Protection from Untrusted Interaction

- Vulnerabilities
- Malware
- Insufficient Control
- User-oriented access control (DAC, MAC, RBAC, …): it is typical for active entities (known as ‘subjects’) to have access to all the user's privileges regardless of the privileges actually required by the program running.

Running untrusted code

Need to run buggy/unstrusted code:

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

<u>Goal</u>: if application "misbehaves" $\Rightarrow$ kill it

User Oriented Control

- User-oriented access controls do not sufficiently mitigate malware threat
  - Man-in-the-middle attacks can intercept communications between hosts and insert malware via trusted websites and hosts.
    - even intercept “secure” encrypted communications .
  - Viruses copy themselves to other programs.
  - Worms propagate across networks, often by exploiting software vulnerabilities.
  - Trojan horses pose as legitimate programs.
  - Malware can be sent via email in targeted attacks.

Trust Based Execution (1)

- One of the simplest access control techniques to mitigate the risk of running programs with all of the user's authority is to only allow particular programs to run.
  - Microsoft AppLocker and Microsoft Software Restriction Policies (SRP)
  - White list, black list
- Using this approach, processes typically still have all the authority of the user;
  - however, only those programs deemed trustworthy (or not “untrusted”) are allowed to run.

Trust Based Execution (2)

- analyse source code or binary files to decide whether the program is trusted to run.
- used by many of the current generation of anti-malware suites, typically based on attributes that identify code as untrusted.
- Signature-based and heuristic lists identify programs based on characteristics of executable files and are typically used to specify black lists to prevent known types of malware from running.
- The techniques used to identify malware have become increasingly sophisticated, as have the various techniques employed by malware to avoid detection

Trust Based Execution (3)

- Reputation-based security, used by systems such as Symantec Quorum, Microsoft SpyNet, and McAfee Artemis, is a relatively new technique, that uses information collected from a large number of users to make judgements about the likely trustworthiness of programs to decide if programs should be trusted to run.

Trust based Execution (4)

- Trust-based security systems provide protection from a limited number of specific threats.
  - many legitimate "trusted" programs can be the source of malicious behaviour: for example, well-intended software authors may accidentally introduce design or implementation flaws, resulting in software vulnerabilities that enable attackers to execute malicious code.
- These approaches do not provide a mechanism for safely running programs without trusting them to run with all of a user's authority.
  - it is not ideal to have to complete trust on any software.
- Furthermore, in many cases it is overly restrictive or impractical to prohibit users from running code obtained from third parties via the Internet.
  - For example, mobile phone "apps" are currently very popular, and the web is becoming increasingly dynamic, including client-side execution of mobile code.
  - All of these mechanisms can fail to protect users from malware.

Trust Based Executions (5)

- All of the above mechanisms can fail to protect users from malware.
  - digital signatures and certificates have failed to accurately reflect the actual origin of programs
- ActiveX has been a prevalent infection vector
- anti-malware black list techniques have failed to identify malware

Application-oriented Access Control

- Restrict subjects based on the identity of the application or process, rather than just the identity of the user.
- This approach is designed to limit the ability of applications to access resources outside of those they require to perform legitimate actions.
- Application-oriented controls can restrict the damage caused by malware or exploited vulnerabilities by limiting the software to those actions authorised by whoever configures the security policy, whether end users, administrators, or software developers.

Confinement (1)

- Ensure misbehaving Application (app) cannot harm rest of system
- Various levels of Implementation
  - **Hardware**: run application on isolated hw
    - Difficult to manage (with network needs)
  - **Virtual machines**: isolate OS's on a single machine
  - **Process:** System Call Interposition--

    Isolate a process in a single operating system

Confinement (2)

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

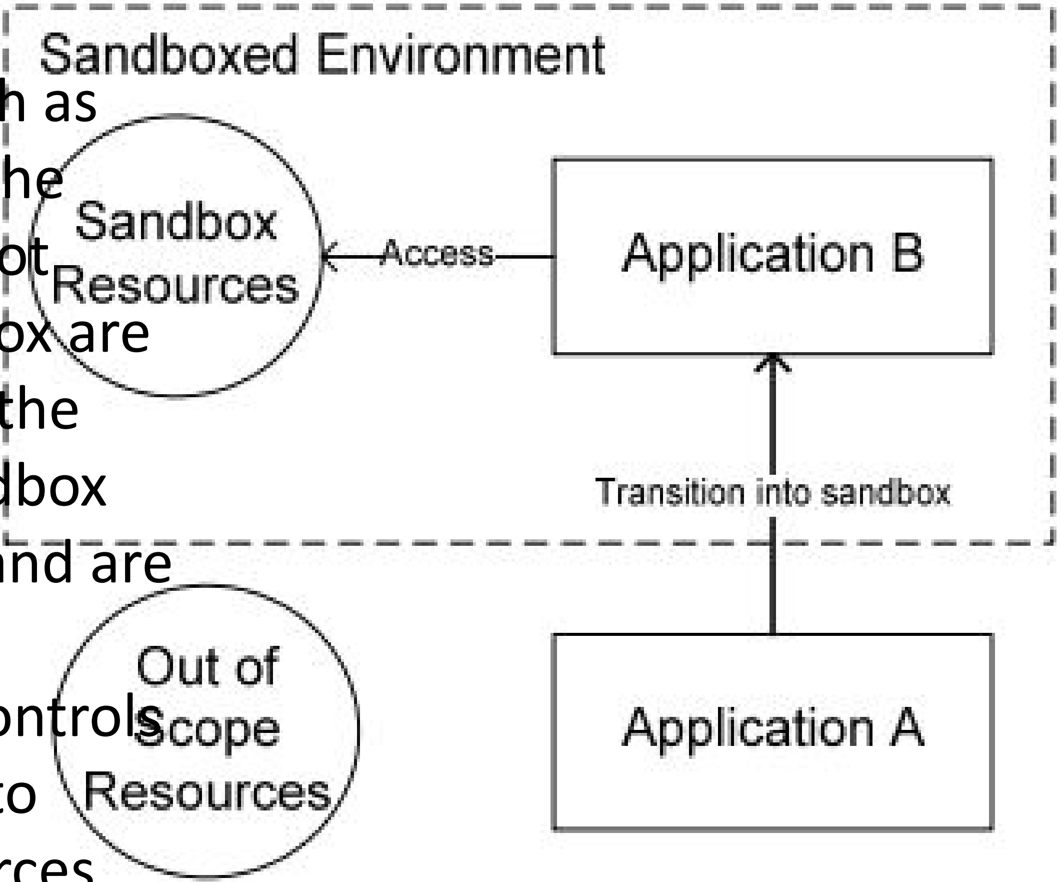
- **Application**: e.g. browser-based confinement

Isolation (Confinement) : Sandboxes and Virtualisation

- One way to restrict a program's ability to access resources is to run it in an environment where the application can only access objects within their so-called ‘sandbox’.
- Typically sandboxes only apply to programs explicitly launched into or from within a sandbox.
  - In most cases no security context changes take place when a new process is started, and all programs in a particular sandbox run with the same set of rights.
- Sandboxes can either be permanent where resource changes persist after the programs finish running, or ephemeral where changes are discarded after the sandbox is no longer in use

Isolation Based Sandbox

Applications (such as Application A in the figure) that are not within the sandbox are typically outside the scope of the sandbox access controls, and are therefore (other security controls permitting) free to access any resources, including those within a sandbox.



Container Based Sandboxes

- Share the kernel but have separate user-space resources.
- More efficient system level virtualization
- Chroot, jails, linux containers

  (Discussed already)

Examples

From Dan Boneh

Implementing confinement

Key component: **reference monitor**

- **Mediates requests** from applications
  - Implements protection policy
  - Enforces isolation and confinement
- Must <u>**always**</u> be invoked:
  - Every application request must be mediated
- **Tamperproof**:
  - Reference monitor cannot be killed
  - … or if killed, then monitored process is killed too
- **Small** enough to be analyzed and validated

A old example: chroot

Often used for “guest” accounts on ftp sites

To use do: (must be root)

chroot /tmp/guest root dir “/” is now “/tmp/guest”

su 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

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"
- 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

Stronger mechanism than simple chroot

<u>To run</u>: **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

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

Chroot()

- Changes root directory for a process and its children
- The namespaces of the application limits it to only access files inside specified directory tree
- A wrapper program “chroot” can be used to launch programs into a “chroot jail”

Chroot()

- Only root can perform but should change identity as soon as possible
- Root an escape a chroot jail by another chroot()
- There are resources such as process controls and networking that are not mediated
- Mechanisms like FreeBSD jails solve some of these problems.

System Call interposition

- These schemes filter system calls
  - Systrace, janus, Etrace, TRON
- MAPbox: Confines programs based on behaviour classes
- De-Merits: System calls can be complex
  - Combined in various ways as they were not designed as a security interface.

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)
- Completely user space (e.g. program shepherding)
- Hybrid (e.g. Systrace)

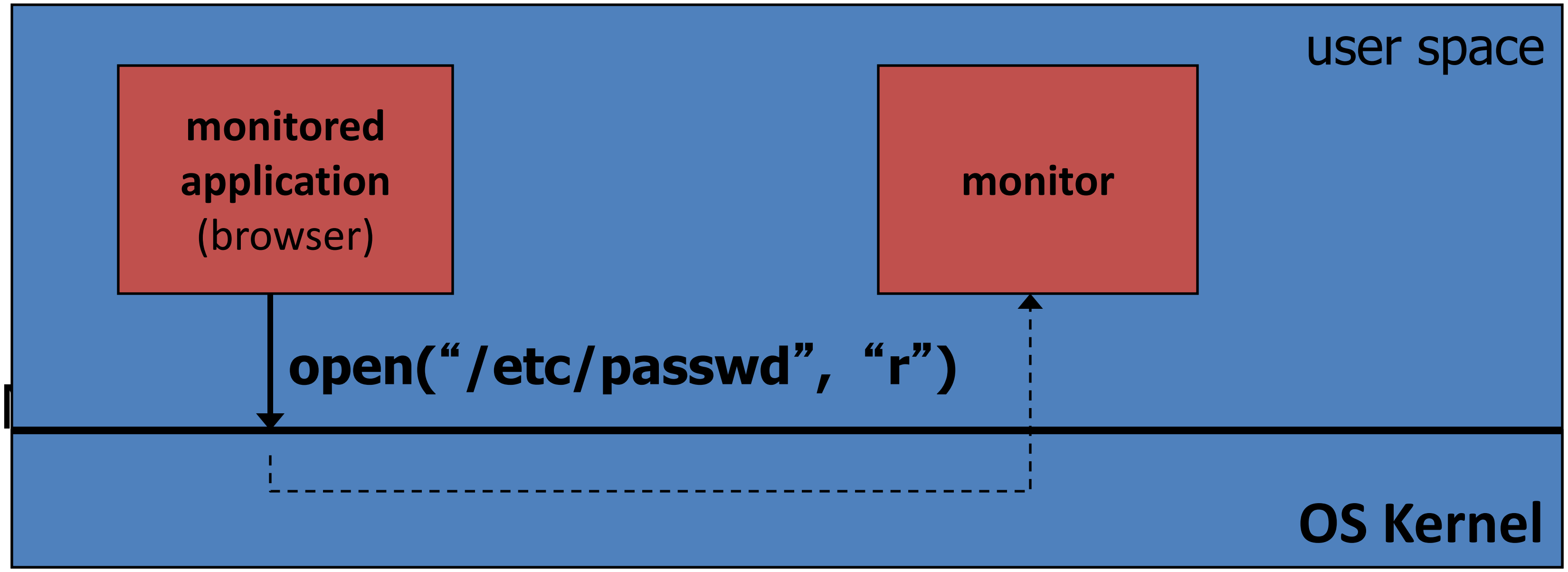
Initial implementation (Janus)

[GWTB'96]

Linux **ptrace**: process tracing

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

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



Complications

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

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

- If app forks, monitor must also fork
  - forked monitor monitors forked app
- If monitor crashes, app must be killed
- Monitor must maintain all OS state associated with app
  - current-working-dir (**CWD**), **UID, EUID, GID**
  - When app does "cd path" monitor must update its CWD
    - otherwise: relative path requests interpreted incorrectly

Problems with ptrace

Ptrace is not well suited for this application:

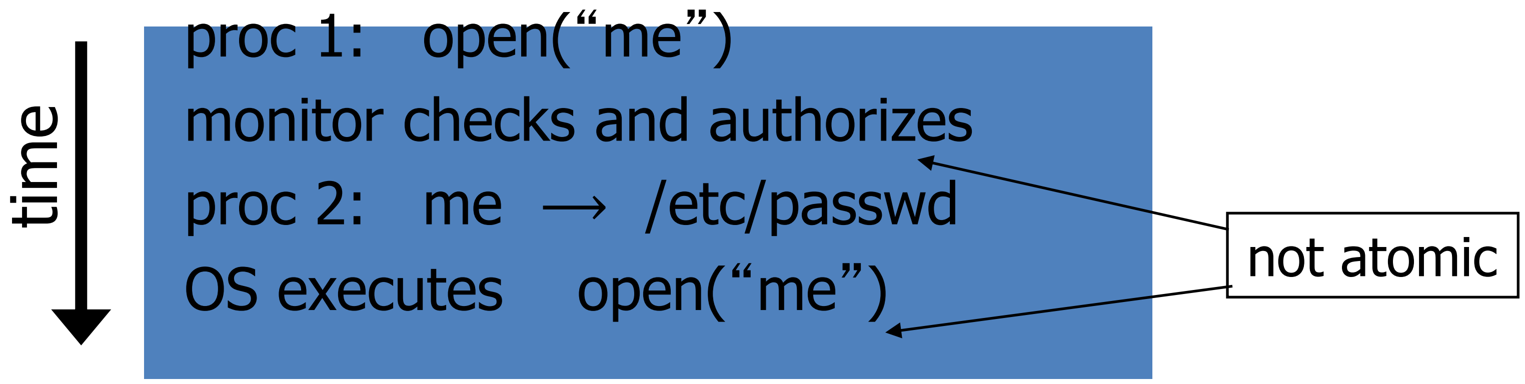
– Trace all system calls or none

inefficient: no need to trace "close" system call

– Monitor cannot abort sys-call without killing app

Security problems: **race conditions**

– <u>Example</u>: symlink: me $\longrightarrow$ mydata.dat


proc 1: open("me")

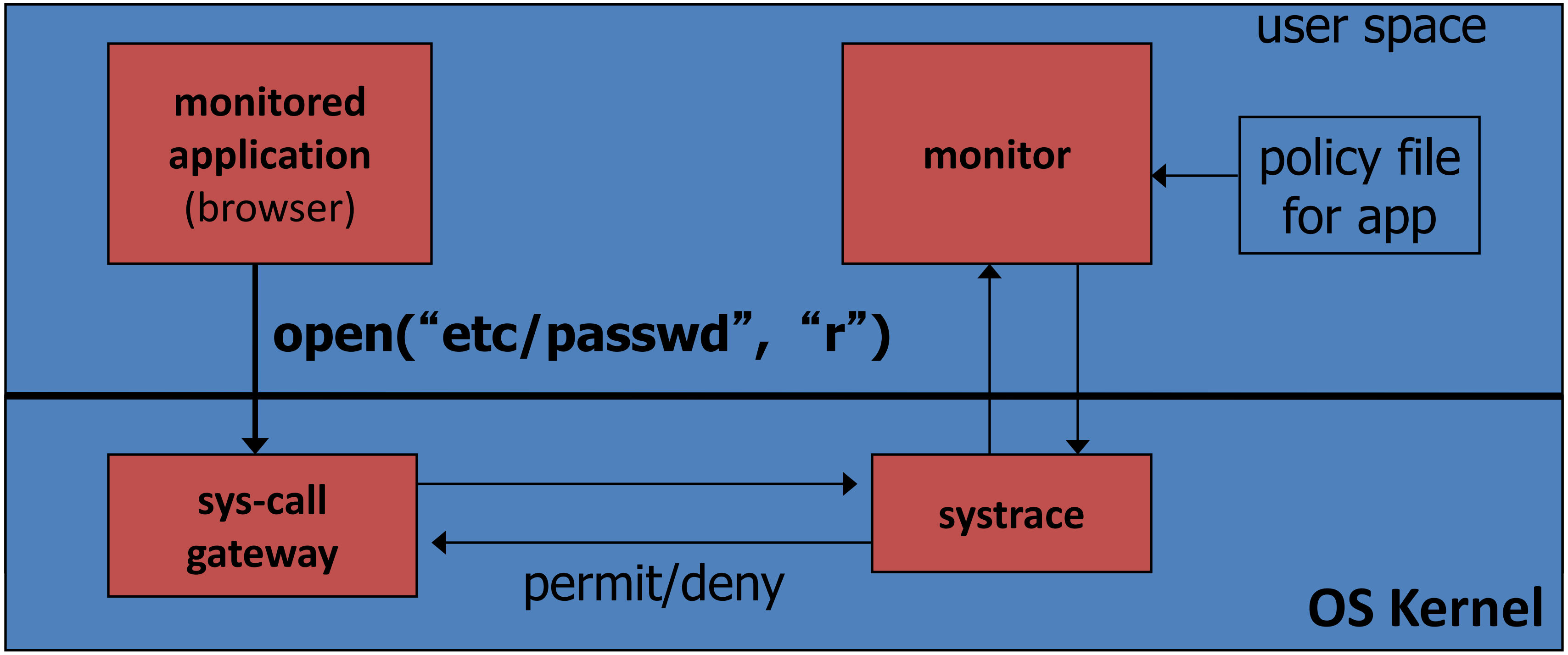
monitor checks and authorizes

proc 2: me $\longrightarrow$ /etc/passwd

OS executes open("me")

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

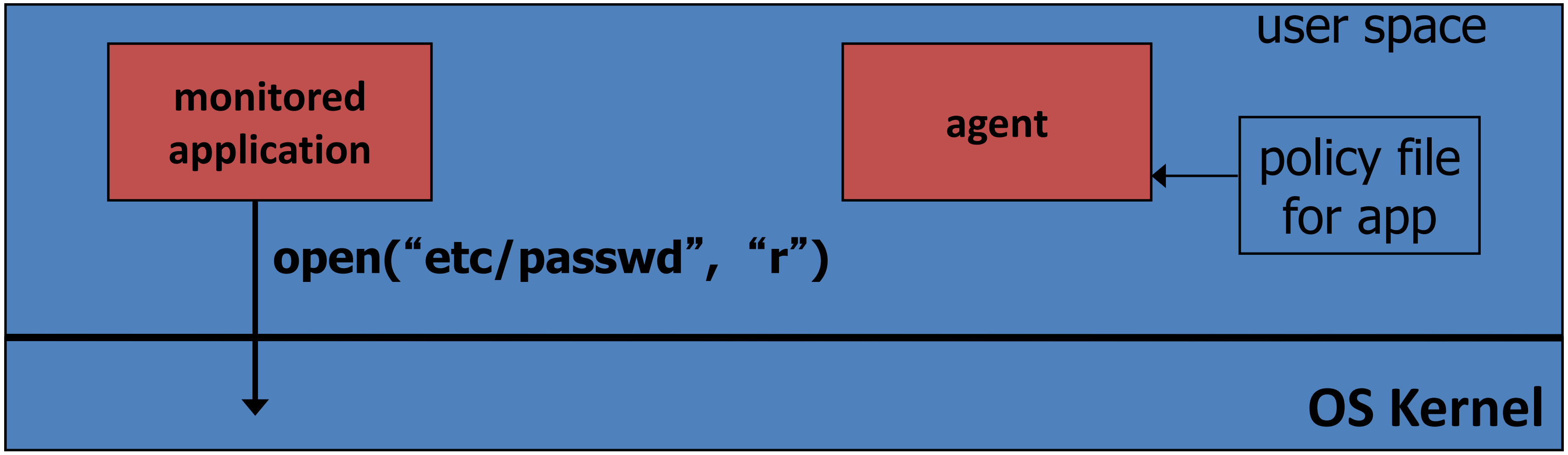
Ostia: a delegation architecture

[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:



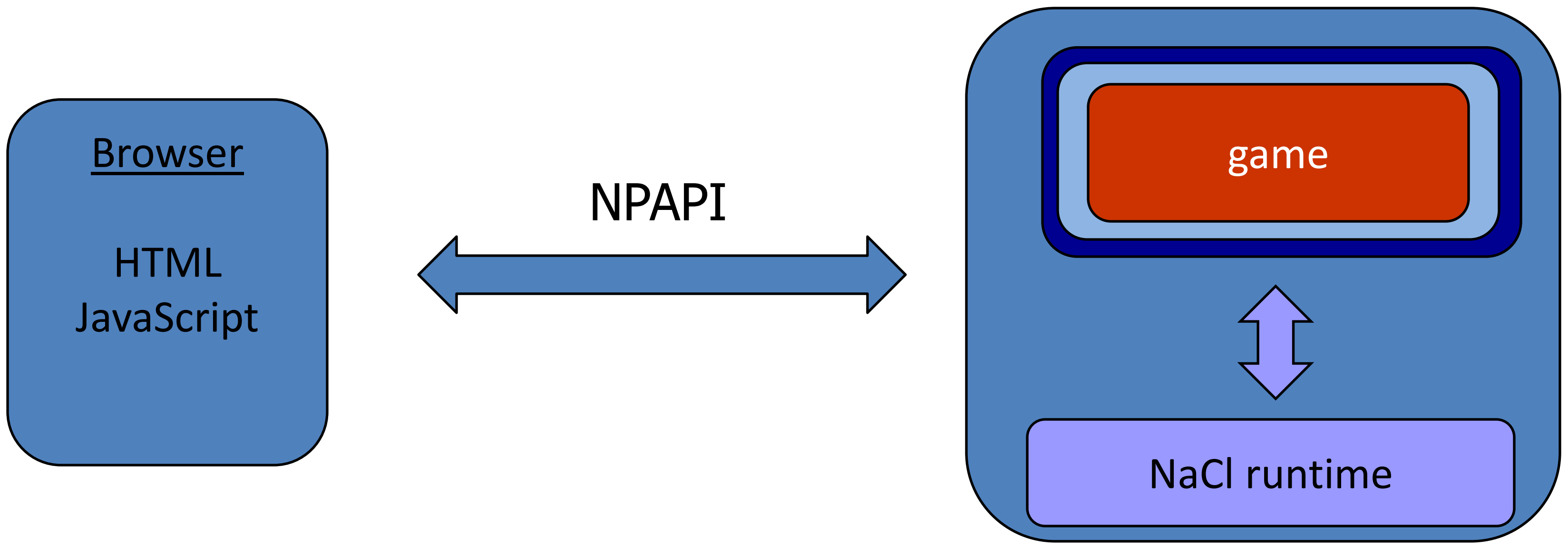
Ostia: a delegation architecture

[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.

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

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

Copy on write Sandboxes

- Allow applications to read all files and all writes are written to a separate area.
- Upon termination ask for writes that are to be kept
- Examples Alcatraz

Linux Capabilities

- On Unix two types of users
  - Privileges (uid =0)
  - Unpriv (uid != 0)
- Root(0) can do anything
  - Bypasses all kernel permission checks
- Capabilities divide these privileges
- Include CAP_CHROOT

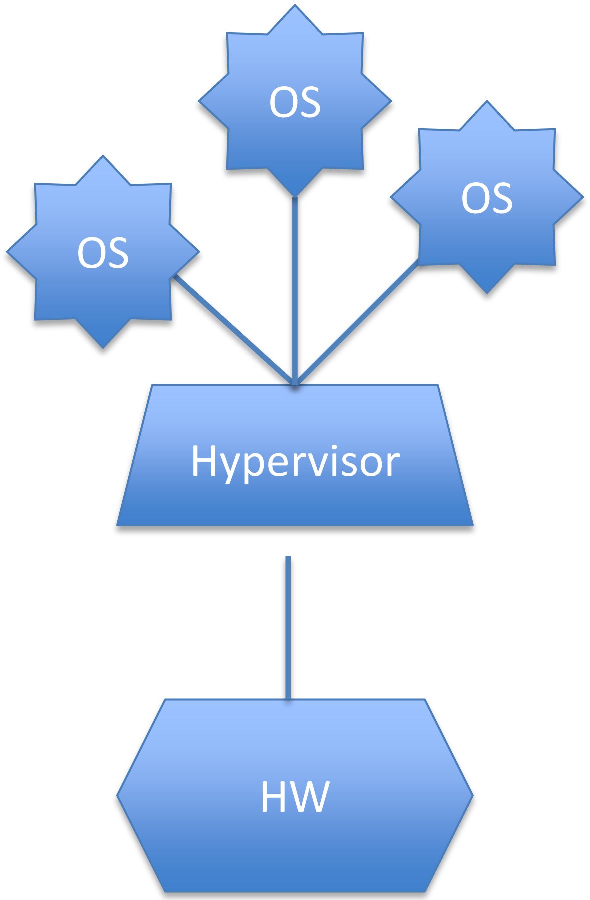
Sandboxes → VM

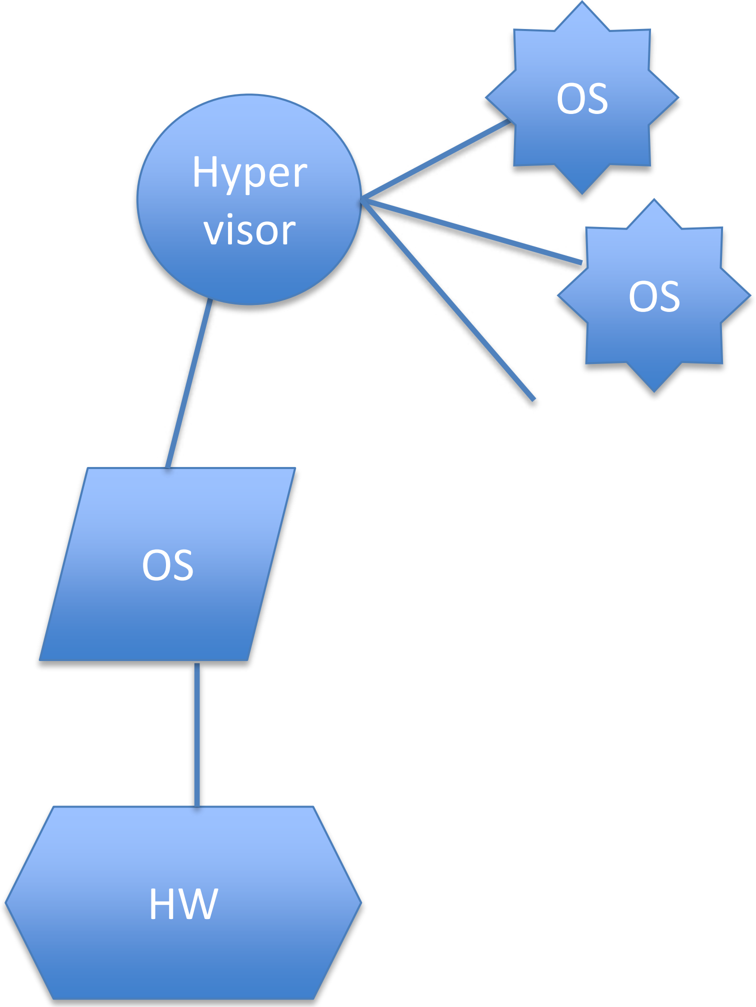
- Most sandboxes provide an isolation-based approach where the effect of programs run inside a sandbox is entirely isolated from resources outside the sandbox's authority. However, due to practical requirements, sandboxing schemes often provide ways of circumventing this isolation in order to copy data into and out of sandboxes.

- System-level sandboxes provide complete operating environments to confined applications. One way of achieving this is through the use of hardware-level virtual machines (VMs). A virtual machine monitor (VMM) can be used to multiplex the physical hardware between multiple self-contained fully virtualised VM operating environments, each containing a complete operating system.

Isolation: System Level Sandboxes

- System level sandboxes provide a complete environment for OS
- Virtualization: A hypervisor (virtual machine monitor (VMM), can multiplex hardware to run hardware level virtual machines (VM)





Type I

Type II

Isolation: System Level Sandboxes

- HW Emulation Base: The guest OS need not know it is being virtualized
  - Vmware, VirtualBox
- Para-virtulaization (Software emulation)
  - The guest knows they are being virtualized and used the API provided by the virtualization
  - Can be more efficient since work can be done by the host
  - Xen, User-mode Linux

Isolation: System Level Sandboxes

- Qubes
  - A VM for each different task
- From Security Perspective lot of uses
  - Separation, isolation, high availability, disaster recovery, multiple OS's etc.
- Can hardware emulation VMs be used to confine individual application?
- From an end-user perspective, hard to manage

Self Contained App

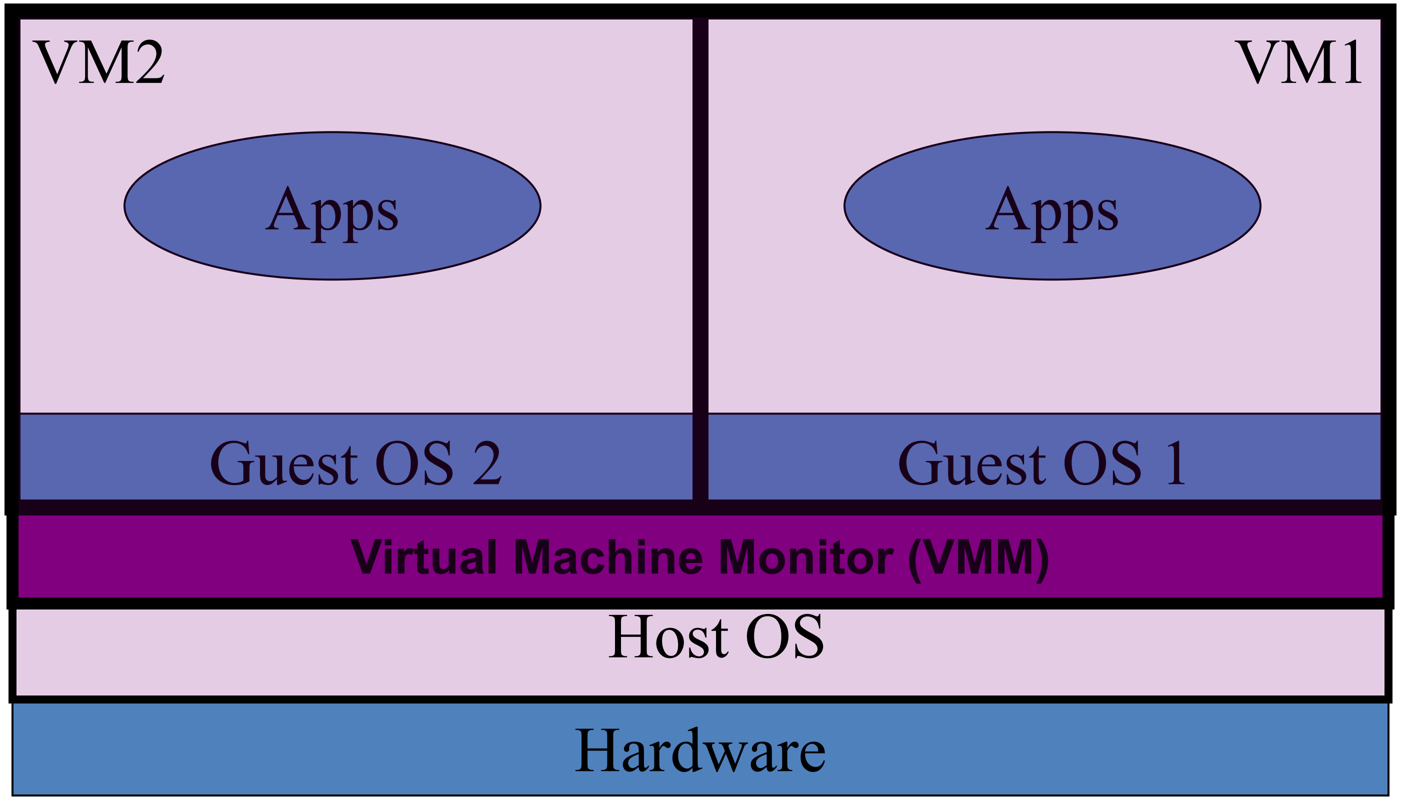
- Force each app to be self contained with no ambient authority to other resources (authorized via user intervention)
- Eg., Java applets, google native code

Isolation

- Adv: Good for shared servers, isolating completely separate systems
- De-Merits:
  - Redundancy of resources (say OS)
  - Inhibit sharing
  - Worlflow and usability

VM from Dan Boneh

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 <u>guest</u> OS and guest apps
- But malware cannot escape from the infected VM
  - Cannot infect <u>host</u> OS
  - Cannot infect other VMs on the same hardware

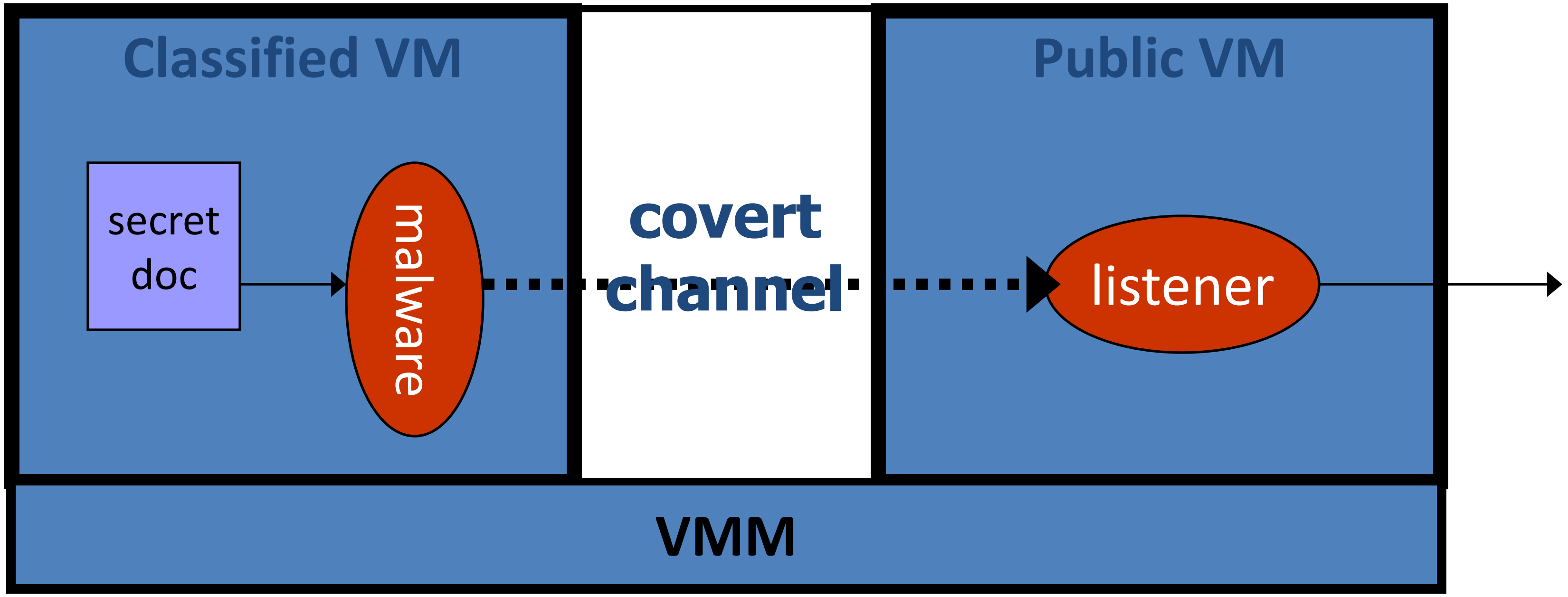
Requires that VMM protect itself and is not buggy

- 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

VMM Introspection: [GR’ 03]

protecting the anti-virus system

Intrusion Detection / Anti-virus

Runs as part of OS kernel and user space process

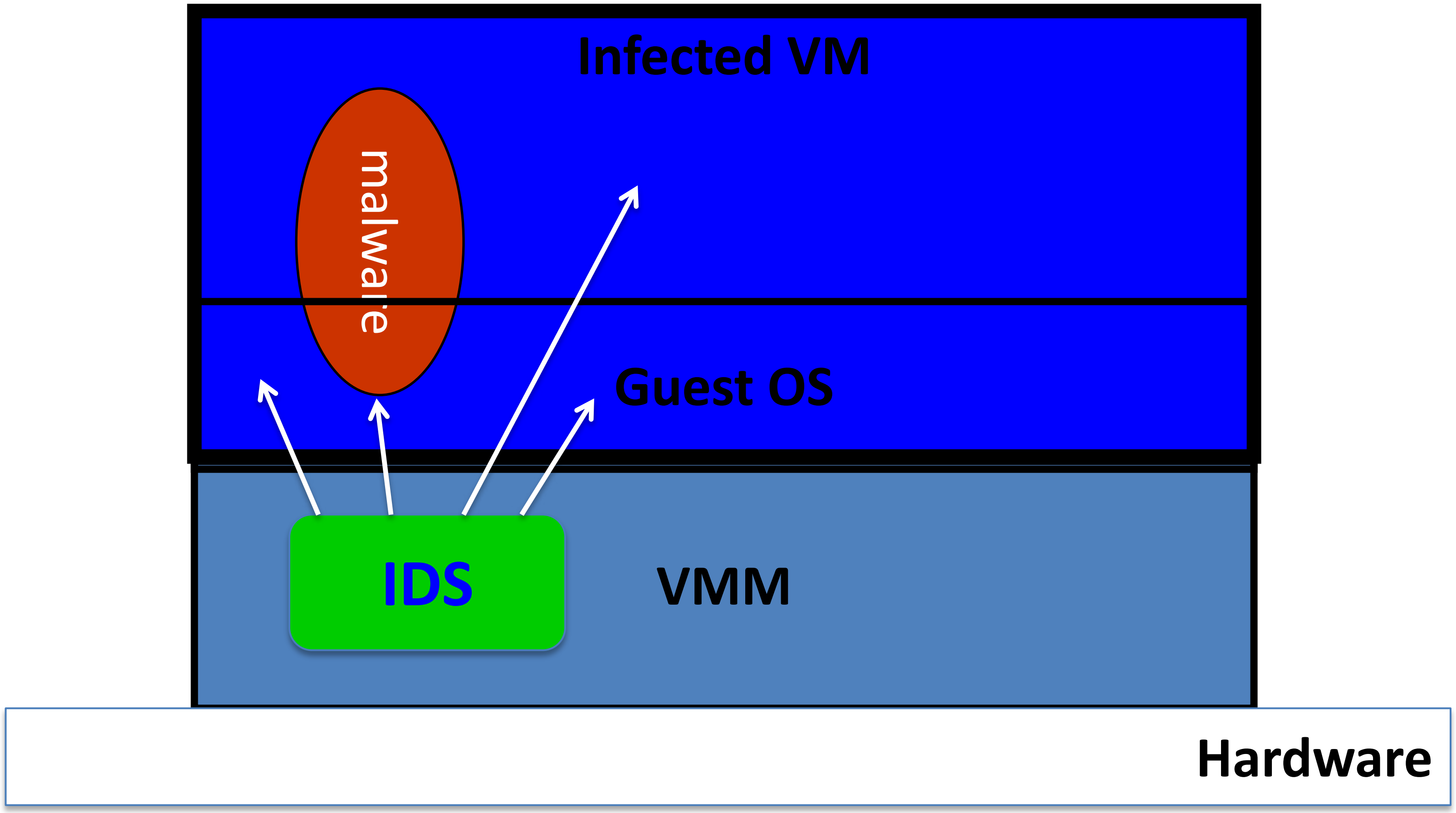
- Kernel root kit can shutdown protection system
- Common practice for modern malware

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


Infected VM
malware
Guest OS
IDS
VMM
Hardware

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

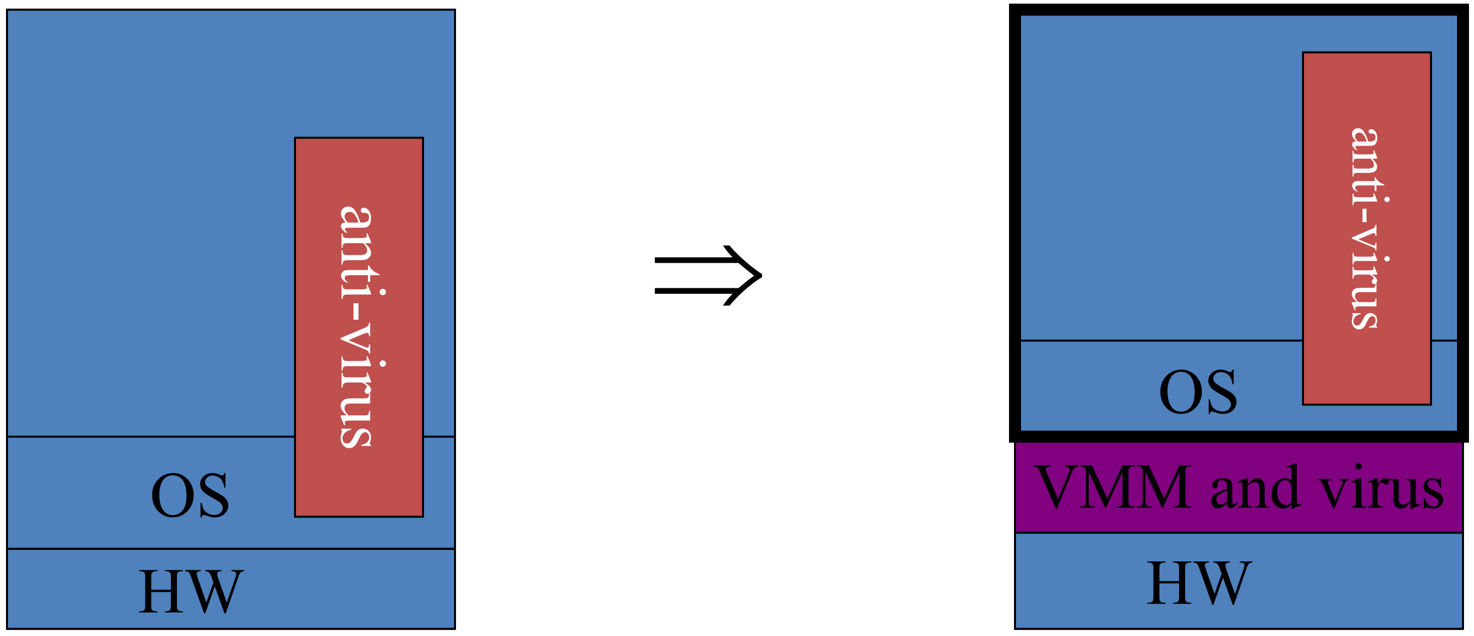
4. **Virus signature detector**
   - Run virus signature detector on GuestOS memory

Subvirting 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



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

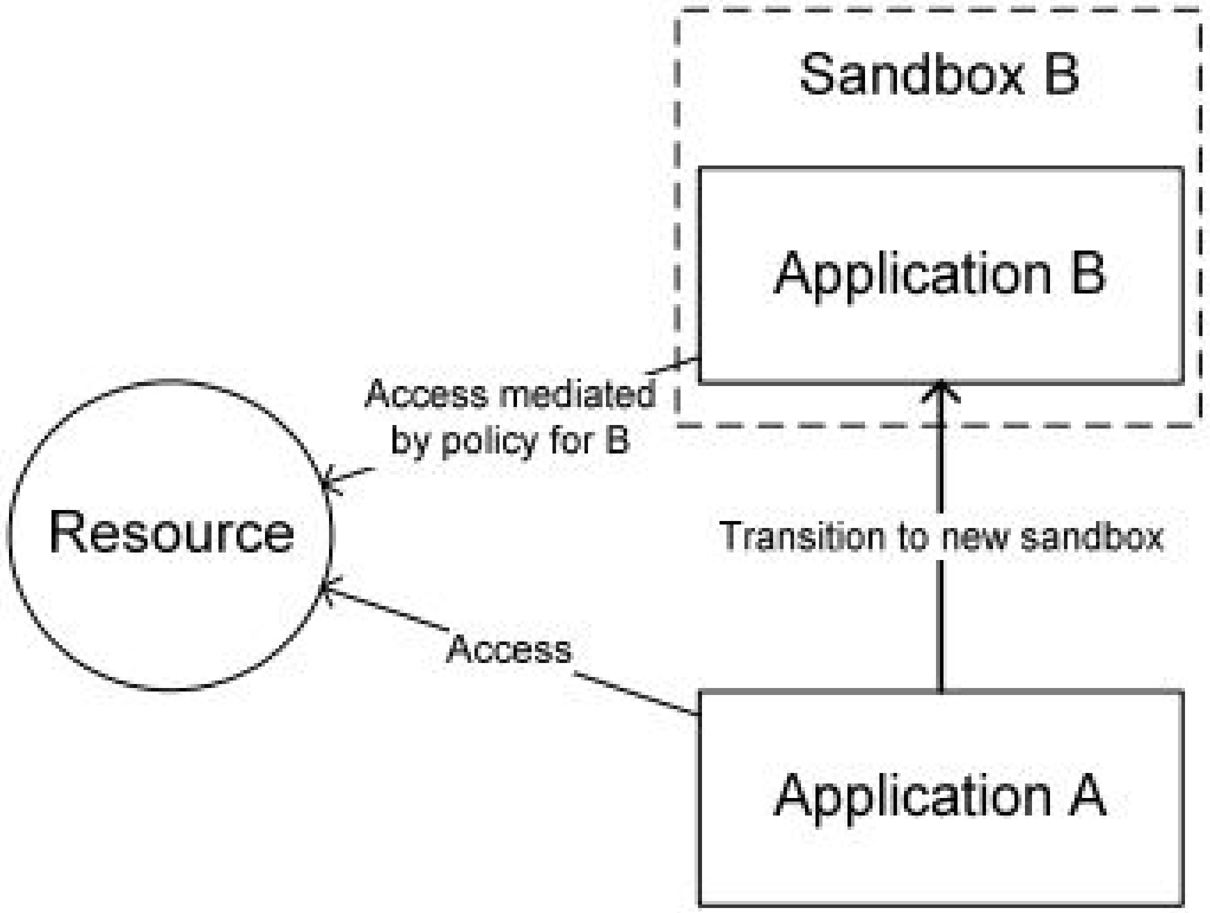
Performance: minimize virtualization overhead

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

Rule Based Sandboxes

- Control what each application can do
- Program is launched into a sandbox and can enforce file access

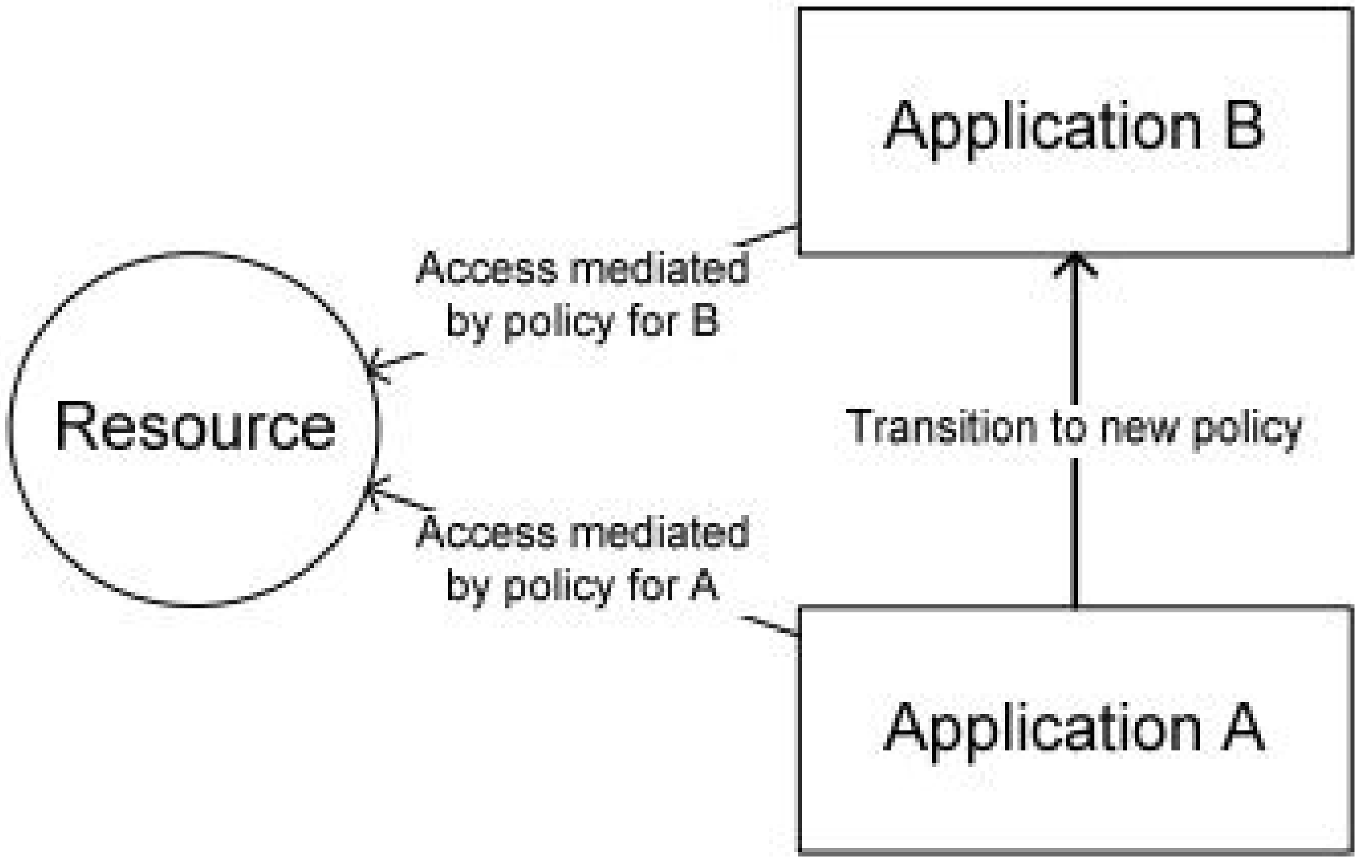
Rule Based Sandbox



Rule-Base System wide Controls

- They don’t require applications to be launched into a sandbox
- Applied to applications that have policies
- Often MAC may be the control
- When one application starts another policy transition may occur, changing the policy that is applied.

Rule Based System Wide Control


Application B
Access mediated by policy for B
Resource
Transition to new policy
Access mediated by policy for A
Application A

Rule Base System Wide Controls

- Coarse Grained
  - Android
  - Camera, GPS access
  - Linux capabilities: break up root permissions so that other privileges are dropped
    - Eg., grant raw network access without granting all of roots other privileges
- Disadv: Not all permissions can be specified in this manner (files for example)

Structure:

Schreuders, Z.C., McGill, T. and Payne, C. (2012) The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls. Computers & Security, 32 . pp. 219-241.

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

SFI-Dan Boneh