Lecture 17 – Isolation

Ryan Cunningham
University of Illinois
ECE 422/CS 461 – Fall 2017

Announcement

- Midterm:
  - Monday, Oct. 16^{th} 7-9pm
  - ECEB 1002 (here)
- Conflict
  - Friday Oct. 13^{th} 4-6pm
  - Siebel Center 4405
  - MUST have an email from you

Security News

- WSJ reports Russian hackers stole NSA tools from a contractor, tipped off by Kaspersky

The confinement principle

Credit to Dan Boneh and Stanford's CS155

Running untrusted code

We often 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

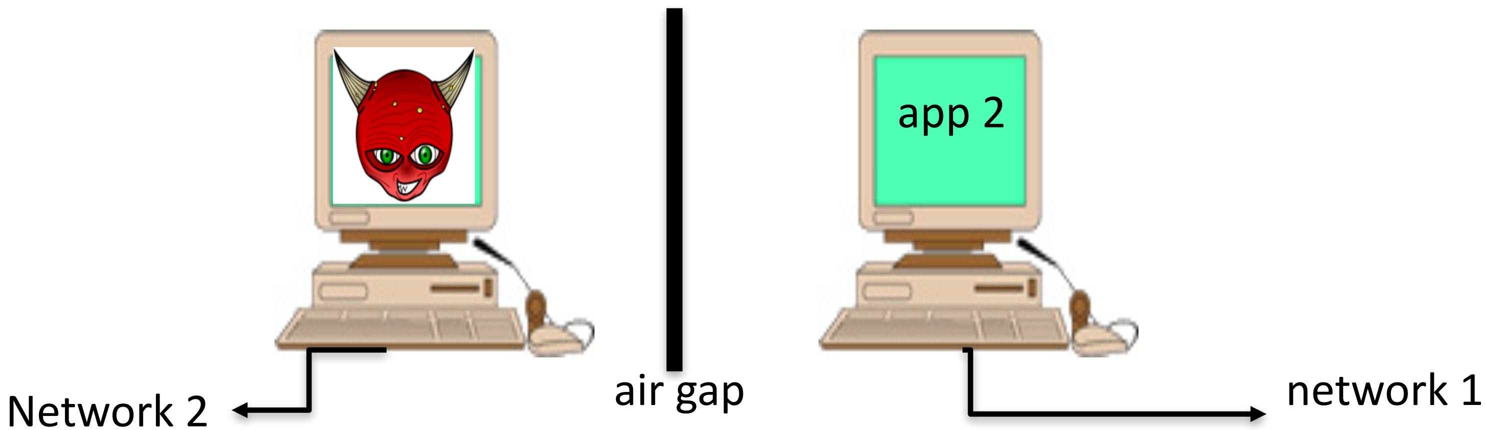
<u>Goal</u>: if application "misbehaves" $\Rightarrow$ kill it

Approach: confinement

<u>Confinement</u>: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

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



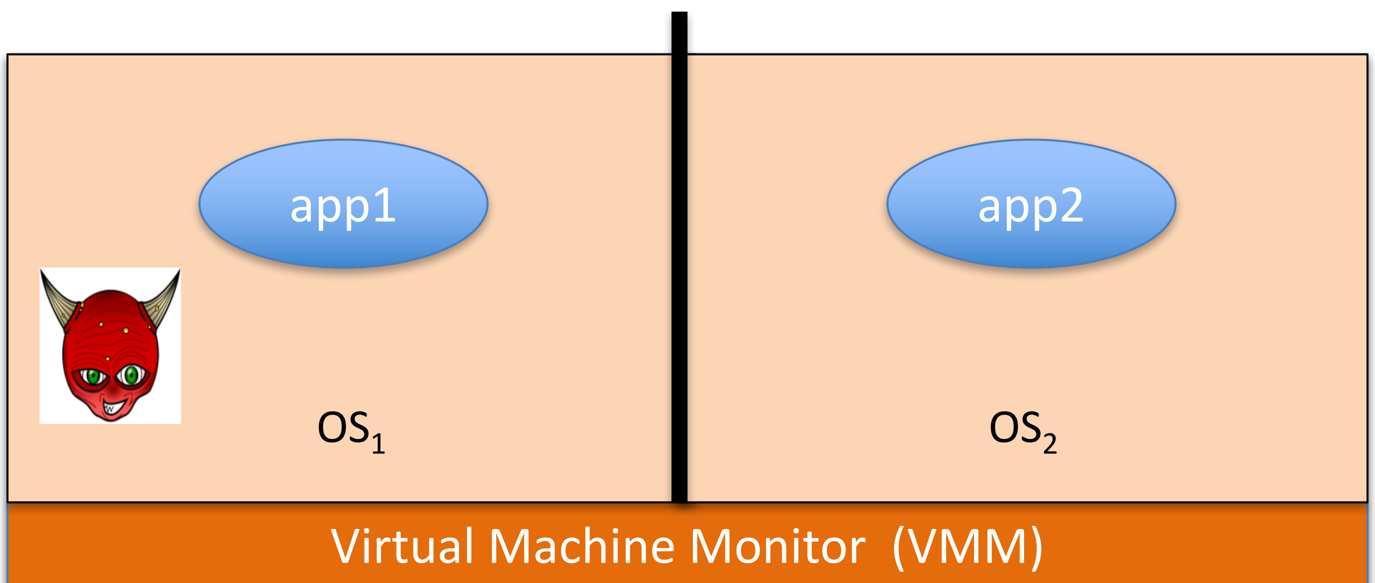
$\Rightarrow$ difficult to manage

Approach: confinement

<u>Confinement</u>: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

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

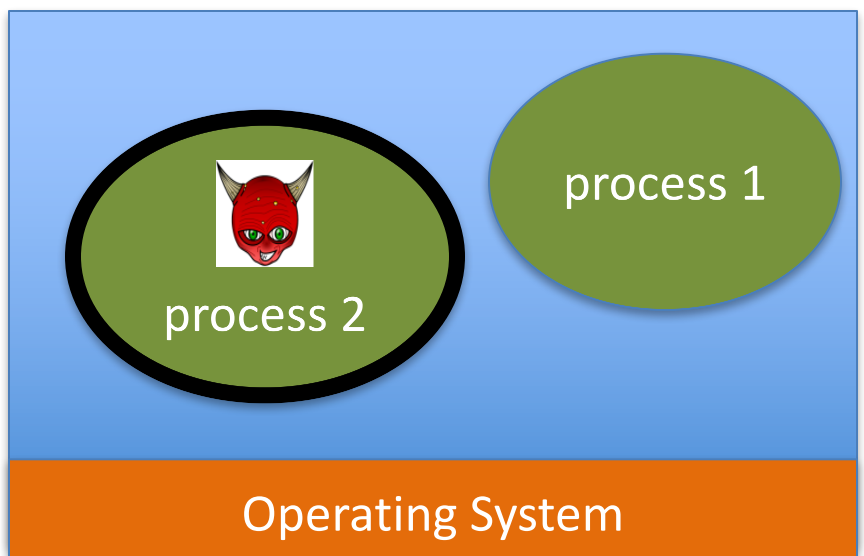


Approach: confinement

<u>Confinement</u>: 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



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 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") $\Rightarrow$

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)

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

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

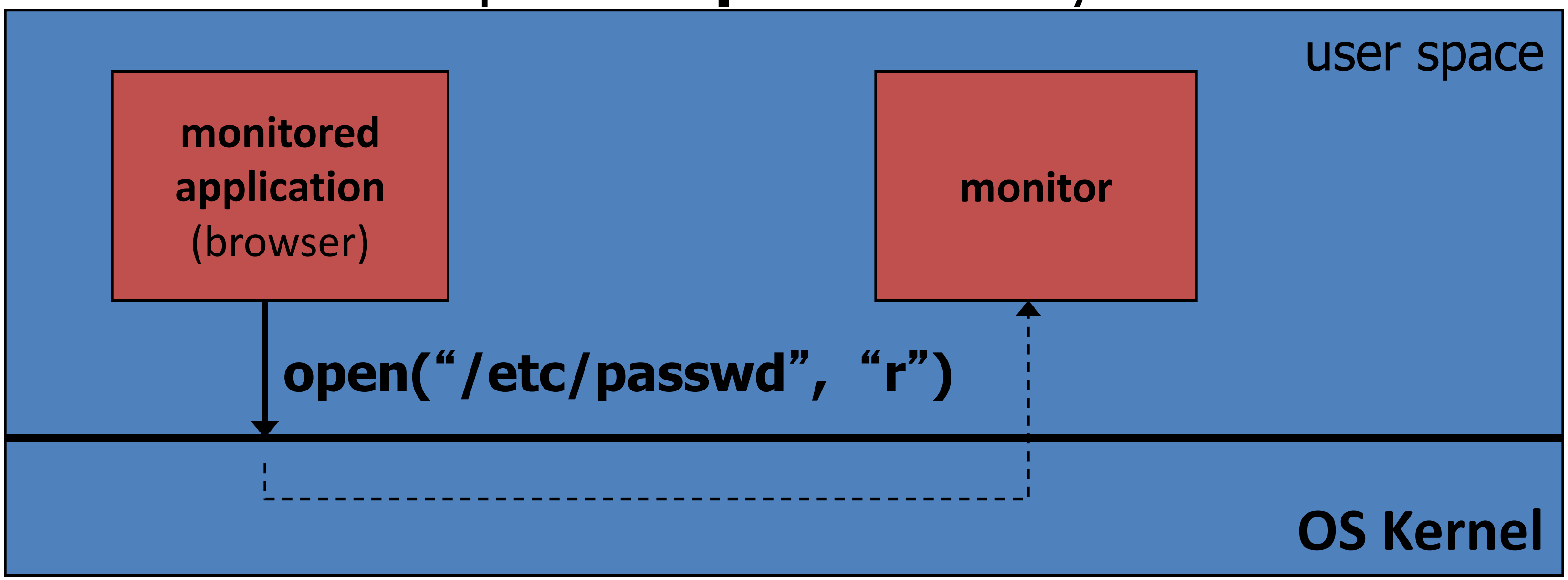
Initial implementation (Janus)

[GWTB'96]

Linux **ptrace**: process tracing

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

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



Monitor kills application if request is disallowed

Complications

```
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**

- Example: symlink: me ⟶ mydata.dat

time

proc 1: open("me")

monitor checks and authorizes

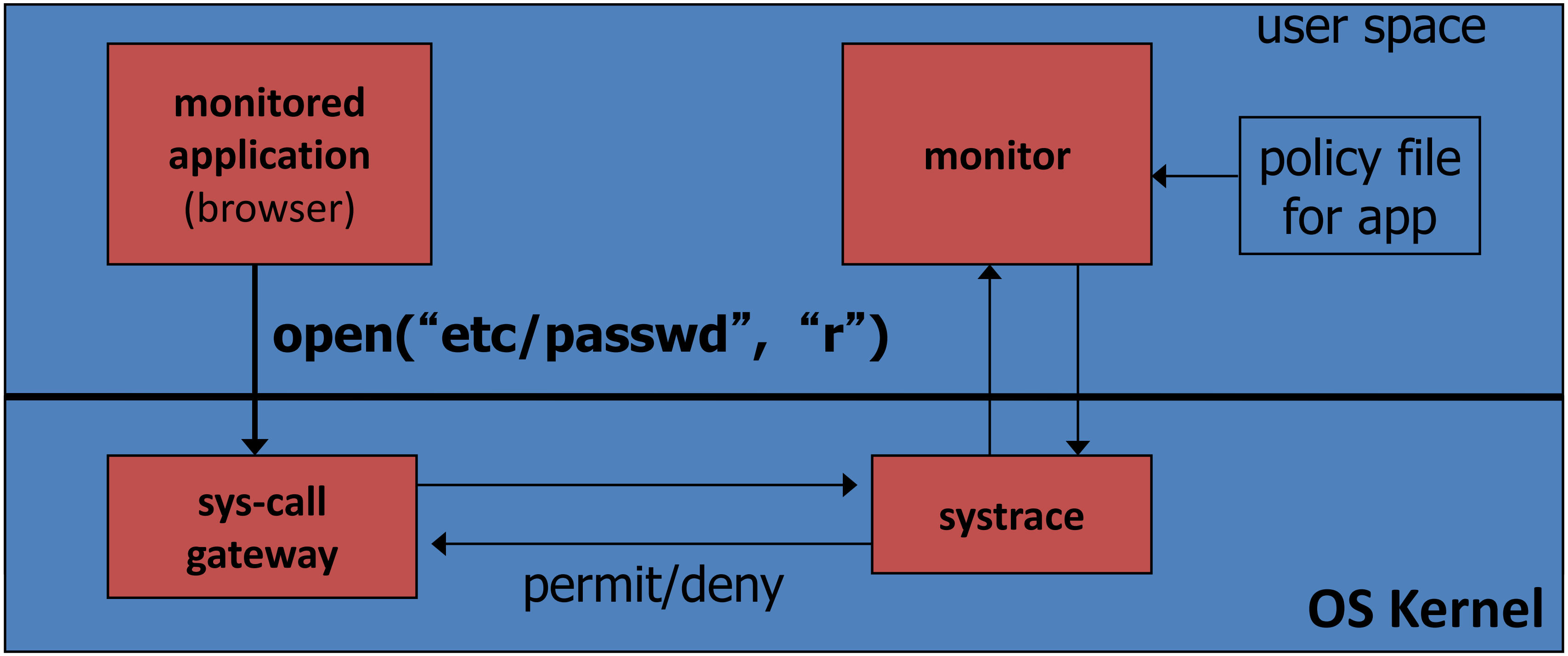
proc 2: me ⟶ /etc/passwd

OS executes open("me")

not atomic

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

Policy

Sample policy file:

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

Manually specifying policy for an app can be difficult:

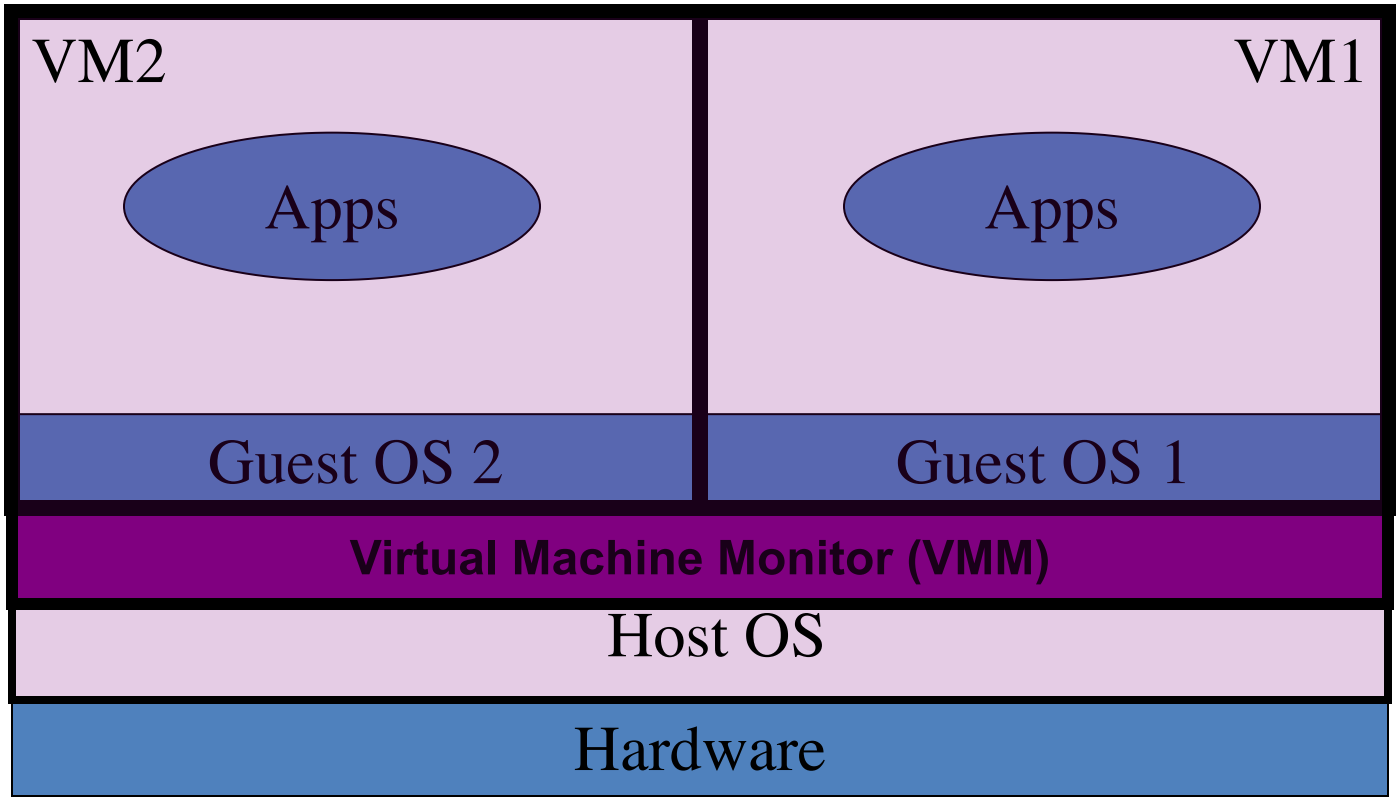
- Systrace can auto-generate policy by learning how app behaves on "good" inputs
- If policy does not cover a specific sys-call, ask user

  … 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

Isolation via
Virtual Machines

Virtual Machines



Example: **NSA NetTop**

single HW platform used for both classified and unclassified data

Why so popular now?

VMs in the 1960's:

- Few computers, lots of users
- VMs allow many users to shares a single computer

VMs 1970's – 2000: non-existent

VMs since 2000:

- Too many computers, too few users
  - Print server, Mail server, Web server, File server, Database , …
- Wasteful to run each service on different hardware
- More generally: VMs heavily used in cloud computing

VMM security assumption

VMM Security assumption:

– Malware can infect guest OS and guest apps

– But malware cannot escape from the infected VM

- Cannot infect host OS
- Cannot infect other VMs on the same hardware

Requires that VMM protect itself and is not buggy

– VMM is much simpler than full OS

… but device drivers run in Host OS

Intrusion Detection / Anti-virus

Runs as part of OS kernel and user space process

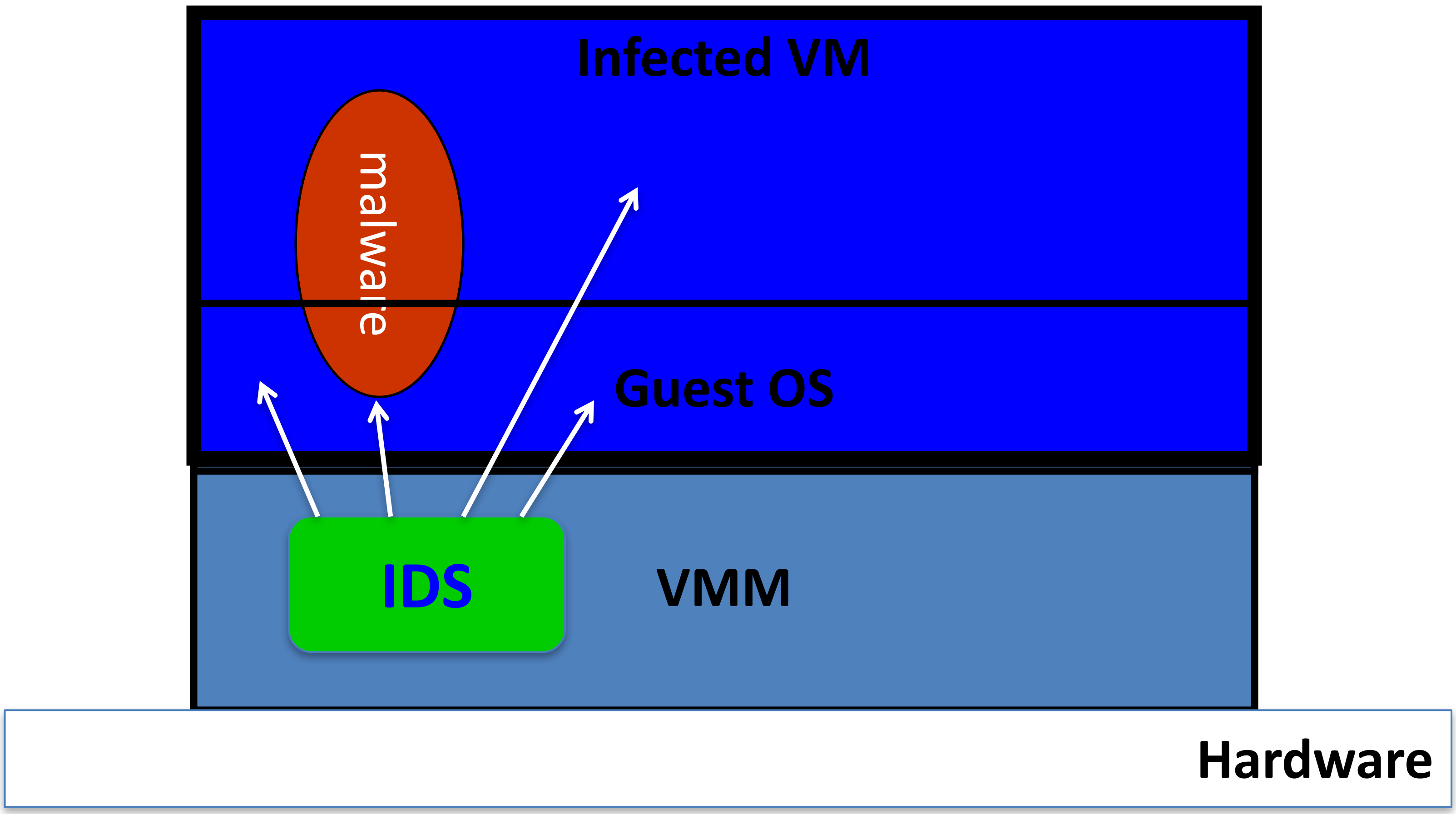
- Kernel root kit can shutdown protection system
- Common practice for modern malware

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

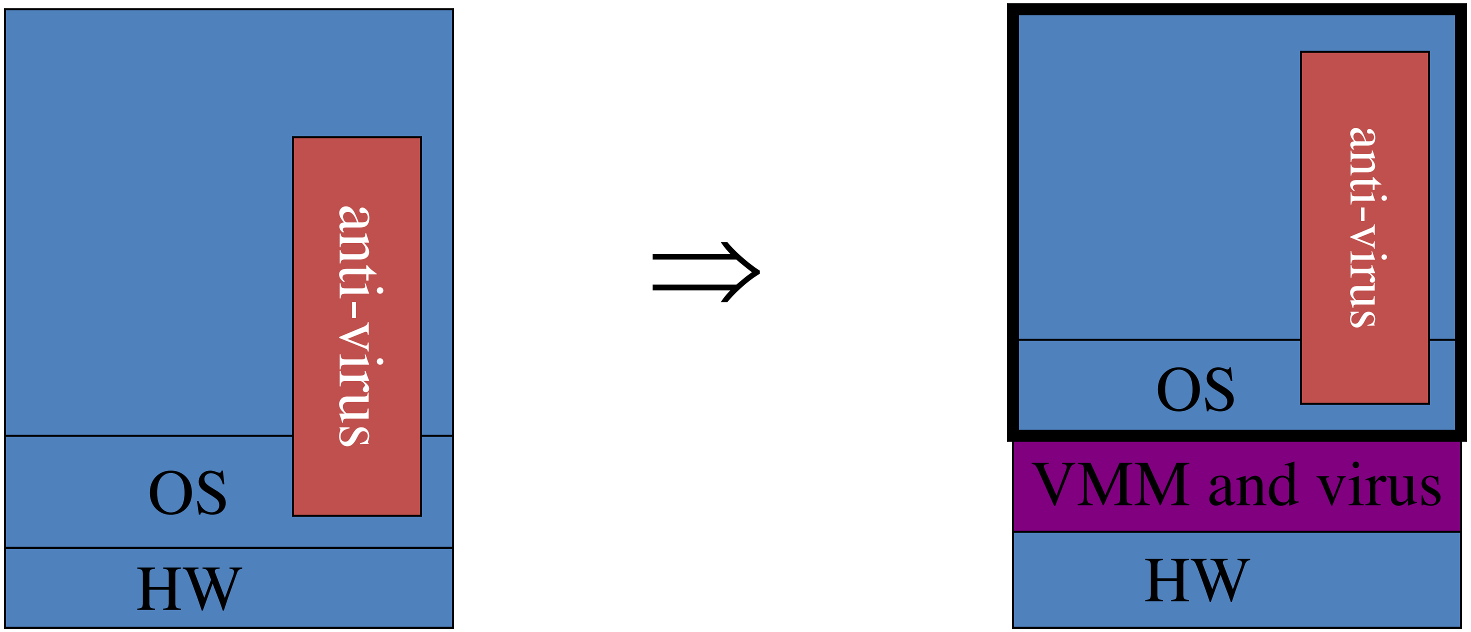
Sample checks

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

Problem: 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



Problem: covert channels

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