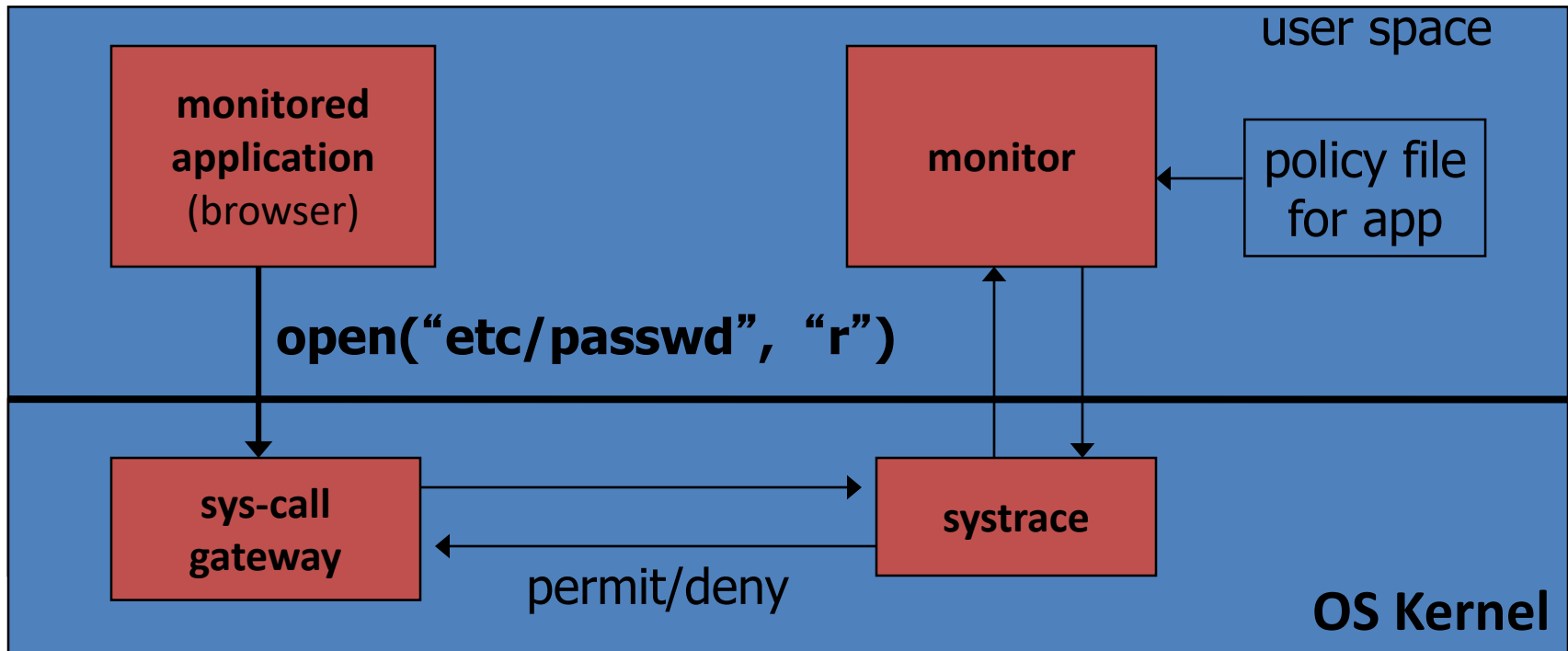
Interception in kernel or userland?  
**Kernel**

Policy decision in kernel or userland?  
**Userland**



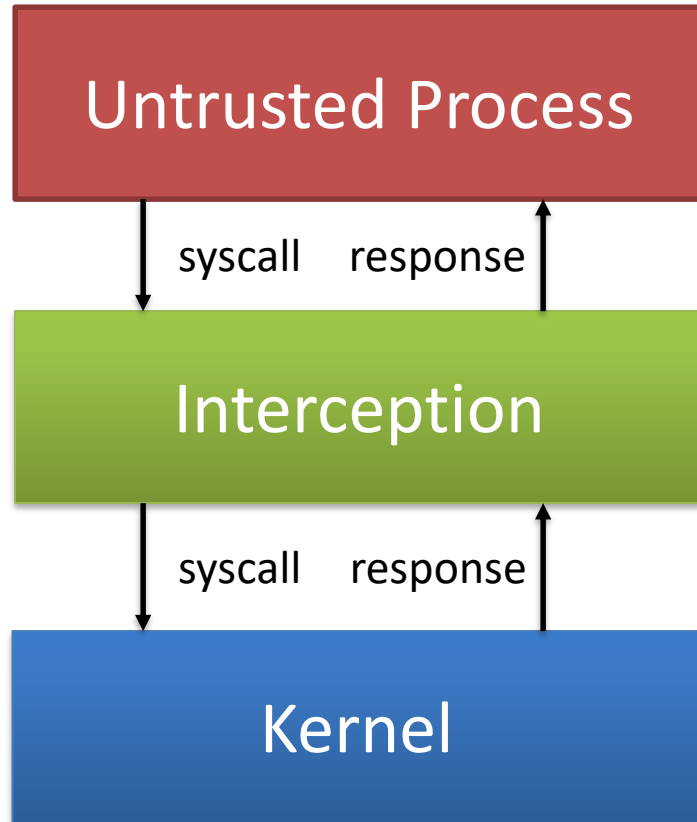
# Alternate design: systrace

[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

# Syscall Interposition: Systrace



Do we trace all syscalls or only some? **Some**

Interception in kernel or userland? **Kernel**

Policy decision in kernel or userland? **Userland**

# Systrace Policies

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
```

# 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

# Ostia: a delegation architecture

[GPR'04]

Previous designs use filtering:

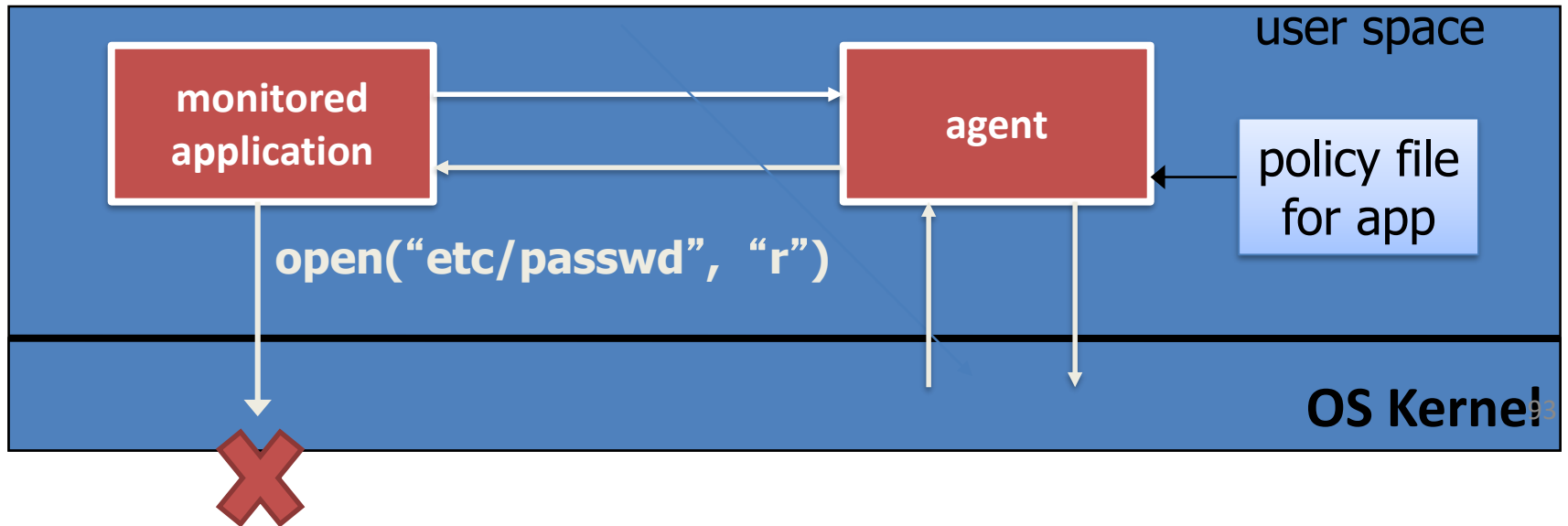
- Filter examines sys-calls and decides whether to block
- Difficulty with syncing state between app and monitor (CWD, UID, ..)
  - Incorrect syncing results in security vulnerabilities (e.g. disallowed file opened)

Instead, a delegation architecture:

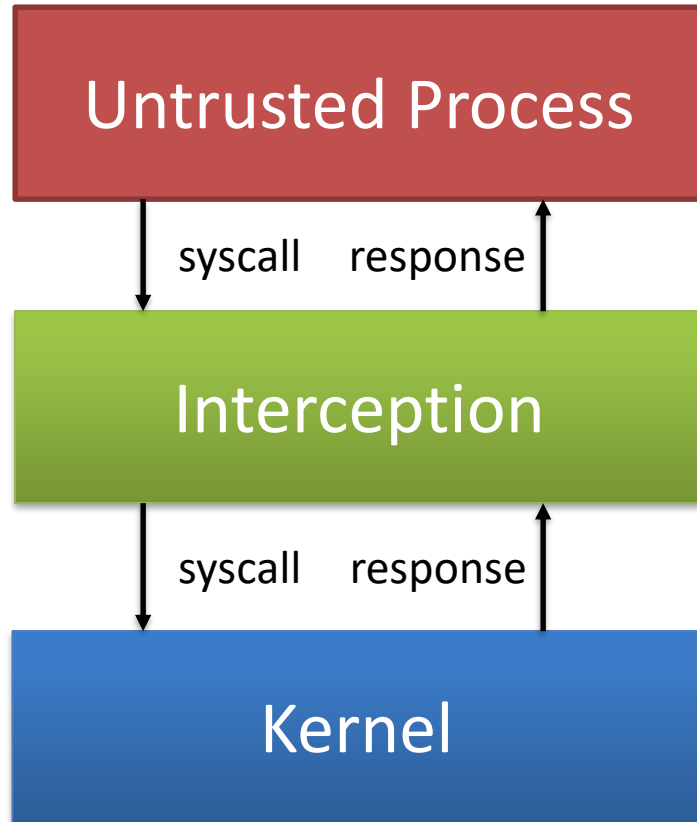
- All syscalls routed to an “agent” which performs them
- Insight: agent can ensure absence of races, symlinks, etc.

# Ostia: a delegation architecture

[GPR'04]



# Syscall Interposition: Ostia



Do we trace all syscalls or only some? **Some**

Interception in kernel or userland?  
**Userland**

Policy decision in kernel or userland?  
**Userland**

# Ostia: a delegation architecture

[GPR'04]

- Monitored app disallowed from making certain syscalls
  - Minimal kernel change ( ... but app can call **close()** itself )
- Sys-call delegated to an agent that decides if call allowed
  - Can be done without changing app
- 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.



# Syscall Interposition: Problems

- Problems with systrace
  - Many subtle race conditions between processing being monitored and monitor process
  - This is despite the great pains systrace et al. go to prevent race conditions
  - e.g., “Exploiting concurrency vulnerabilities in system call wrappers” by R. N. M. Watson
- Problems with Ostia
  - Delegation is inherently slow and wasteful

# Syscall Interposition: Problems (contd)

- Interposition frameworks are C programs
- They themselves had buffer overflow vulns
  - Disastrous consequences
  - Security is worse than no syscall interposition!
- Spurt of papers from 2004 onwards, but sort of died down around 2010 due to the above problems

# But then comes 2014

- And everything old is new again
- Linux adds a new system call “seccomp”
  - To be fair, there was a baby seccomp in Linux since 2005 and the filtering mode was actually added in 2012
- Seccomp allows filtering via BPF programs

# seccomp

- Yet another syscall filtering platform
- Filtering is done using BPF
- What is BPF?
  - A small programming language
  - Programs come from userspace
  - But executed in kernel (what are the dangers?)
  - Verifier ensures programs do only allowed things
  - No loops (why?), no uninitialized reads (why?)

# Seccomp Example Code

- <https://gist.github.com/fntlnz/08ae20befb91beefd9a53cd91cdc6d507>

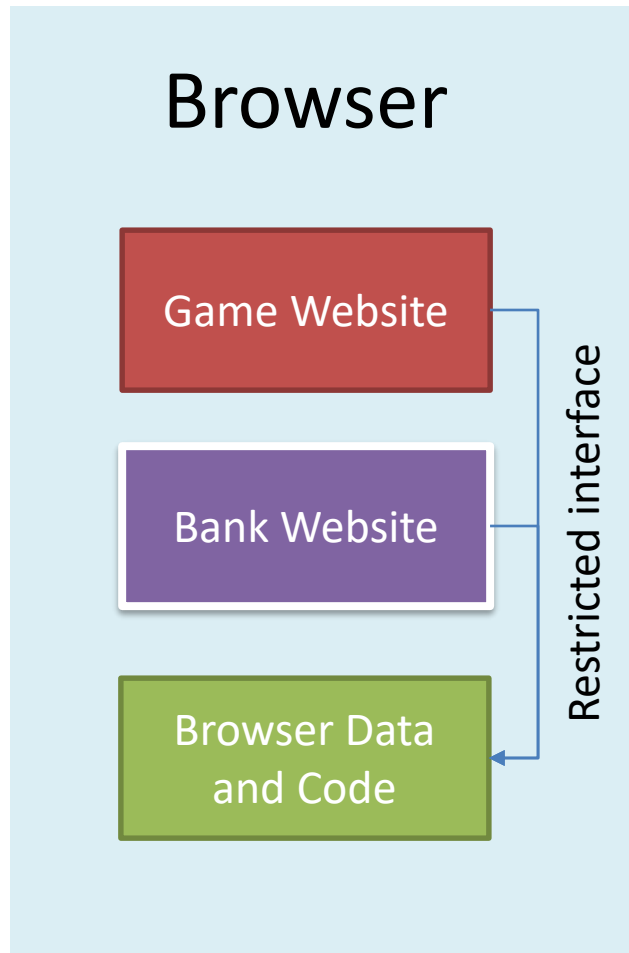


# Software Fault Isolation

---

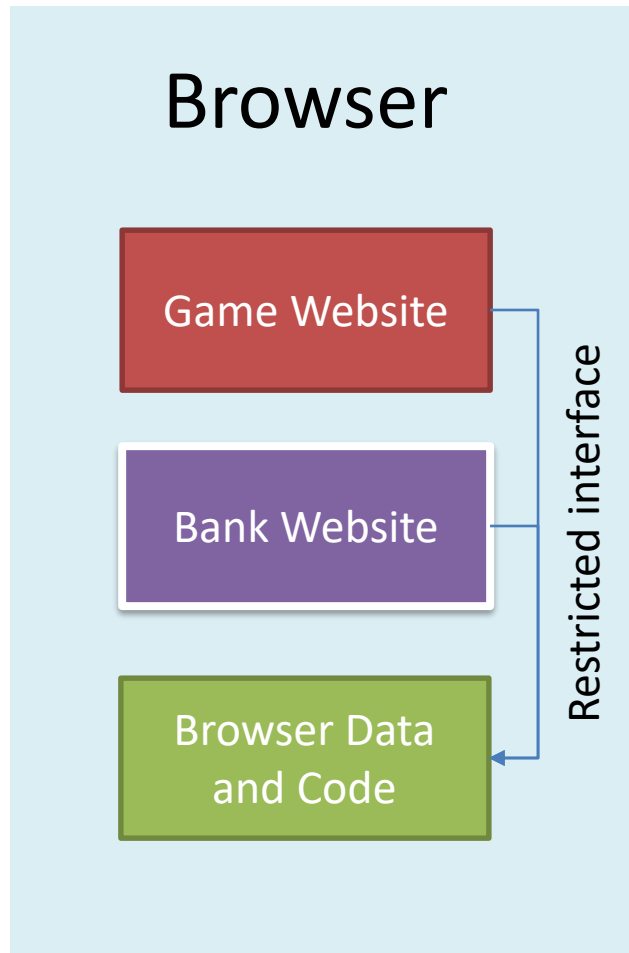
Isolating components  
within a single program

# Component Isolation



- Want game to not be able to read/write bank data
- Shouldn't modify or invoke browser internal functions or make syscalls directly

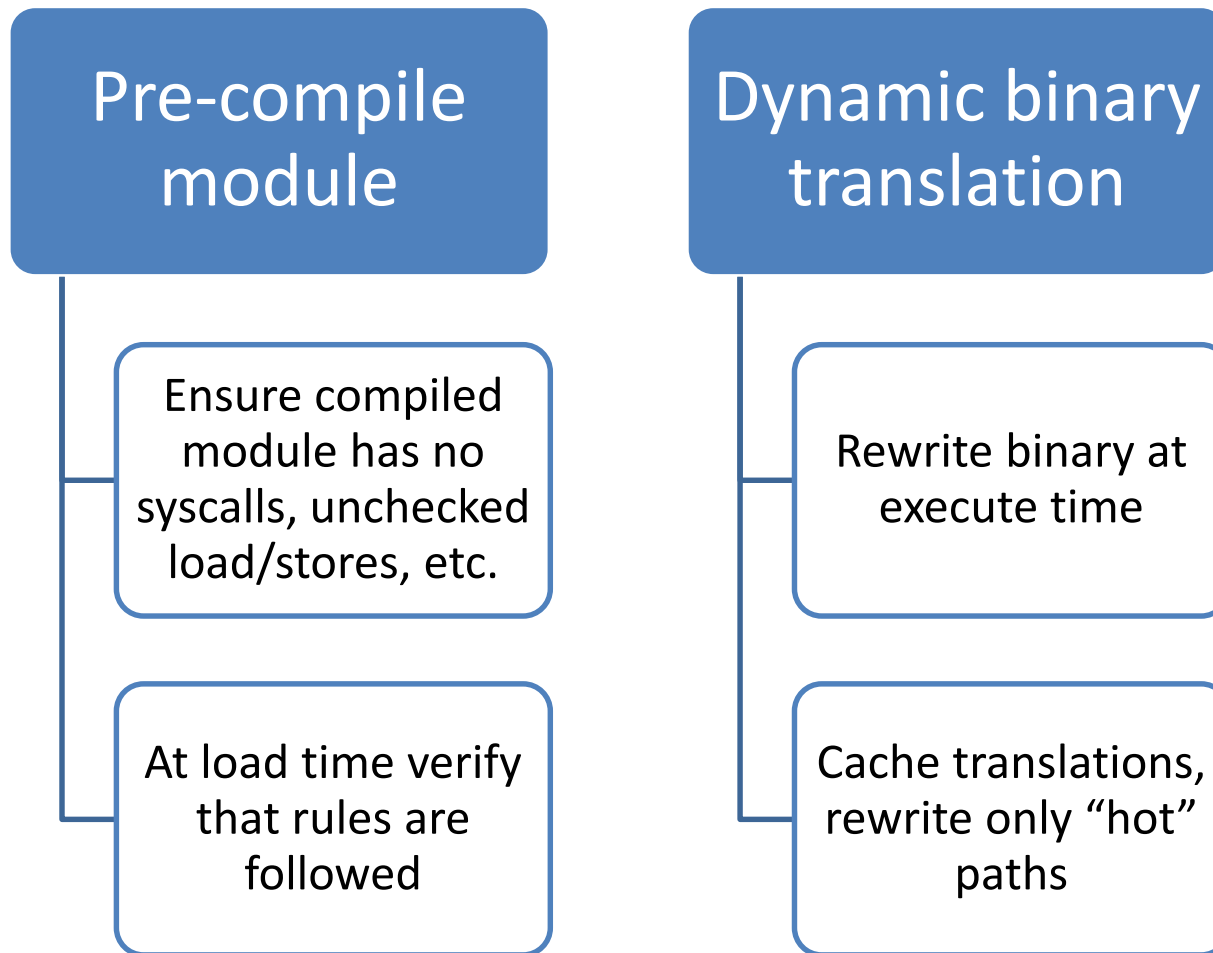
# Component Isolation using SFI



- Allocate contiguous region of virtual addresses to each component
- Ensure that loads/writes only access this region
- Ensure no syscalls, and interaction with browser via restricted API



# Two Approaches to SFI



# At a high-level SFI means

- Just disallow all syscalls in binary
  - Make them through some monitor instead
- Check each load/store
  - Make sure it's address is within a specified range
  - This involves adding a branch before each load
- Seems easy-peasy so what's the catch?

# I got x86 Problems

- `mov edx, 0xC23BCB50 ; ba 50 cb 3b c2`
- But `50 cb 3b c2` is:
  - `push eax`
  - `return`
  - `cmp ...`
- X86 opcodes range in length from 1 to 15 bytes
- There are a lot of prefixes
  - They modify the meaning of the next opcode
  - Examples: REP, REPNE, LOCK etc.

# Things to worry about

- Hard to convert binary into instruction stream
  - Attacker may jump into middle of an instruction
- Attacker could cause jump into middle of loop
  - Redundant checks inside loop?
  - Even redundant checks won't be enough?

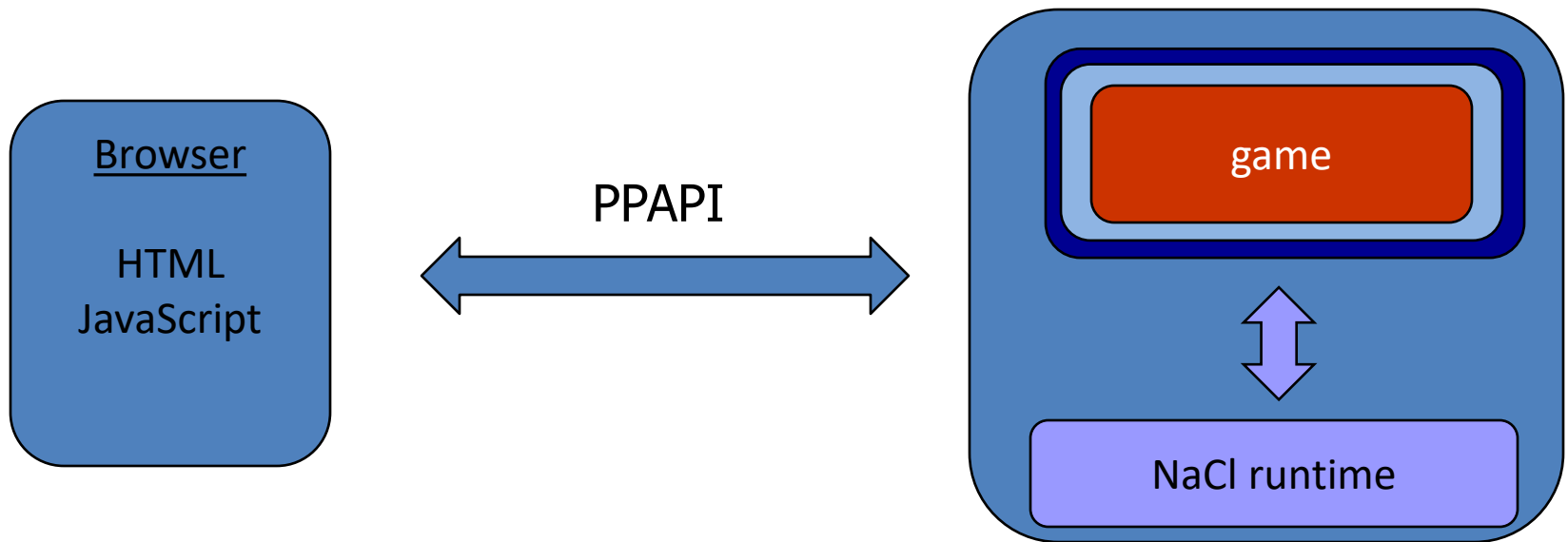
# NativeClient

Native Client: A Sandbox for Portable, Untrusted x86 Native Code, IEEE SP 2009

- Want to run games etc. in the browser
- But JavaScript is (used to be?) very slow
- How about if we run compiled code instead?

Related efforts: WebAssembly, asm.js

# 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

# NaCl's Approach to Sandboxing

## Load time verification

- No direct syscalls, restricted instr alignment, only few prefixes allowed
- Ensures “reliable disassembly”

## Runtime

- Inner sandbox: uses **x86 segmentation**
- Outer sandbox: intercept system calls
- “Defense in depth”

## Runtime

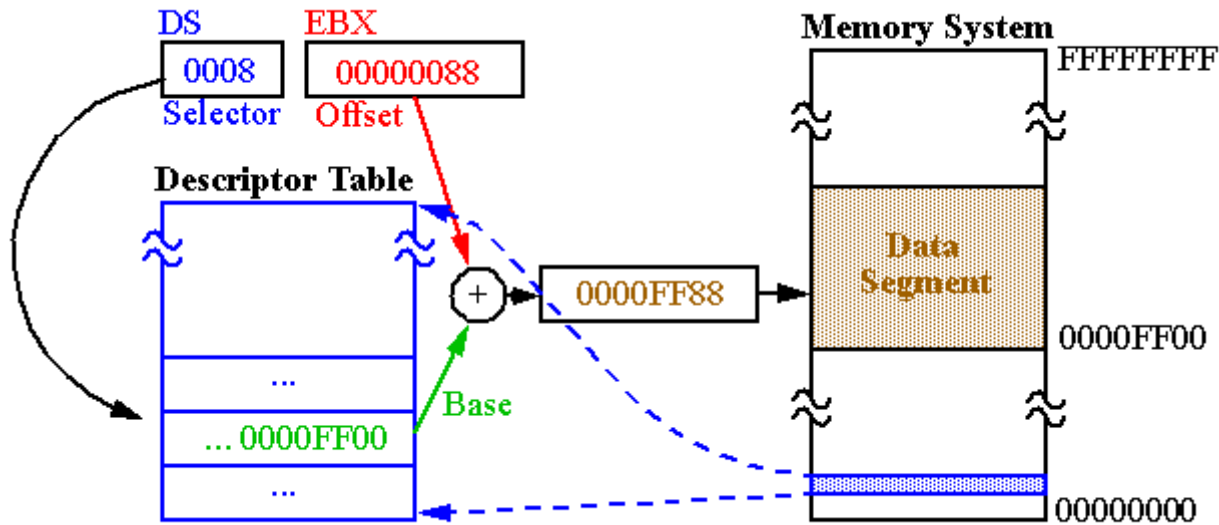
- Provides access to certain system calls, browser state

# Segment Registers on x86

- A relic of the days when paging was too expensive
- X86 has 6 segment registers: CS, DS, SS, ES, FS, GS
- In the old days:
  - `mov [bx], 0xffff`
  - Effective address =  $ds * 16 + bx$
  - Store 0xffff at effective address
- In the new days (i.e., post 1985 or so):
  - A bit more complex, see next slide



# X86 Protected Mode Segmentation



[http://ece-research.unm.edu/jimp/310/slides/micro\\_arch2.html](http://ece-research.unm.edu/jimp/310/slides/micro_arch2.html)

- Segment registers point to a **descriptor table entry**
- Base value in entry combined with “offset” to produce effective address
- Entries also have a limit value

# OK, what is the point of all this?

- Can set segment register values in user mode
- Use segment regs to constrain execution of sandboxed code to certain memory addresses
- Just need to set base and limit appropriately

# NaCl Compilation

- Using modified GCC toolchain
- Only SDK compiler can produce nexx files
- Compiler does not emit blacklisted instructions
- No instructions span a 32-byte boundary
- No ret instructions in the emitted code
- Call/branch manually masked to be 32b aligned
- Secure stack for runtime, insecure for application

# nacljmp Pseudoinstruction

nacljmp eax is transformed to  
and eax, 0xFFFFFFFFE0  
jmp [eax]

- Ensure control flow changes are to 32b aligned instr
- Indirect jumps and returns impl using this

# NaCl Validator

- Disassembles NEXE to make sure no bad instrs in it
- Starts are 32b aligned entry point
- Exits if any blacklisted instructions are found
  - Privileged instructions (mov cr3, etc.)
  - Modifications to segment registers
  - ret, sysenter, prefix bytes
  - Ensures all branches to 32b aligned address

```
and eax, 0xfffffe0
call eax
```

good

```
call eax
```

bad

# NaCl Runtime

- Provides access to mmap etc.
- Call/return code in top 64K of NaCl module
- This code can modify segment registers
- Sets up trusted stack etc.

# Summarizing NaCl

- Very comprehensive effort for x86 sandboxing
- Only ~5% slower than unsandboxed code
- Sort of out of date due to WebAssembly

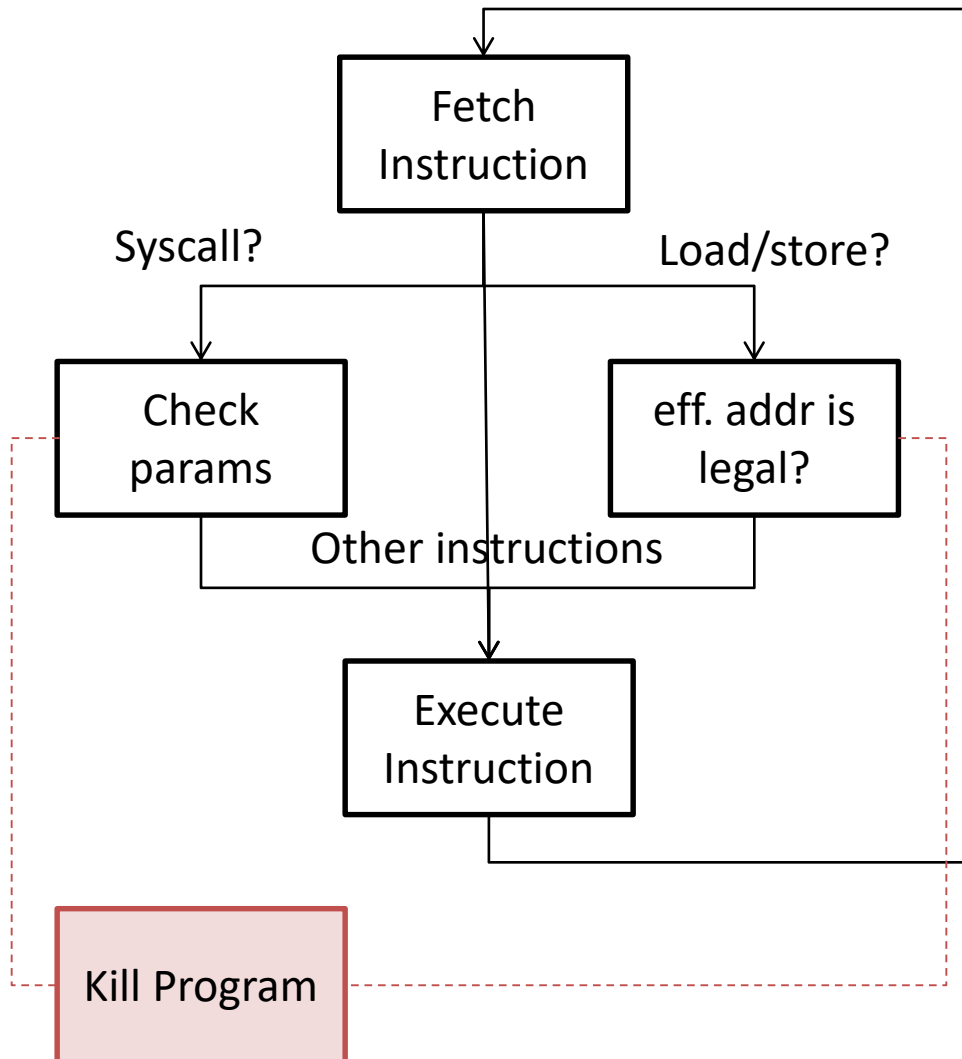
# Bonus: NaCl Bug

```
and edx, 0xfffffffffe0
call [edx]
```

- Discovered in 2009 by Alex Radocea
- Ensures address in edx is 32b aligned,
- But we are jumping to the value that edx points to!

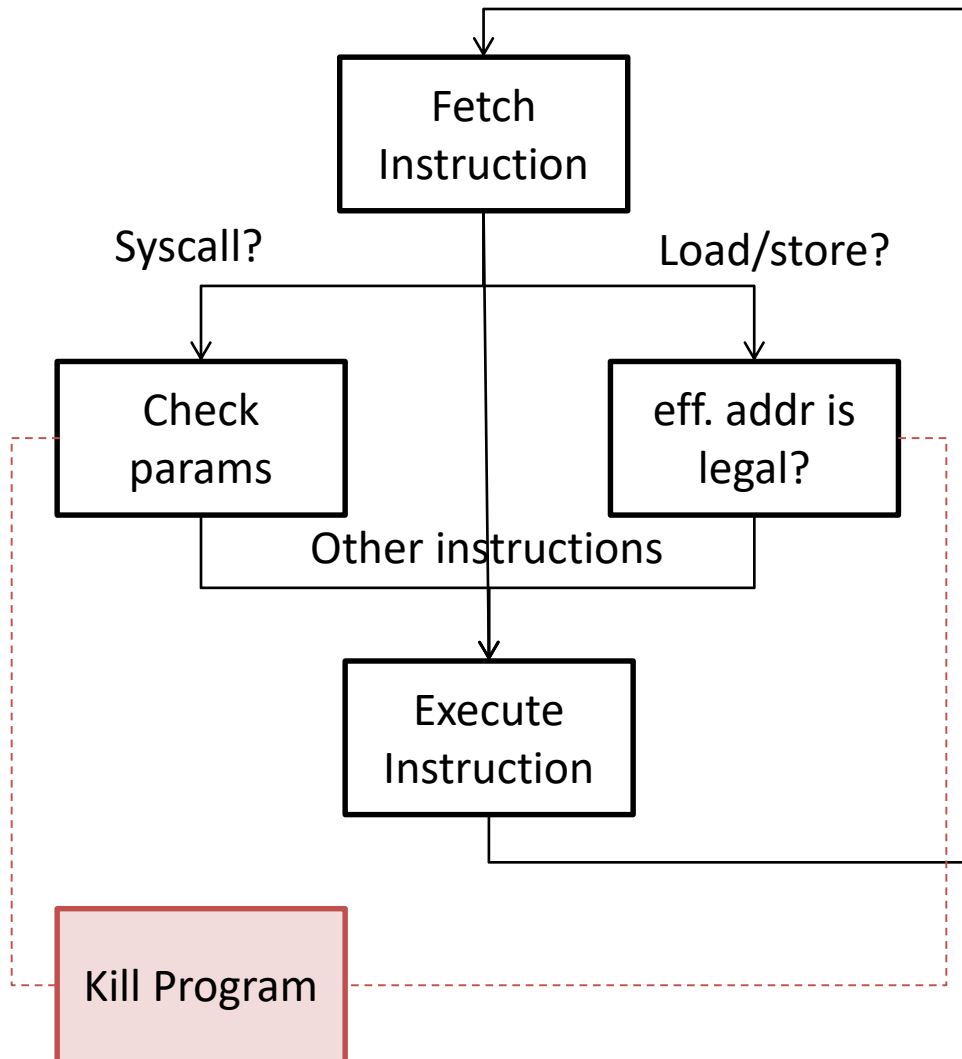


# Program Shepherding: Strawman



- Idea: interpret each instruction
- Check if instruction is doing something disallowed by policy
- If so, kill program
- If not, update state according to instruction semantics

# Strawman: Pros and Cons



## Pros

- No disassembly problems
- Can strictly enforce policy

## Cons

- Extremely slow
- Execute 10s/100s of machine instr for each target instr

# Dynamic Binary Translation to the Rescue!

- Fetch basic block
- If it contains no problematic instructions then compile it to machine code and keep in cache
- If BB is visited again, just execute from cache
- If it contains problematic instrs, introduce checks in the compiled code and then insert into cache

# Program Shepherd

| Restricting                | Least restrictive |                     | Most restrictive                                                    |                                                               |                                             |
|----------------------------|-------------------|---------------------|---------------------------------------------------------------------|---------------------------------------------------------------|---------------------------------------------|
| Code origins               | Any               |                     | Dynamically written 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 <code>xor</code> 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 target 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                                        |
| <code>execve</code>        | Any               |                     | Static arguments                                                    | Only if the operation can be validated not to cause a problem | None                                        |
| <code>open</code>          | Any               |                     | Disallow writes to specific files (e.g., <code>/etc/passwd</code> ) | Only to a subregion of the file system                        | None                                        |

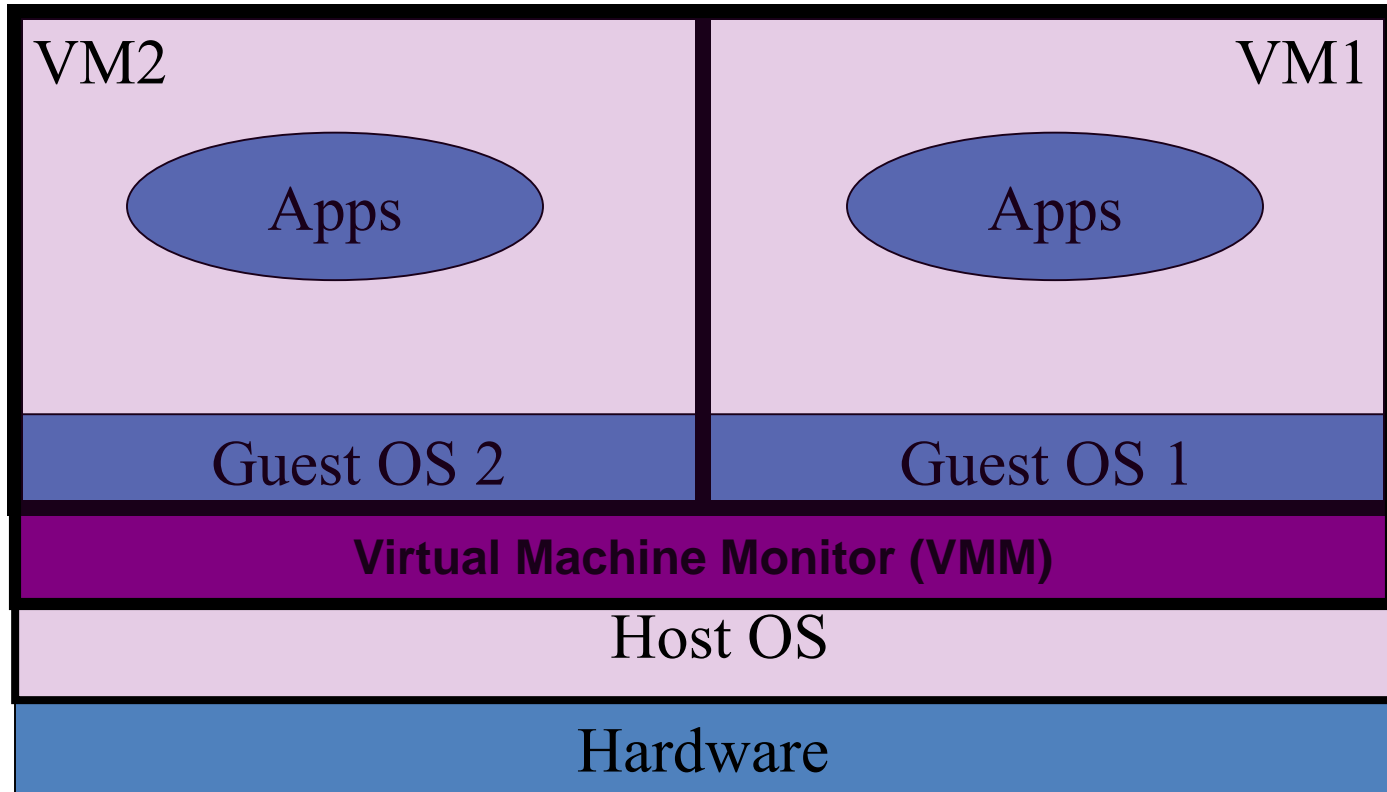


# Isolation

---

## 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
- VMs heavily used in cloud computing

# VMM security assumption

## **VMM Security assumption:**

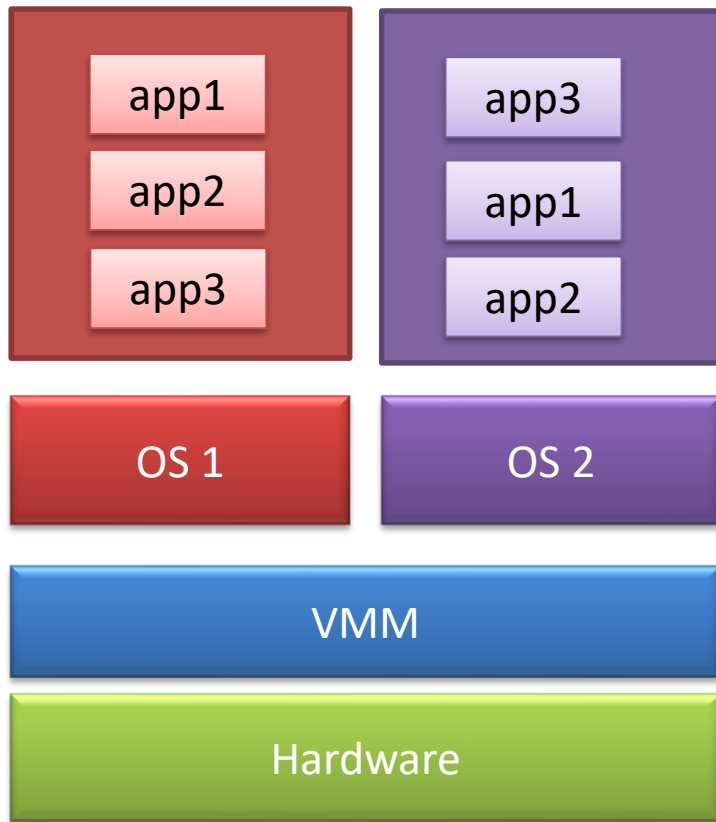
- Malware can infect guest OS and guest apps
- But malware cannot escape from the infected VM
  - Cannot infect host OS
  - Cannot infect other VMs on the same hardware

Requires that VMM protect itself and is not buggy

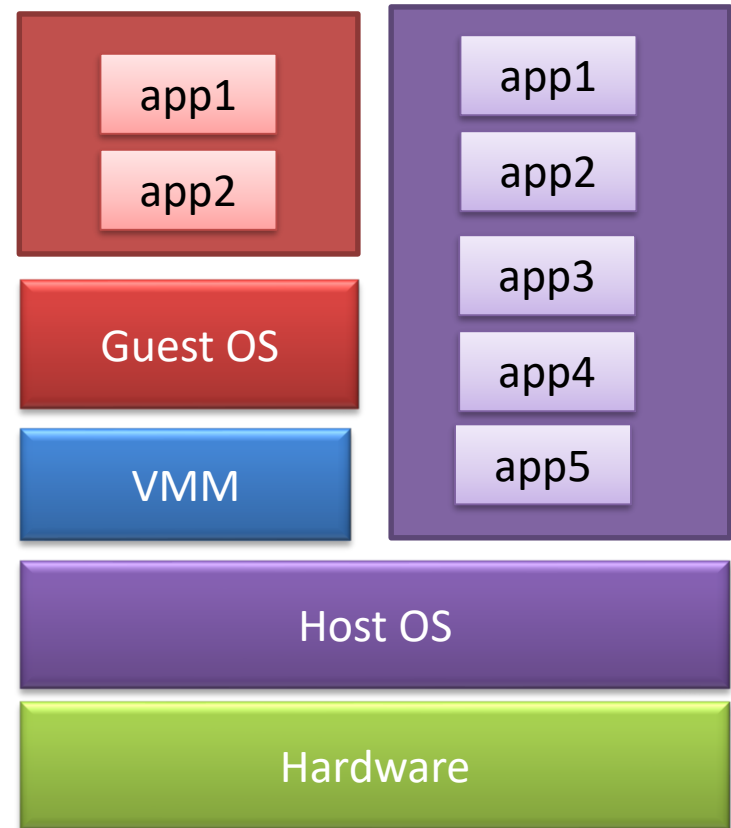
- VMM is much (?) simpler than full OS



# Types of VMs



Type I

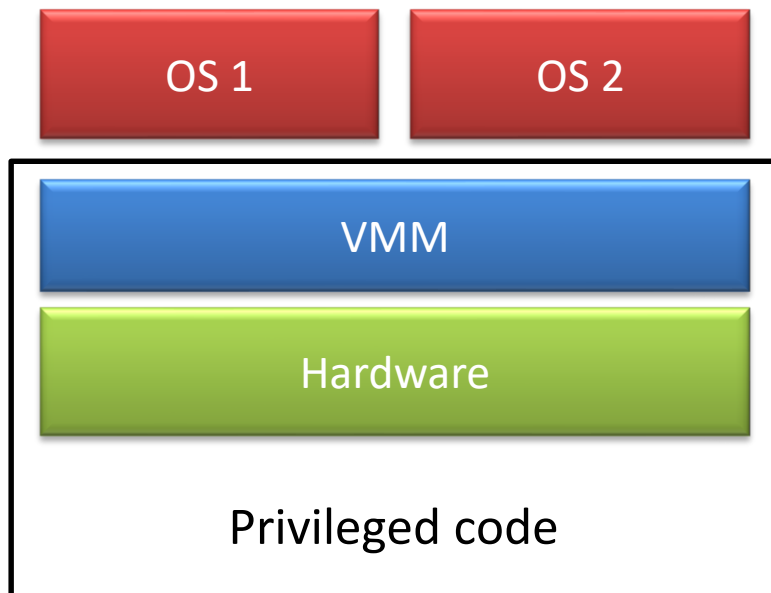


Type II

# Type I VMM Strawman

## Strawman implementation

- VMM becomes the OS
- OS is a regular user-space application



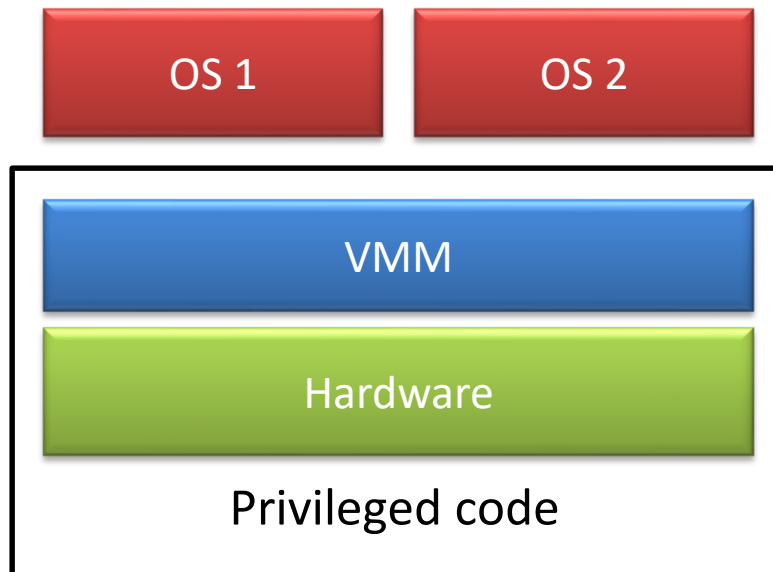
So what is wrong with this picture?

OS needs to execute priv instrs: set PT base, I/O and interrupts etc.

# Type I VMM Strawman -- Improved

Strawman-ish implementation

- VMM becomes the OS
- OS is **interpreted** user-space application
- Priv instrs interpreted appropriately by VMM



Does this work?

Yes, but it is dog slow

# VMM Implementations: Reality

- VMM acts like the OS (runs at higher priv)
- Most guest OS instructions executed directly
- When guest OS exec priv inst, processor traps
  - Traps into VMM which handles it appropriately
  - Returns to OS after instruction is executed
- VMM maintains **shadow page tables**
  - Remember that OS maps VA  $\rightarrow$  PA
  - VMM maps PA  $\rightarrow$  MA (machine addr)
  - Each VA  $\rightarrow$  PA update converted to VM  $\rightarrow$  MA update

# A Detour: Few Words about Rootkits

# Topics

## Rootkits

- User-mode Rootkits
- Kernel Rootkits
- Detecting Rootkits
- Recovery from a Rootkit

# What is a rootkit?

Collection of attacker tools installed after an intruder has gained access

- Log cleaners
- File/process/user hiding tools
- Network sniffers
- Backdoor programs

# Rootkit Goals

1. Remove evidence of original attack and activity that led to rootkit installation.
2. Hide future attacker activity (files, network connections, processes) and prevent it from being logged.
3. Enable future access to system by attacker.
4. Install tools to widen scope of penetration.
5. Secure system so other attackers can't take control of system from original attacker.



# Concealment Techniques

- Remove log and audit file entries.
- Modify system programs to hide attacker files, network connections, and processes.
- Modify logging system to not log attacker activities.
- Modify OS kernel system calls to hide attacker activities.

# Installation Concealment

- Use a subdirectory of a busy system directory like /dev, /etc, /lib, or /usr/lib
- Use dot files, which aren't in ls output.
- Use spaces to make filenames look like expected dot files: “. “ and “.. “
- Use filenames that system might use
  - /dev/hdd (if no 4<sup>th</sup> IDE disk exists)
  - /usr/lib/libX.a (libX11 is real Sun X-Windows)
- Delete rootkit install directory once installation is complete.

# Attack Tools

- Network sniffer
  - Including password grabber utility
- Password cracker
- Vulnerability scanners
- Autorooter
  - Automatically applies exploits to host ranges
- DDOS tools

# History of Rootkits

**1989:** Phrack 25 Black Tie Affair: wtmp wiping.

**1994:** Advisory CA-1994-01 about SunOS rootkits.

**1996:** Linux Rootkits (lrk3 released.)

**1997:** Phrack 51 halflife article: LKM-based rootkits

**1998:** Silvio Cesare's kernel patching via kmem.

**1999:** Greg Hoglund's NT kernel rootkit paper

# History of Rootkits

**2005:** Sony ships CDs with rootkits that hide DRM and spyware that auto-installs when CD played.

**2006:** SubVirt rootkit moves real OS to a VM.

# Rootkit Types

## User-mode Rootkits

- **Binary Rootkits** replace user programs.
  - Trojans: ls, netstat, ps
  - Trojan backdoors: login, sshd.
- **Library Rootkits** replace system libraries.
  - Intercept lib calls to hide activities and add backdoors.

## Kernel Rootkits

- Modify system calls/structures that all user-mode programs rely on to list users, processes, and sockets.
- Add backdoors to kernel itself.

# Binary Rootkits

- Install trojan-horse versions of common system commands, such as ls, netstat, and ps to hide attacker activities..
- Install programs to edit attacker activity from log and accounting files.
- Install trojan-horse variants of common programs like login, passwd, and sshd to allow attacker continued access to system.
- Install network sniffers.

# Linux Root Kit (LRK) v4 Features

|            |                                               |
|------------|-----------------------------------------------|
| chsh       | Trojaned! User->r00t                          |
| crontab    | Trojaned! Hidden Crontab Entries              |
| du         | Trojaned! Hide files                          |
| fix        | File fixer!                                   |
| ifconfig   | Trojaned! Hide sniffing                       |
| inetd      | Trojaned! Remote access                       |
| linsniffer | Packet sniffer!                               |
| login      | Trojaned! Remote access                       |
| ls         | Trojaned! Hide files                          |
| netstat    | Trojaned! Hide connections                    |
| passwd     | Trojaned! User->r00t                          |
| ps         | Trojaned! Hide processes                      |
| rshd       | Trojaned! Remote access                       |
| sniffchk   | Program to check if sniffer is up and running |
| syslogd    | Trojaned! Hide logs                           |
| tcpd       | Trojaned! Hide connections, avoid denies      |
| top        | Trojaned! Hide processes                      |
| wted       | wtmp/utmp editor!                             |
| z2         | Zap2 utmp/wtmp/lastlog eraser!                |



# Linux Root Kit (LRK) v4 Trojans

ifconfig – Doesn't display PROMISC flag when sniffing.

login – Allows login to any account with the rootkit password.  
If root login is refused on your terminal login as "rewt".  
Disables history logging when backdoor is used.

ls – Hides files listed in /dev/ptyr. All files shown with 'ls -/' if SHOWFLAG enabled.

passwd – Enter your rootkit password instead of old password to become root.

ps – Hides processes listed in /dev/ptyp.

rshd – Execute remote commands as root: rsh -l  
rootkitpassword host command

syslogd – Removes log entries matching strings listed in /dev/ptys.

# Binary Rootkit Detection

## Use non-trojaned programs

- ptree is generally uncompromised
- tar will archive hidden files, the list with -t
- lsof is also generally safe
- Use known good tools from CD-ROM.

## File integrity checks

- tripwire, AIDE, Osiris
- rpm -V -a
- Must have known valid version of database offline or attacker may modify file signatures to match Trojans.

# Library Rootkits

- t0rn rootkit uses special system library libproc.a to intercept process information requested by user utilities.
- Modify libc
  - Intercept system call data returning from kernel, stripping out evidence of attacker activities.
  - Alternately, ensure that rootkit library providing system calls is called instead of libc by placing it in /etc/ld.so.preload

# Kernel Rootkits

Kernel runs in supervisor processor mode

- Complete control over machine.

Rootkits modify kernel system calls

- `execve` modified to run Trojan horse binary for some programs, while other system calls used by integrity checkers read original binary file.
- `setuid` modified to give root to a certain user.

Advantage—Stealth

- Runtime integrity checkers cannot see rootkit changes.
- All programs impacted by kernel Trojan horse.
- Open backdoors/sniff network without running processes.

# Types of Kernel Rootkits

## Loadable Kernel Modules

- Device drivers are LKMs.
- Can be defeated by disabling LKMs.
- ex: Adore, Knark

## Alter running kernel in memory.

- Modify /dev/kmem directly.
- ex: SucKit

## Alter kernel on disk.

# Kernel Rootkit Detection

List kernel modules

- lsmod
- cat /proc/modules

Examine kernel symbols (/proc/kallsyms)

- Module name listed in [] after symbol name.

# Kernel Rootkit Detection

Check system call addresses

- Compare running kernel syscall addresses with those listed in System.map generated at kernel compile.

All of these signatures can be hidden/forged.

# Knark (circa 2004)

- Linux-based LKM rootkit
- Features
  - Hide/unhide files or directories
  - Hide TCP or UDP connections
  - Execution redirection
  - Unauthenticated privilege escalation
  - Utility to change UID/GID of a running process.
  - Unauthenticated, privileged remote execution daemon.
  - Kill -31 to hide a running process.
- modhide: assistant LKM that hides Knark from module listing attempts.



# Lots of Open Source Rootkits Now

- I installed something called Trunkkit
  - <https://git.cse.iitk.ac.in/spramod/cs628a-trunkkit>
- You can also write one!
- Not a lot of code: ~500 or so lines

# Rootkit Detection

## Offline system examination

- Mount and examine disk using another OS kernel+image.
- Knoppix: live CD linux distribution.

## Computer Forensics

- Examine disk below filesystem level.
- Helix: live CD linux forensics tool.

# Rootkit Detection Utilities

## chkrootkit

- Detects >50 rootkits on multiple UNIX types.
- Checks commonly trojaned binaries.
- Examines log files for modifications.
- Checks for LKM rootkits.
- Use `-p` option to use known safe binaries from CDROM.

## carbonite

- LKM that searches for rootkits in kernel.
- Generates and searches frozen image kernel process structures.

# Detection Countermeasures

- Hide rootkit in unused sectors or in unused fragments of used sectors.
- Install rootkit into flash memory like PC BIOS, ensuring that rootkit persists even after disk formatting and OS re-installation.

# Rootkit Recovery

- Restore compromised programs from backup
  - Lose evidence of intrusion.
  - Did you find all the trojans?
- Backup system, then restore from tape
  - Save image of hard disk for investigation.
  - Restore known safe image to be sure that all trojans have been eliminated.
  - Patch system to repair exploited vulnerability.

# Key Points

- Backdoors allow intruder into system without using exploit again.
- Rootkits automatically deeply compromise a system once root access is attained.
- Rootkits are easy to use, difficult to detect.
- Don't trust anything on a compromised system—access disk from a known safe system, like an Ubuntu Live CD.
- Recovery requires a full re-installation of the OS and restoration of files from a known good backup.

# References

1. Oktay Altunergil, "Scanning for Rootkits," <http://www.linuxdevcenter.com/pub/a/linux/2002/02/07/rootkits.html>, 2002.
2. Silvio Cesare, "Runtime kernel kmem patching," <http://vx.netlux.org/lib/vsc07.html>, 1998.
3. William Cheswick, Steven Bellovin, and Avriel Rubin, *Firewalls and Internet Security*, 2<sup>nd</sup> edition, 2003.
4. Anton Chuvakin, "An Overview of UNIX Rootkits," iDEFENSE whitepaper, 2003.
5. Dave Dittrich, "Rootkits FAQ," <http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq>, 2002.
6. Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.
7. Samuel T. King et. al., "SubVirt: Implementing malware with virtual machines", <http://www.eecs.umich.edu/virtual/papers/king06.pdf>, 2006.
8. McClure, Stuart, Scambray, Joel, Kurtz, George, *Hacking Exposed*, 3<sup>rd</sup> edition, McGraw-Hill, 2001.
9. Peikari, Cyrus and Chuvakin, Anton, *Security Warrior*, O'Reilly & Associates, 2003.
10. pragmatic, (nearly) Complete Loadable Linux Kernel Modules, [http://www.thc.org/papers/LKM\\_HACKING.html](http://www.thc.org/papers/LKM_HACKING.html), 1999.
11. Marc Russinovich, "Sony, Rootkits and Digital Rights Management Gone Too Far," <http://blogs.technet.com/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx>
12. Jennifer Rutkowska, "Red Pill: or how to detect VMM using (almost) one CPU instruction," <http://www.invisiblethings.org/papers/redpill.html>, 2004.
13. Ed Skoudis, *Counter Hack Reloaded*, Prentice Hall, 2006.
14. Ed Skoudis and Lenny Zeltser, *Malware: Fighting Malicious Code*, Prentice Hall, 2003.
15. Ranier Wichman, "Linux Kernel Rootkits," <http://la-samhna.de/library/rootkits/index.html>, 2002.

Extra Material Follows

**END OF MODULE 2**



# Isolation Summary

- Access control – mandatory vs discretionary
- Unix school of isolation:
  - users/permissions
- New approaches to isolation
  - chroot, jails and containers
  - System call interposition
  - Software fault isolation
  - Virtual machine monitors

# Isolation Summary (Continued)

- Problem: full isolation is too restrictive
  - Even isolated processes need to communicate through some regulated interface
  - But this ends being the place where attacks occur
- Challenges in isolation
  - Specifying policy
  - Unexpected interactions
  - Covert channels

# Intrusion Detection Systems (IDS)

Another Digression

# Intrusion and Intrusion Detection

- Intrusion : Attempting to break into or misuse your system.
- Intruders may be from outside the network or legitimate users of the network.
- Intrusion can be a physical, system or remote intrusion.

# Different ways to intrude

- Buffer overflows
- Unhandled input
- Unsanitized input
- Race conditions
- ...

# Intrusion Detection Systems (IDS)

Signature Based:

Intrusion Detection Systems look for attack signatures, which are specific patterns that usually indicate malicious or suspicious intent.

Anomaly Detection Based:

Machine learning techniques used to characterize normal behavior from anomalous behavior

# Intrusion Detection Systems (IDS)

- Different ways of classifying an IDS

IDS based on

- anomaly detection
- signature based misuse
- host based
- network based

# Anomaly based IDS

- This IDS models the normal usage of the network as a noise characterization.
- Anything distinct from the noise is assumed to be an intrusion activity.
  - E.g flooding a host with lots of packet.
- The primary strength is its ability to recognize novel attacks.



# Drawbacks of Anomaly detection IDS

- Assumes that intrusions will be accompanied by manifestations that are sufficiently unusual so as to permit detection.
- These generate many false alarms and hence compromise the effectiveness of the IDS.

# Signature based IDS

- This IDS possess an attacked description that can be matched to sensed attack manifestations.
- The question of what information is relevant to an IDS depends upon what it is trying to detect.
  - E.g DNS, FTP etc.

# Signature based IDS (contd.)

- Interpret a certain series of packets, or a certain piece of data contained in those packets, as an attack.
- For example, look for the string “phf” as an indicator of a CGI program attack.
- Signature analysis systems are typically based off of simple pattern matching algorithms. E.g. simply look for a sub string within a stream of data carried by network packets.
- When it finds this sub string (for example, the “phf” in “GET /cgi-bin/phf?”), it identifies those network packets as vehicles of an attack.

# Drawbacks of Signature based IDS

- They are unable to detect novel attacks.
- Have to be programmed again for every new pattern to be detected
- Suffer from false alarms

# Host/Applications based IDS

- The host operating system or the application logs in the audit information.
- Audit information includes events like:
  - Identification/authentication mechanisms (logins)
  - file opens
  - program executions,
  - admin activities etc.
- Audit is analyzed to detect traces of intrusion.

# Drawbacks of the host based IDS

- The kind of information needed to be logged in is a matter of experience.
- Unselective logging of messages may greatly increase the audit and analysis burdens.
- Selective logging runs the risk that attack manifestations could be missed.

# Strengths of the host based IDS

- Attack verification
- System specific activity
- Works on encrypted n/w environments
- Near Real-Time detection and response.
- No additional hardware

# Stack based IDS

- Integrated closely with the TCP/IP stack,
- Allowing packets to be watched as they traverse their way up the OSI layers
- Allows the IDS to pull the packets from stack
  - before the OS or the application have chance to process the packets.



# Network based IDS

- IDS looks for attack signatures in network traffic via a promiscuous interface.
- Filter is usually applied to determine which traffic will be discarded or passed on to attack recognition module

# Strengths of Network based IDS

- Cost of ownership reduced
- Packet analysis
- Real time detection and response
- Malicious intent detection
- Operating system independence

# Commercial ID Systems

- ISS – Real Secure from Internet Security Systems:
  - Real time IDS.
  - Contains both host and network based IDS.
- Tripwire – File integrity assessment tool.
- Zeek and Snort – open source public-domain system.

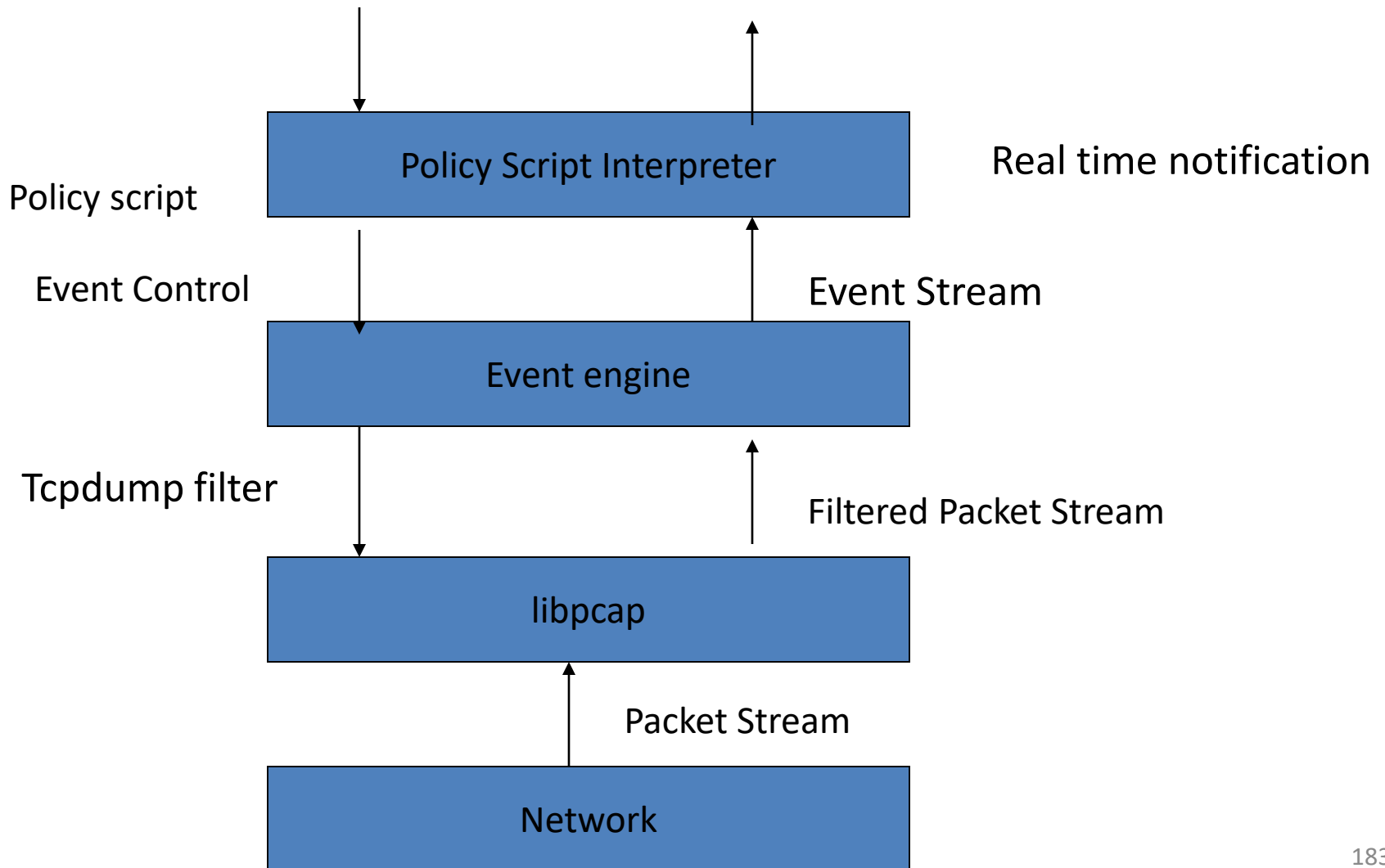
# Zeek: Real time IDS

- Formerly called Bro
- Network based IDS
- Supports many protocols
  - DHCP, DNS, TCP, FTP, ICMP, IMAP, BitTorrent, IRC, SSL, among others
  - <https://docs.zeek.org/en/stable/script-reference/proto-analyzers.html>

# Design goals for Zeek

- High-speed, large volume monitoring
- No packet filter drops
- Real time notification
- Mechanism separate from policy
- Extensible

# Structure of the Zeek System



# Zeek - libpcap

- Packet capture library used by tcpdump.
- Isolates Zeek from network link technology.
- Filters the incoming packet stream from the network
  - E.g. port finger, port ftp, tcp port 113 (Ident), port telnet, port 111 (Portmapper).
  - Can also capture packets with the SYN, FIN, or RST Control bits set.

# Zeek – Event Engine

- The filtered packet stream from the libpcap is handed over to the Event Engine.
- Performs several integrity checks to assure that the packet headers are well formed.
- It looks up the connection state associated with the tuple of the two IP addresses and the two TCP or UDP port numbers.
- It then dispatches the packet to a handler for the corresponding connection.



# Zeek – TCP Handler

- For each TCP packet, the connection handler verifies that the entire TCP Header is present and validates the TCP checksum.
- If successful, it then tests whether the TCP header includes any of the SYN/FIN/RST control flags and adjusts the connection's state accordingly.
- Different changes in the connection's state generate different events.

# Policy Script Interpreter

- The policy script interpreter receives the events generated by the Event Engine.
- It then executes scripts written in the Zeek language which generates events like logging real-time notifications, recording data to disk or modifying internal state.
- Adding new functionality to Zeek consists of adding a new protocol analyzer to the event engine and then writing new events handlers in the interpreter.

# Future of IDS

- To integrate the network and host based IDS for better detection.
- Developing IDS schemes for detecting novel attacks rather than individual instantiations.

# Looking Forward

- In the 90 and 00s, attacks were pretty dumb
  - IDS could be somewhat effective
- Now, advanced persistent threats (APTs) are becoming more and more common
  - Vodafone Greece Hack (2007)
  - Operation Aurora (2009)
  - Stuxnet (2010)
  - German Parliament network (2014)
  - DNC Hack (2015-ish)

# How do we deal with APTs?

- We don't have good tools to deal with them
- Software is still hopelessly insecure while attacks keep getting smarter
- New problems due to adversarial machine learning
- Need more focus on reliable defences
  - Safe programming languages
  - Newer hardware architectures (enclaves etc.)
  - Automated verification

# Homework

Think about different isolation mechanisms

- Why do we have so many?
- What are the trade-offs, pros- and cons- of each?
- What attacks are prevented by a particular class of mechanisms that aren't prevent by another?
- Can these mechanisms be stacked one on top of the attack? If so, which ones?
- What assumptions are being by these mechanisms? What is trusted?

# VMM Introspection: [GR' 03]

protecting the anti-virus system

# Intrusion Detection / Anti-virus

Runs as part of OS kernel and user space process

- Kernel root kit can shutdown protection system
- Common practice for modern malware

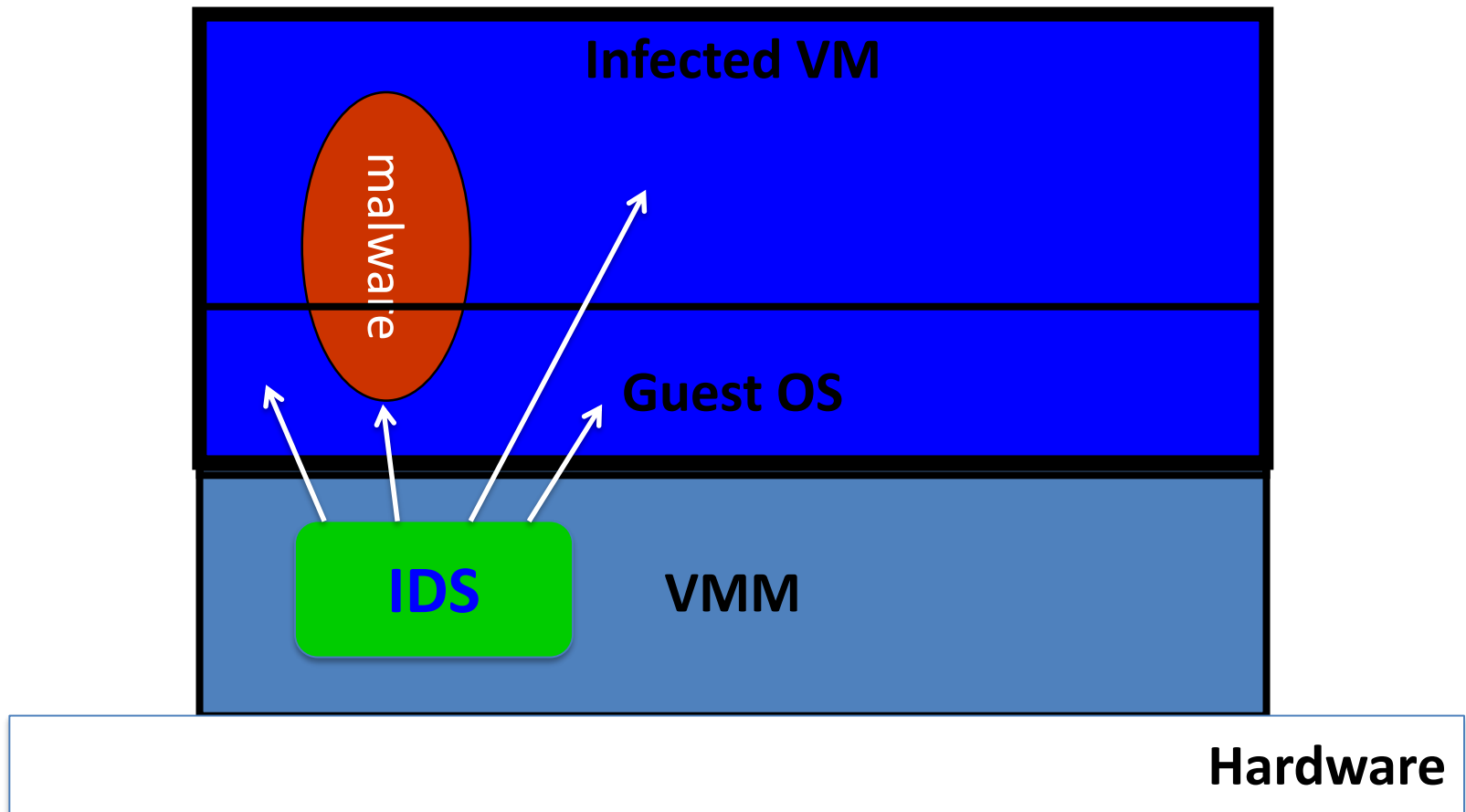
Standard solution:     **run IDS system in the network**

- Problem:   insufficient visibility into user's machine

Better:   **run IDS as part of VMM (protected from malware)**

- VMM can monitor virtual hardware for anomalies
- VMI:   Virtual Machine Introspection
  - Allows VMM to check Guest OS internals





# Sample checks

## **Stealth root-kit malware:**

- Creates processes that are invisible to “ps”
- Opens sockets that are invisible to “netstat”

### **1. Lie detector check**

- Goal: detect stealth malware that hides processes and network activity
- Method:
  - VMM lists processes running in GuestOS
  - VMM requests GuestOS to list processes (e.g. ps)
  - If mismatch: kill VM

# Sample checks

## 2. **Application code integrity detector**

- VMM computes hash of user app code running in VM
- Compare to whitelist of hashes
  - Kills VM if unknown program appears

## 3. **Ensure GuestOS kernel integrity**

- example: detect changes to `sys_call_table`

## 4. **Virus signature detector**

- Run virus signature detector on GuestOS memory



# Isolation

---

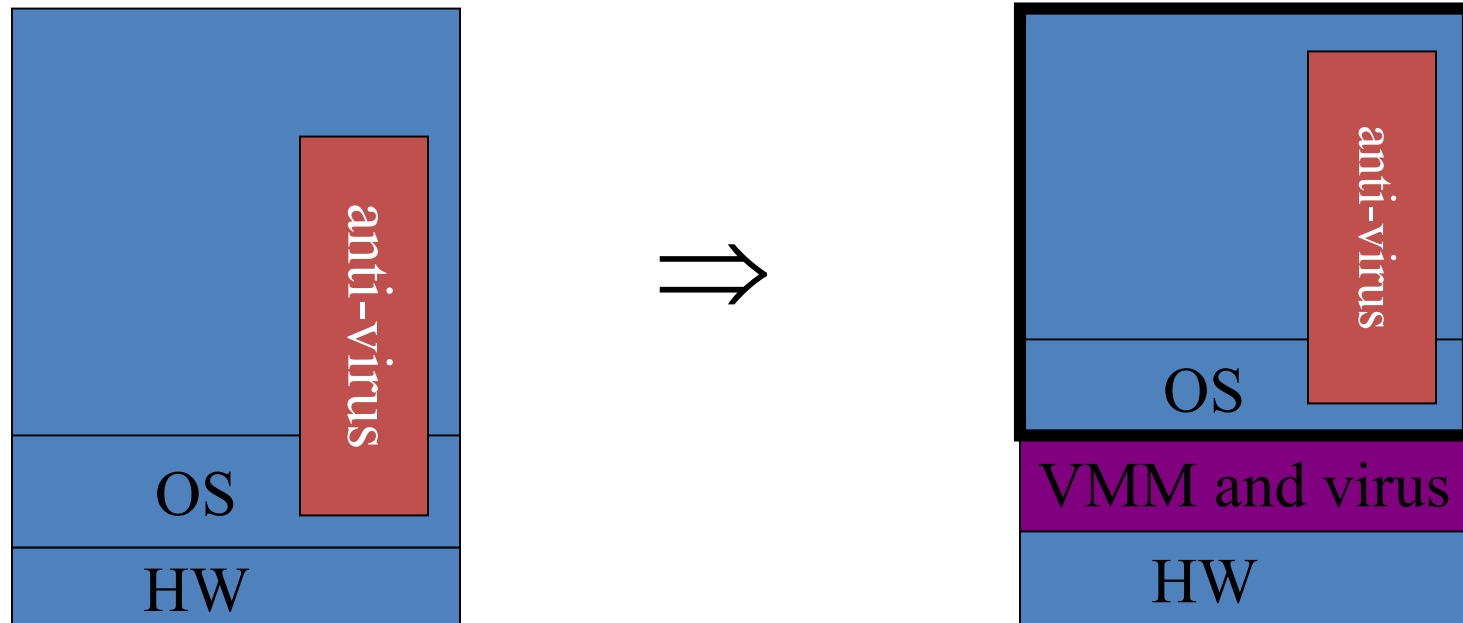
## Subverting VM Isolation

# Subvirt

[King et al. 2006]

Virus idea:

- Once on victim machine, install a malicious VMM
- Virus hides in VMM
- Invisible to virus detector running inside VM



# The MATRIX







# VM Based Malware (blue pill virus)

- **VMBR:** a virus that installs a malicious VMM (hypervisor)
- **Microsoft Security Bulletin: (Oct, 2006)**
  - Suggests disabling hardware virtualization features by default for client-side systems
- **But VMBRs are easy to defeat**
  - A guest OS can detect that it is running on top of VMM



# VMM Detection

Can an OS detect it is running on top of a VMM?

## Applications:

- Virus detector can detect VMBR
- Normal virus (non-VMBR) can detect VMM
  - refuse to run to avoid reverse engineering
- Software that binds to hardware (e.g. MS Windows) can refuse to run on top of VMM
- DRM systems may refuse to run on top of VMM

# VMM detection (red pill techniques)

- VM platforms often emulate simple hardware
  - VMWare emulates an ancient i440bx chipset
    - ... but report 8GB RAM, dual CPUs, etc.
- VMM introduces time latency variances
  - Memory cache behavior differs in presence of VMM
  - Results in relative time variations for any two operations
- VMM shares the TLB with GuestOS
  - GuestOS can detect reduced TLB size
- ... and many more methods [**GAWF' 07**]

# VMM Detection

Bottom line: **The perfect VMM does not exist**

VMMs today (e.g. VMWare) focus on:

Compatibility: ensure off the shelf software works

Performance: minimize virtualization overhead

- VMMs do not provide **transparency**
  - **Anomalies reveal existence of VMM**



# Isolation

---

## Software Fault Isolation

# Software Fault Isolation [Whabe et al., 1993]

**Goal:** confine apps running in same address space

- Codec code should not interfere with media player
- Device drivers should not corrupt kernel

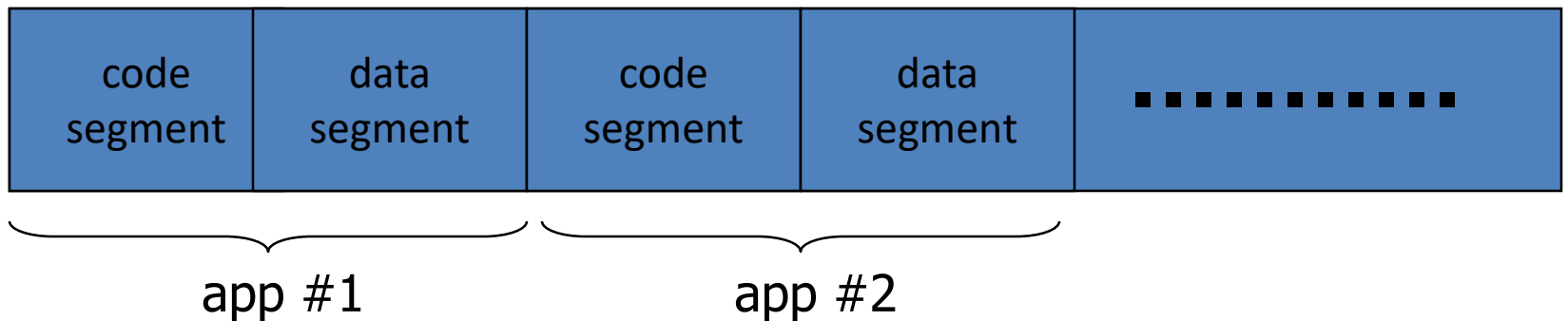
Simple solution: runs apps in separate address spaces

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

# Software Fault Isolation

SFI 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 M
- dr1, dr2:** de
  - compiler pretends that segments don't exist
  - **dr2** contains segment ID
- Indirect load instruction **R12 ← [R34]** becomes:

Guard ensures code does not  
load data from another segment

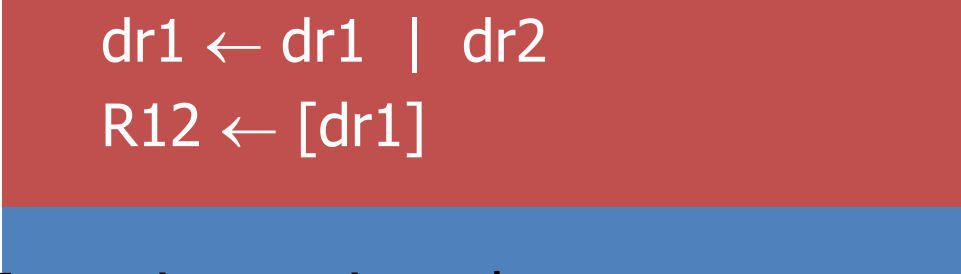
```
dr1 ← R34
scratch-reg ← (dr1 >> 20)
compare scratch-reg and dr2
trap if not equal
R12 ← [dr1]
```

```
: get segment ID
: validate seg. ID

: do load
```

# Address sandboxing technique

- **dr2:** holds segment ID
- Indirect load instruction **R12 ← [R34]** becomes:



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 places on all unsafe instructions

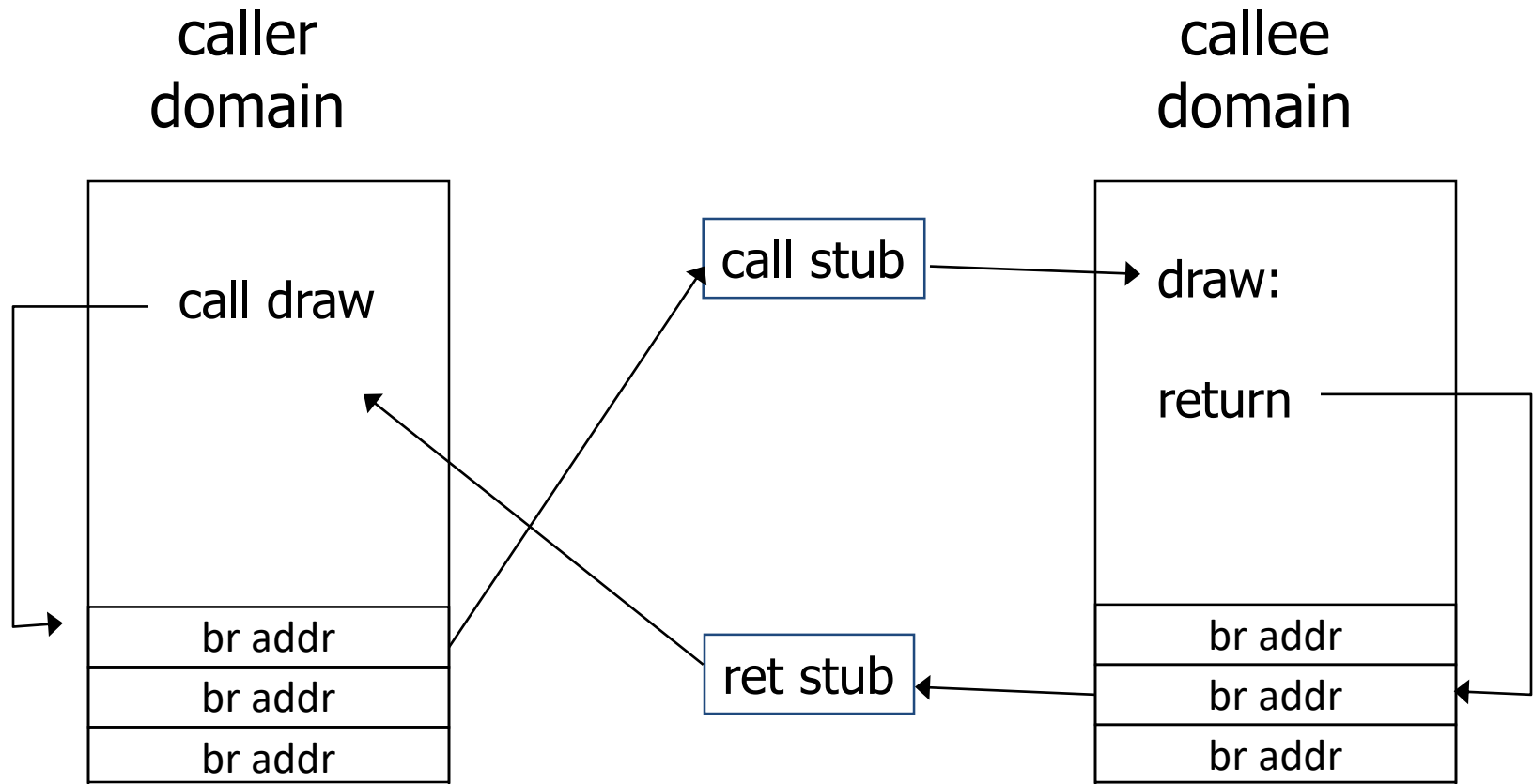


**Problem:** what if `jmp [addr]` jumps directly into indirect load?  
(bypassing guard)

**Solution:**

`jmp` guard must ensure `[addr]` does not bypass load guard

# Cross domain calls



- Only stubs allowed to make cross-domain jumps
- Jump table contains allowed exit points
  - Addresses are hard coded, read-only segment

# SFI Summary

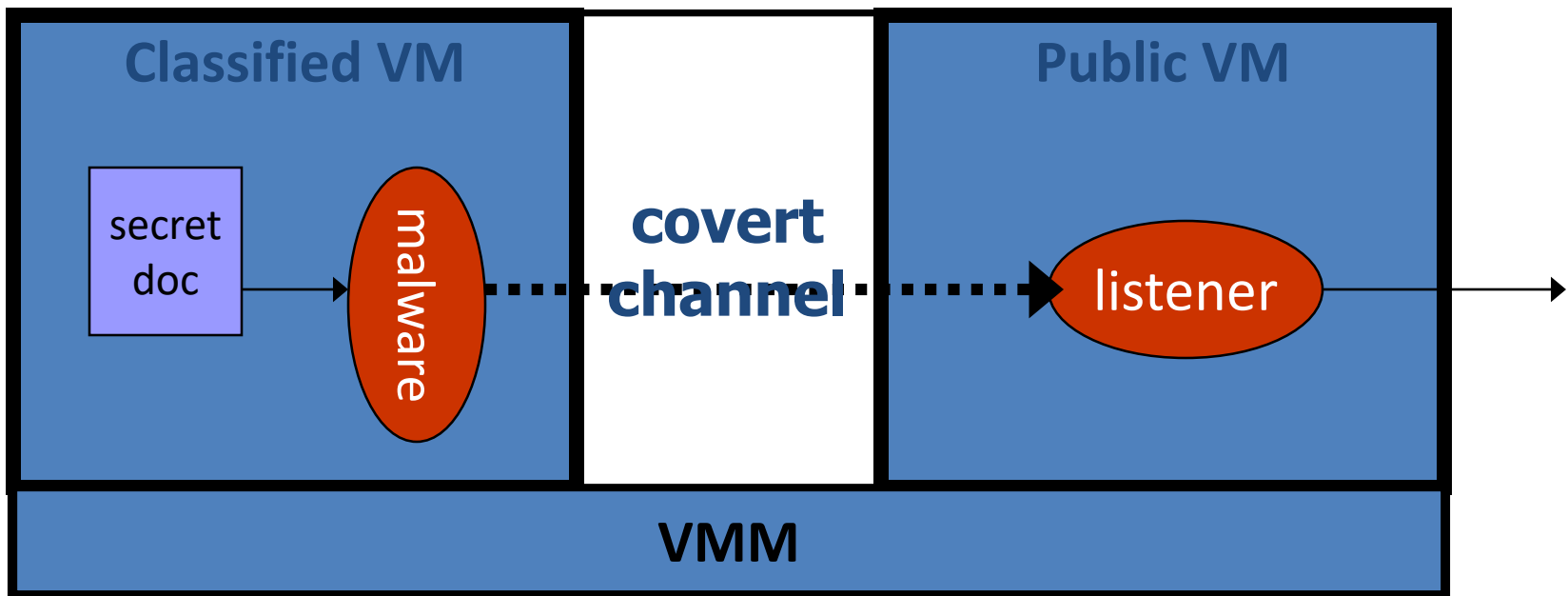
- Shared memory: use virtual memory hardware
  - map same physical page to two segments in addr space
- Performance
  - Usually good: mpeg\_play, 4% slowdown
- Limitations of SFI: harder to implement on x86 :
  - variable length instructions: unclear where to put guards
  - few registers: can't dedicate three to SFI
  - many instructions affect memory: more guards needed

# Isolation: summary

- Many sandboxing techniques:
  - Physical air gap, Virtual air gap (VMMs),*
  - System call interposition, 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

# 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

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

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