
# Information Security – Theory vs. Reality

0368-4474-01, Winter 2011

## **Lecture 5:** Process confinement

Eran Tromer

Slides credit: Dan Boneh, Stanford course CS155, 2010

# Running untrusted code

- We often need to run buggy/unstrusted code:
  - Programs from untrusted Internet sites:
    - toolbars, viewers, codecs for media players, “rich content”, “secure banking”
  - Old or insecure applications: ghostview, Outlook
  - Legacy daemons: sendmail, bind
  - Honeypots
- Goal: if application misbehaves, kill it

# Confinement

- **Confinement**: ensure application does not deviate from pre-approved behavior
- Can be implemented at many levels:
  - **Hardware**: isolated hardware ("air gap")
    - Difficult to manage
    - Sufficient?
  - **Virtual machines**: isolate OSs on single hardware
  - **System call interposition**:
    - Isolates a process in a single operating system
  - Isolating threads sharing same address space:
    - **Software Fault Isolation** (SFI), e.g., Google Native Code
  - Interpreters for non-native code
    - JavaScript, JVM, .NET CLR

# 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

# A simple example: chroot

- Often used for “guest” accounts on ftp sites
- To use do: (must be root)

```
# chroot  /home/guest
# su guest
```

root dir “/” is now “/home/guest”
EUID set to “guest”

- Now “/home/guest” is added to file system accesses for applications in jail

**open(“/etc/passwd”, “r”) ⇒**

**open(“/home/guest/etc/passwd”, “r”)**

⇒ application cannot access files outside of jail

# Jailkit

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

- **jailkit** project: auto builds files, libs, and dirs needed in jail environment
  - **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
- **Restricts only filesystem access. Unaffected:**
  - **Network access**
  - **Inter-process communication**
  - **Devices, users, … (see later)**

# 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

- 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 process inside jail
  - root is limited, e.g. cannot load kernel modules

# Problems with chroot and jail

- Coarse policies:
  - All-or-nothing access to file system
  - Inappropriate for apps like web browser
    - Needs read access to files outside jail
      (e.g. for sending attachments in gmail)

- Do not prevent malicious apps from:
  - Accessing network and messing with other machines
  - Trying to crash host OS

**System call interposition**:

a better approach to confinement

# System call interposition

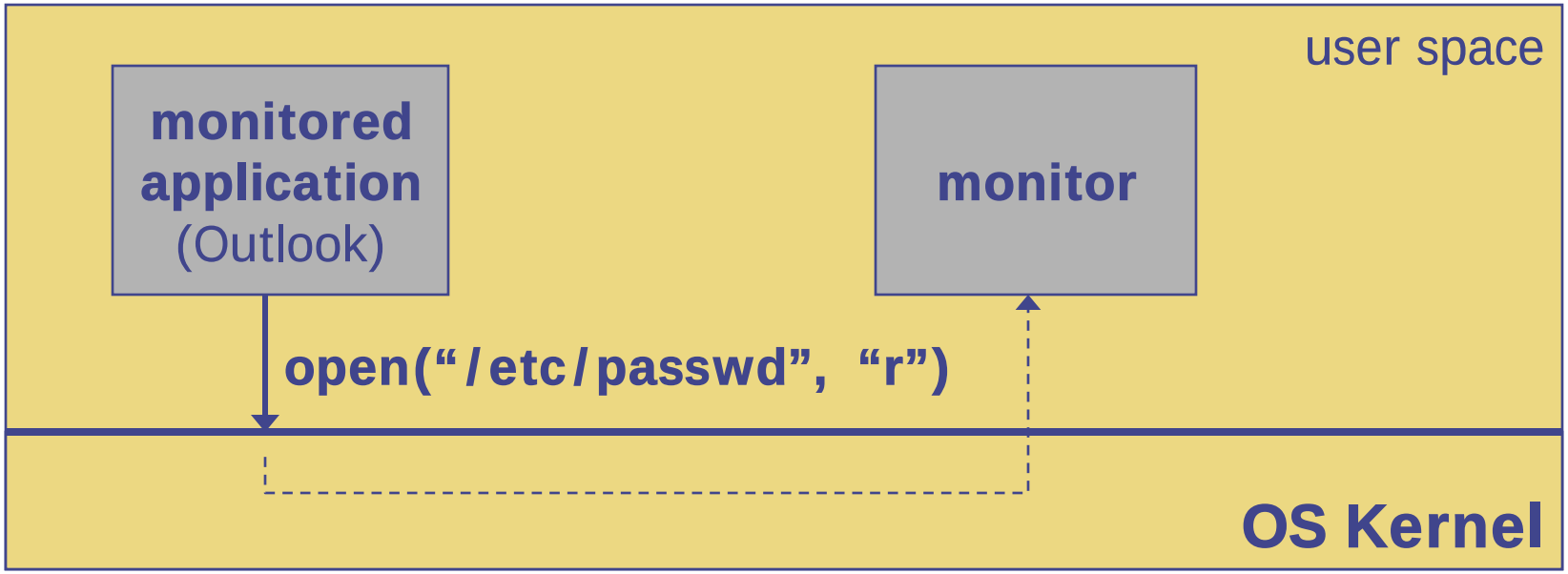
- Observation: to damage host system (i.e. make persistent changes) app must make system calls
  - To delete/overwrite files: unlink, open, write
  - To do network attacks: socket, bind, connect, send
- Monitor app system calls and block unauthorized calls
- Implementation options:
  - Completely kernel space (e.g. GSWTK)
  - Completely user space
    - ~~Capturing system calls via dynamic loader (LD_PRELOAD)~~
    - Dynamic binary rewriting (program shepherding)
  - Hybrid (e.g. Systrace)

# Initial implementation (Janus)

- Linux ptrace: process tracing

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

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




- Monitor kills application if request is disallowed

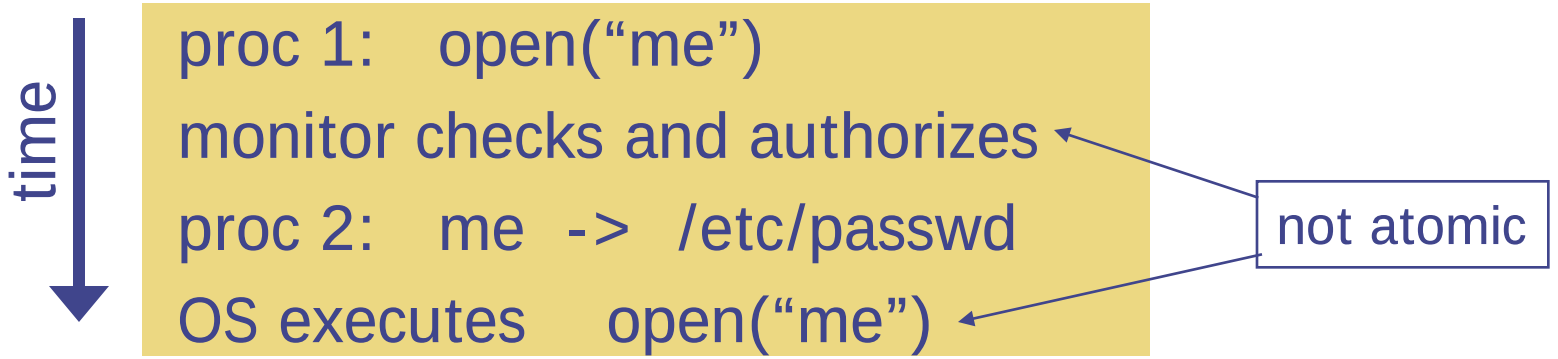
# Complications

- Monitor must maintain all OS state associated with app
  - current-working-dir (CWD), UID, EUID, GID
  - Whenever app does “cd path” monitor must also update its CWD
    - otherwise: relative path requests interpreted incorrectly
- If app forks, monitor must also fork
  - Forked monitor monitors forked app
- Monitor must stay alive as long as the program runs
- Unexpected/subtle OS features: file description passing, core dumps write to files, process-specific views (chroot, /proc/self)

# Problems with ptrace

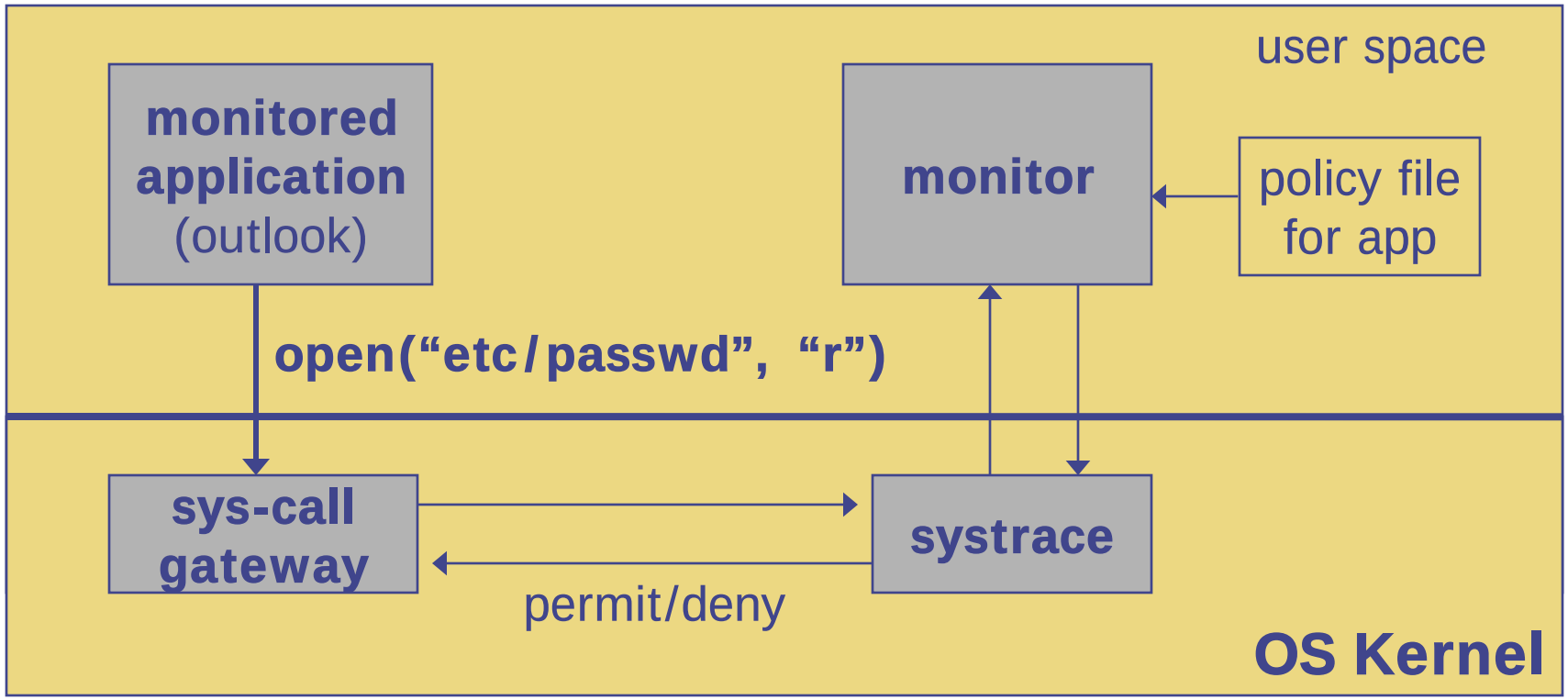
- ptrace is too coarse for this application
  - Trace all system calls or none
    - e.g. no need to trace “close” system call
  - Monitor cannot abort sys-call without killing app

- Security problems: **race conditions**
  - Example: symlink: me -> mydata.dat




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

# Alternate design: Systrace




- Systrace only forwards monitored sys-calls to monitor (saves context switches)
- Systrace resolves sym-links and replaces sys-call path arguments by full path to target
- When app calls execve, monitor loads new policy file
- Fast path in kernel for common/easy cases, ask userspace for complicated/rare cases

# Policy

- Sample policy file:

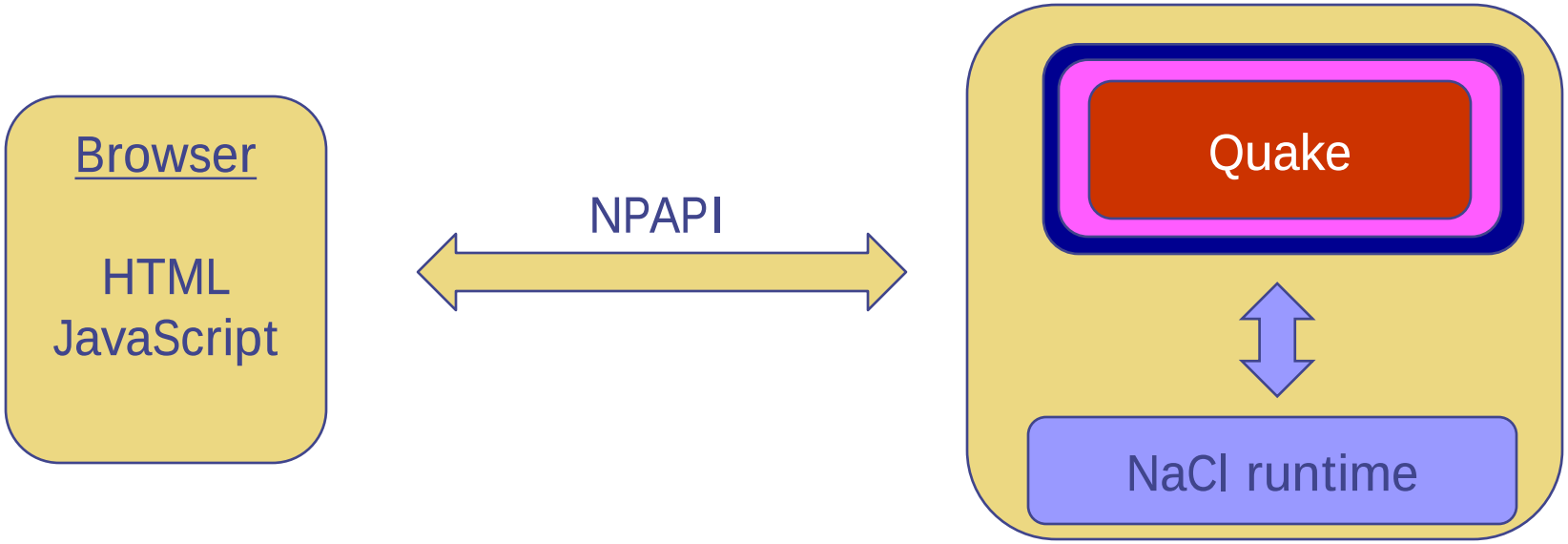
```
path allow   /tmp/*
path deny   /etc/passwd
network deny all
```

- Specifying policy for an app is quite difficult
  - Systrace can auto-gen 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 main reason this approach is not widely used

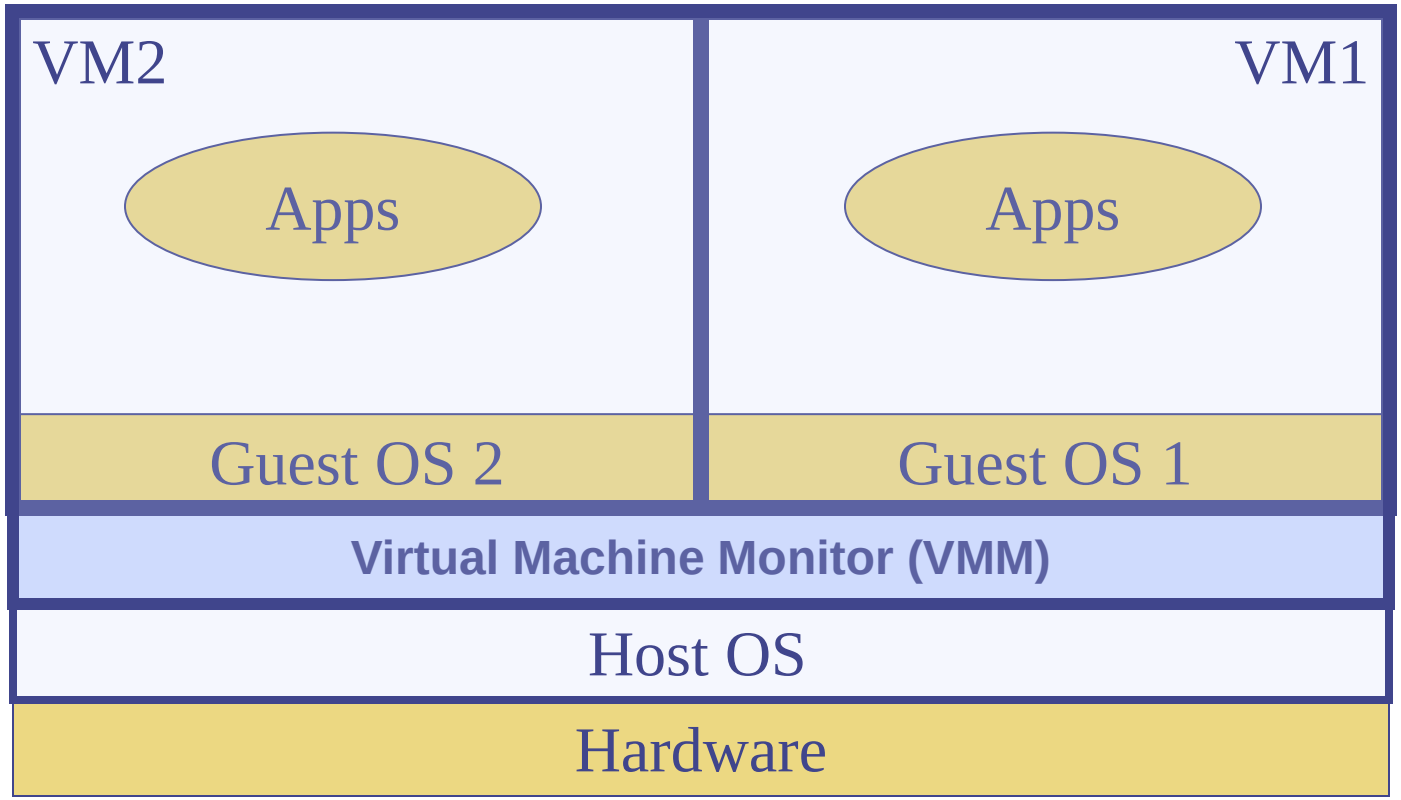
# NaCl: a modern-day example




- Quake: untrusted x86 code
- Two sandboxes:
  - Iuter sandbox: restricts capabilities using sys call interposition
  - Inner sandbox:
    - Uses x86 memory segmentation to isolate application memory from one another
    - Restricts allowed machine code to protect the segmentation

# Confinement using Virtual Machines

# Virtual Machines




Example: **NSA NetTop, NSA patent**

- 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 server, ...
  - Wasteful to run each service on a different computer
    - VMs save hardware while isolating services
  - More generally: VMs heavily used in cloud computing

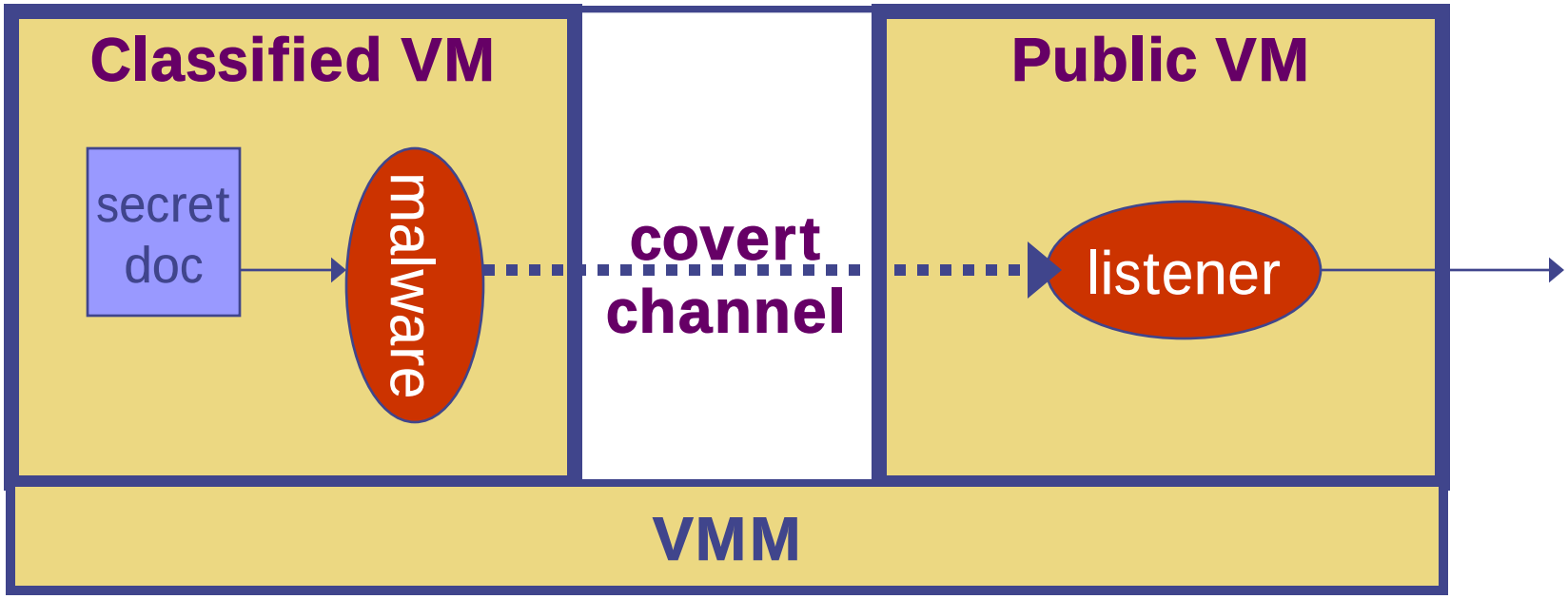
# VMM security assumption

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

- Requires that VMM protect itself and is not buggy
  - VMM is much simpler than full OS
  - ... but device drivers run in Host OS

# Problem: covert channels

- Covert channel: unintended communication channel between isolated components
  - Can be used to leak classified data from secure component to public component



# An example covert channel

- Both VMs use the same underlying hardware

- To send a bit $b \in \{0,1\}$ malware does:
  - b= 1: at 1:30.00am do CPU intensive calculation
  - b= 0: at 1:30.00am do nothing

- At 1:30.00am listener does a CPU intensive calculation and measures completion time
  - Now $b = 1 \Leftrightarrow$ completion-time > threshold

- Many covert channel exist in running system:
  - File lock status, cache contents, interrupts, ...
  - Very difficult to eliminate
- Recall cache attacks from Lecture 1

# VMM Introspection

protecting the anti-virus system

# Example: 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 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

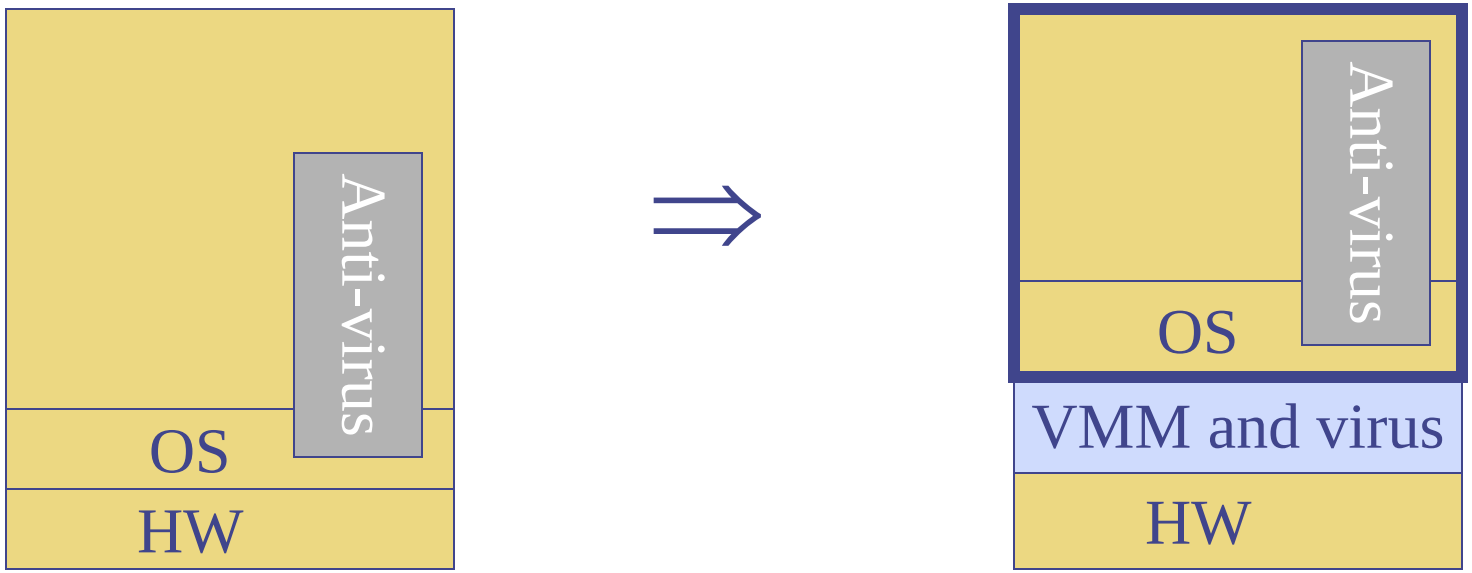
4. Virus signature detector
   - Run virus signature detector on GuestOS memory

5. Detect if GuestOS puts NIC in promiscuous mode

# Subvirt:
## subverting VMM confinement

# Subvirt

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



# The MATRIX


The Red Pill
The Blue Pill

# VM Based Malware (blue pill virus)

- VMBR: **a virus that installs a malicious VMM (hypervisor)**

- **Microsoft Security Bulletin: (Oct, 2006)**
  http://www.microsoft.com/whdc/system/platform/virtual/CPUVirtExt.mspx

  - 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)

1. VM platforms often emulate simple hardware
   - VMWare emulates an ancient i440bx chipset

     ... but report 8GB RAM, dual Opteron CPUs, etc.
2. VMM introduces time latency variances
   - Memory cache behavior differs in presence of VMM
   - Results in relative latency in time variations for any two operations
3. VMM shares the TLB with GuestOS
   - GuestOS can detect reduced TLB size
4. Deduplication (VMM saves single copies of identical pags)

... 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**

# Summary

- Many sandboxing techniques:
  - Physical air gap,
  - Virtual air gap (VMMs),
  - System call interposition
  - Software Fault isolation (e.g., NaCl)
  - Application specific (e.g. Javascript in browser)
- Often complete isolation is inappropriate
  - Apps need to communicate through regulated interfaces
- Hardest aspect of sandboxing:
  - Specifying policy:  what can apps do and not do