ACCESS CONTROL
ISOLATION AND CONFINEMENT
VIRTUALIZATION

VITALY SHMATIKOV

Cannot Avoid Running Untrusted Code

Programs from untrusted sources

- Mobile apps, JavaScript…

Applications are exposed to untrusted content

- Browsers, PDF viewers, email agents…

Honeypots

Confinement

Goal: ensure that misbehaving application cannot harm the rest of the system

Defense in Depth

Any piece of code can be buggy or compromised

Systems need multiple layers of protection

Example: What if there's a vulnerability in Chrome's JavaScript interpreter?

- Chrome should prevent malicious website from accessing other tabs
- OS should prevent access to other processes (e.g., password manager)
- Hardware should prevent permanent malware installation in device firmware
- Network should prevent malware from infecting nearby computers

Confinement at Multiple Levels

Hardware

- Run application on isolated machine (air gap)

Virtual machines

- Multiple OSes on the same machine, isolated from each other

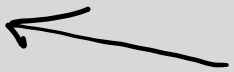
Process containers

- Isolate a process in an OS via system call interposition

Threads

- Isolate threads sharing address space via software fault isolation

Application sandboxes



Example: browser sandbox for JavaScript

Reference Monitor

Observes execution of the program/process and mediates its requests

- At what level? Instructions, memory accesses, system calls, network packets…

Enforces confinement

- Halts or confines execution if the program is about to violate the security policy ?

Cannot be circumvented by the monitored process

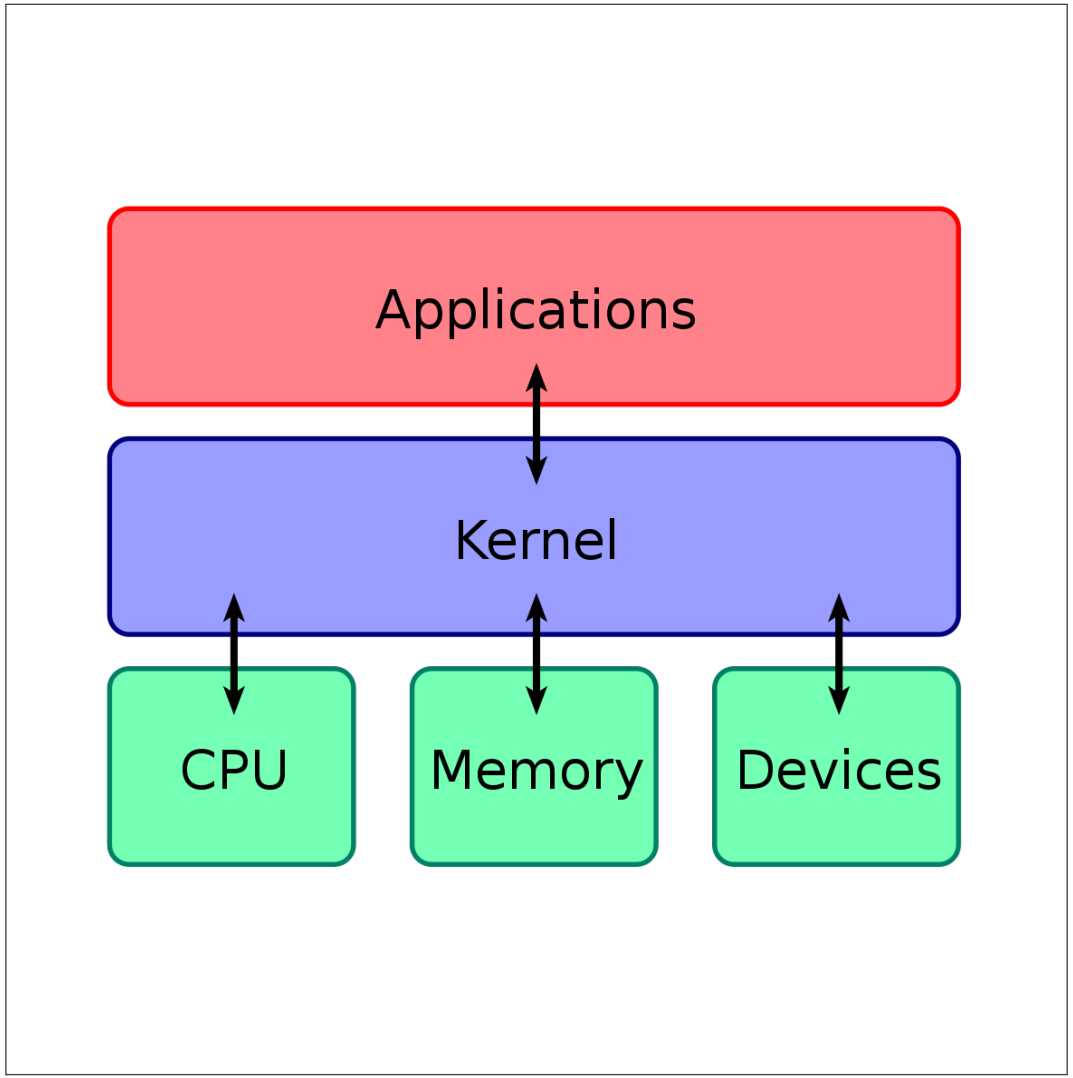
Principle of Least Privilege

Users and programs should only have access to the data and resources needed to perform routine, authorized tasks

"Faculty can only change grades for classes they teach"
"Only employees with background checks have access to classified documents

This requires privilege separation: dividing system into components, each with limited access

Compartmentalization is key!



Operating System Basics

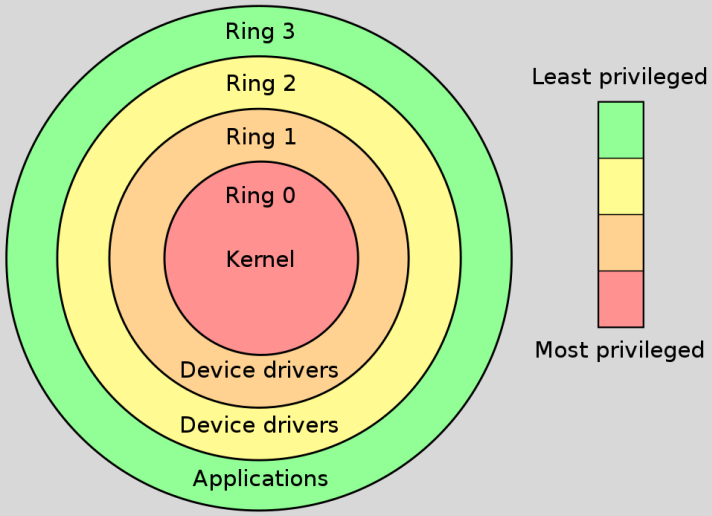
- Multi-tasking, multi-user OS are now the norm
- **Kernel** mediates between applications and resources
- **Applications** consist of one or more processes
- **Processes** have executable program, allocated memory, resource descriptors (e.g., file descriptors), processor state

Protection Rings

Different parts of system must operate at different privilege levels

Protection rings included in all typical CPUs today and used by most operating systems

- Lower number = higher privilege
- Ring 0 is supervisor
- Inherit privileges over higher levels



Intel x86 protection rings

Principle of least privilege:
User account or process should have least privilege level required to perform their intended functions

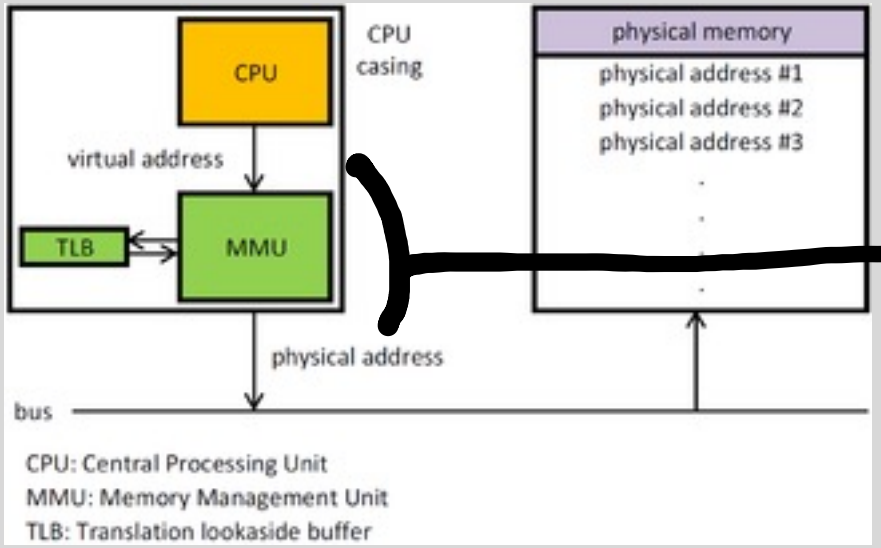
Security Policies

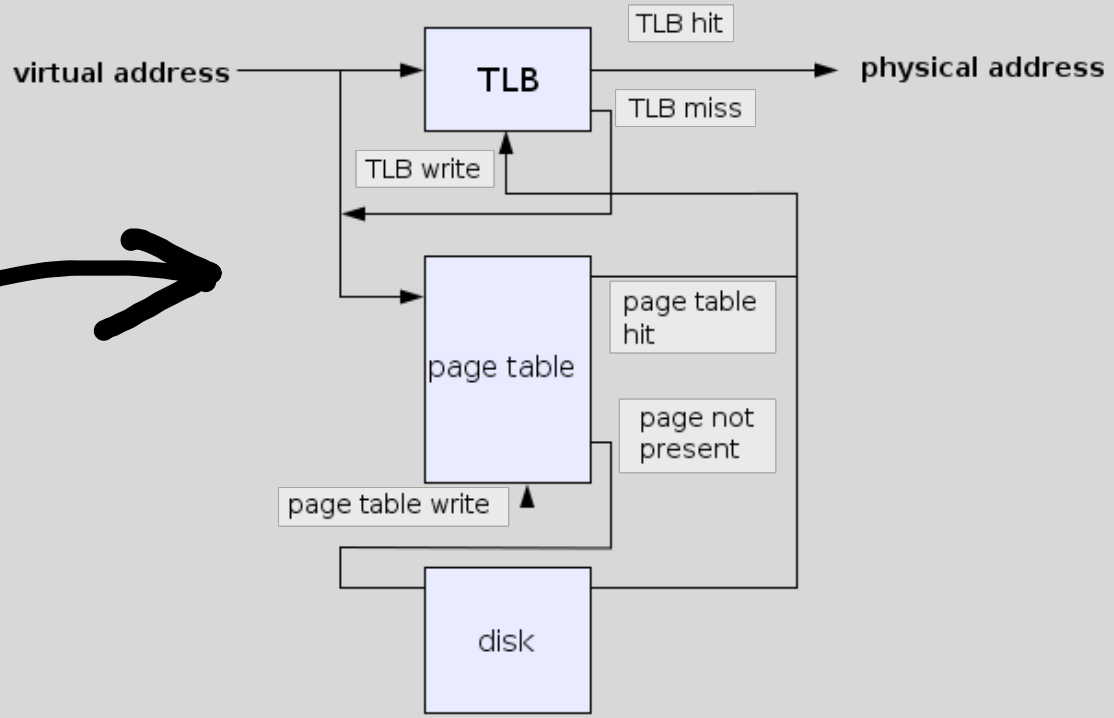
Principle of least privilege and privilege separation apply to any subject performing an operation on a protected object

Examples of security policies

- UNIX: A user should only be able to read their own files
- UNIX: A process should not be able to read another process's memory
- Mobile: An app should not able to edit other apps' data
- Web: A domain should only be able to read its own cookies

Memory Management





What prevents a process from reading another process's memory?

ACLs vs. Capabilities

Examples of capability-based systems we have seen in this course?

Capabilities: subject presents an unforgeable ticket that grants access to an object. System doesn't care who subject is, just that they have access.

ACL: system checks where subject is on list of users with access to the object.

UNIX

Developed at Bell Labs in the early 1970s

Simple, elegant, very influential design

Modern descendants: MacOS and iOS, Linux, FreeBSD, Android…

Ken Thompson and Dennis Ritchie

Recipients of 1983 Turing Award

UNIX Security Model

	Unix
Subjects (Who)	Users, Processes
Objects (What)	Memory, Files, Hardware devices …
Operations	Read, write, execute

UNIX Users

Service accounts used to run background processes (e.g., web server)

User accounts

- Typically tied to a specific human
- Every user has a unique integer ID (UID)

Many system operations can only run as root

Users and Superusers

A user has username, group name, password

Root is an administrator / superuser (UID 0)

- Can read and write any file or system resource (network, etc.)
- Can modify the operating system
- Can become any other user
  - Execute commands under any other user’s ID
- Can the superuser read passwords?

Access Control in UNIX

Everything is a file

- Files and also sockets, pipes, hardware devices….
- Files are laid out in a tree

inode data structure records OS management information about the file

- UID and GID of the file owner
- Type, size, location on disk
- Time of last access (atime), last inode modification (ctime), last file contents modification (mtime)
- Discretionary ACL via permission bits

Users can set some controls (as opposed to mandatory access control, set in a central place)

UNIX Permission Bits

File type
- regular file
d directory
b block file
c character file
l symbolic link
p pipe
s socket

Permission bits
r read
w write
x execute (if directory, traverse it)
s setuid, setgid (if directory, files have gid of dir owner)
t sticky bit (if directory, append-only)

Each file has 12 ACL bits:
rwx for each of owner, group, all; set user ID; set group ID; sticky bit

UNIX Process Permissions

Process (normally) runs with the permissions of the user who invoked process

Suppose user wants to change password…

- Need to modify /etc/shadow password file
- /etc/shadow is owned by root
- Can user's process modify /etc/shadow?
- How does passwd program change user's password?

Process IDs in UNIX

Each process has three UIDs (similar for GIDs)

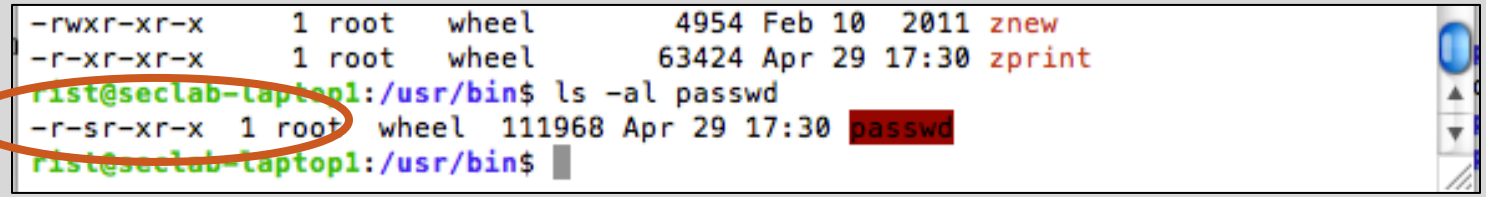
- **Real ID:** user who started the process
- **Effective ID:** determines effective access rights of the process
- **Saved ID:** used to swap IDs, gaining or losing privileges

Known as setuid programs

If an executable's setuid bit is set, it will run with the effective privileges of its owner, not the user who started it

- Example: when I run lpr to access a printer, real UID is shmat (13630), effective UID is root (0), saved UID is shmat (13630)

Setuid Programs



- setuid bit – execute with privileges of file's owner
- setgid bit – execute with privileges of file's group

- So passwd is a **setuid program**

runs at permission level of owner, not user who invoked it

Least privilege at process granularity:
passwd runs as root to access /etc/shadow
Can we do better?

Privilege Escalation

Privilege escalation: bug that allows lower-privilege user to perform actions as a higher-privilege user (typically, root)

99% of local vulnerabilities in UNIX systems exploit setuid-root programs to obtain root privileges

- The other 1% target the OS itself

Acquiring and Dropping Privilege

- To acquire privilege, assign privileged UID to effective ID
- To drop privilege temporarily, remove privileged UID from effective ID and store it in saved ID
  - Can restore it later from saved ID
- To drop privilege permanently, remove privileged UID from both effective and saved ID

Example:

- Apache Web Server must start as **root** because only root can create a socket that listens on port 80 (a privileged port)
- Without privilege reduction, any Apache bug would give attacker root access to server
- Instead, Apache creates children like this:

```
if (fork() == 0) {
    int sock = socket(":80");
    setuid(getuid("www-data"));
}
```

Setuid management is tricky and error-prone!

Checking Access Rights

User should only be able to access a file if he has the permission to do so

But what if the user is running as setuid-root?

- For example, the printing program usually runs with root privileges so it can access the printer… but root can read any file! How does the printing program know that the user who invoked it has the right to read (and print) a given file?

UNIX has a special access() system call

Race Condition

```
if( access("/tmp/myfile", R_OK) != 0 ) {
        exit(-1);
}
file = open( "/tmp/myfile", "r" );
read( file, buf, 100 );
close( file );
print( "%s\n", buf );
```

access() checks RUID,
but open() only checks EUID

This is known as a TOCTTOU attack
("Time of Check To Time of Use")

Changes the file to which the filename points

ln –s /etc/shadow /tmp/myfile

Prints out shadow file
(including password hashes)

concurrent execution

This is an example of a concurrency vulnerability

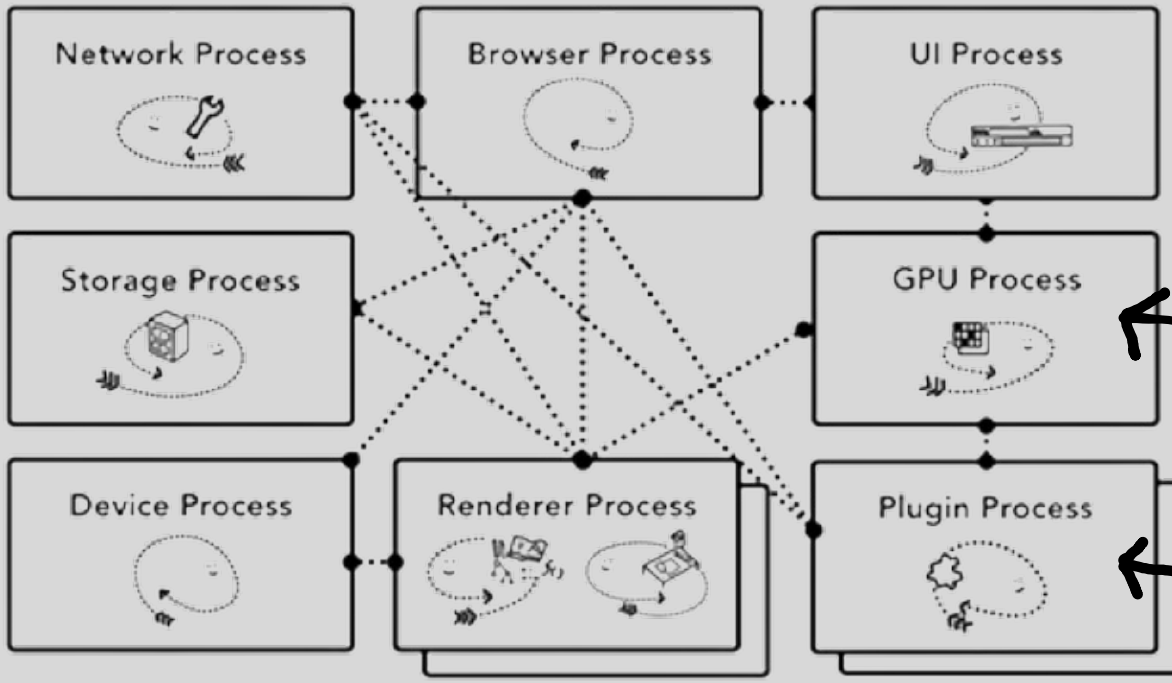
Summary of UNIX Access Control

- Simple model provides protection for most situations
- Flexible enough to make most simple systems possible in practice

- Coarse-grained ACLs don't account for enterprise complexity
- ACLs don't handle different applications within a single user account
- Nearly all system operations require root access — bugs give attacker full access

Chrome Architecture

Controls "Chrome" part of the application (address bar, bookmarks, etc.), handles invisible, privileged parts of the browser (e.g., network requests)

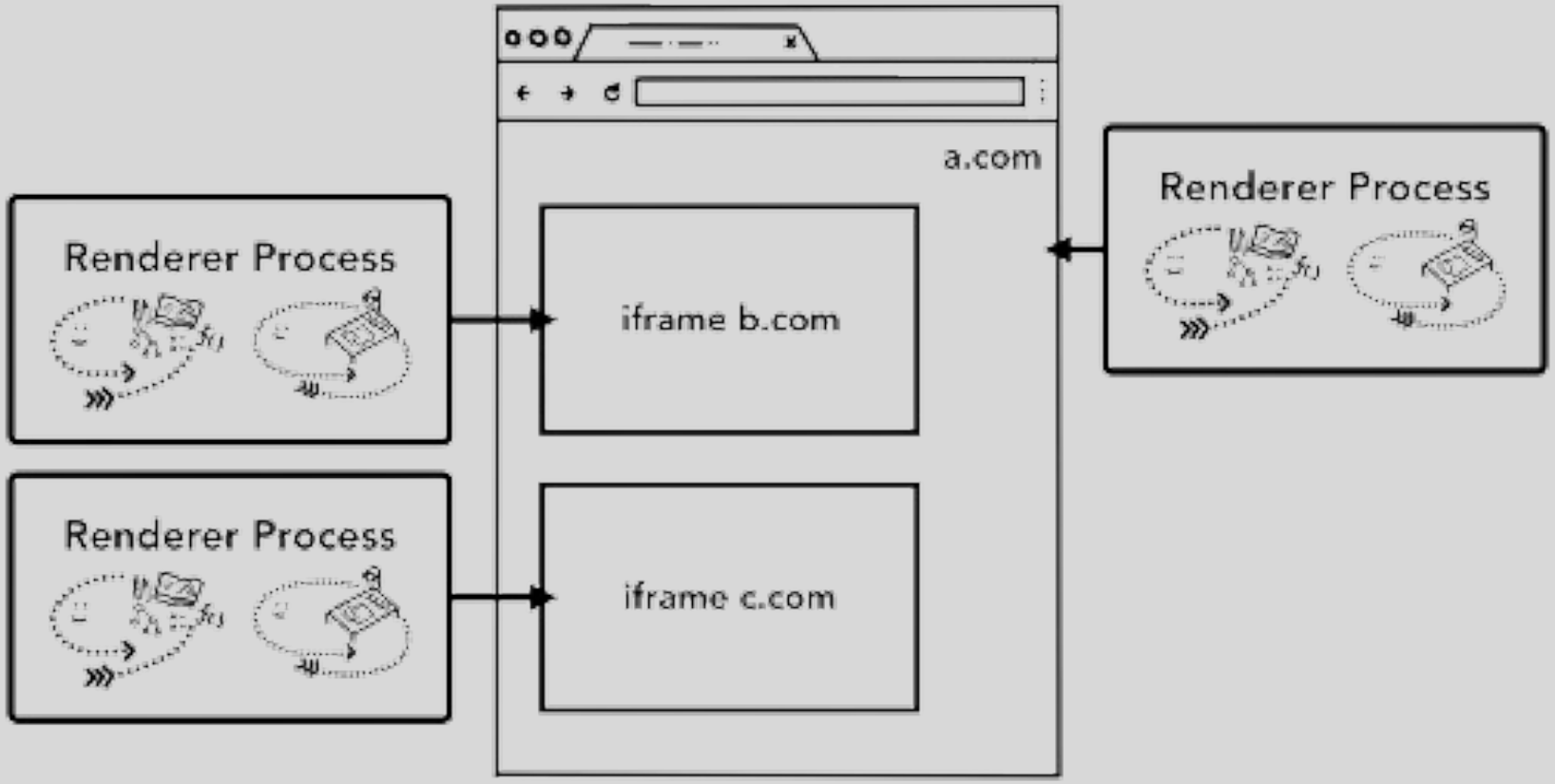


Handles GPU tasks in isolation from other processes. It is isolated because GPU handles requests from multiple apps and draws them on the same surface.

Controls any plugins used by website

Controls everything inside the tab where a webpage is displayed

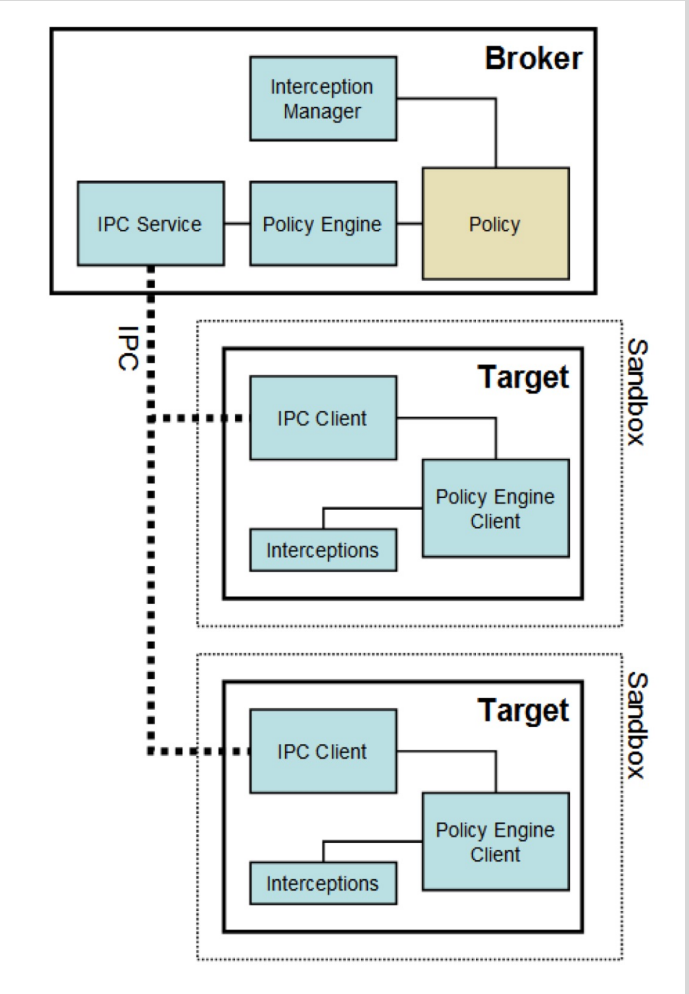
Process-Based Website Isolation



Chrome Sandboxing

Broker (main browser) controls/supervises activities of the sandboxed processes

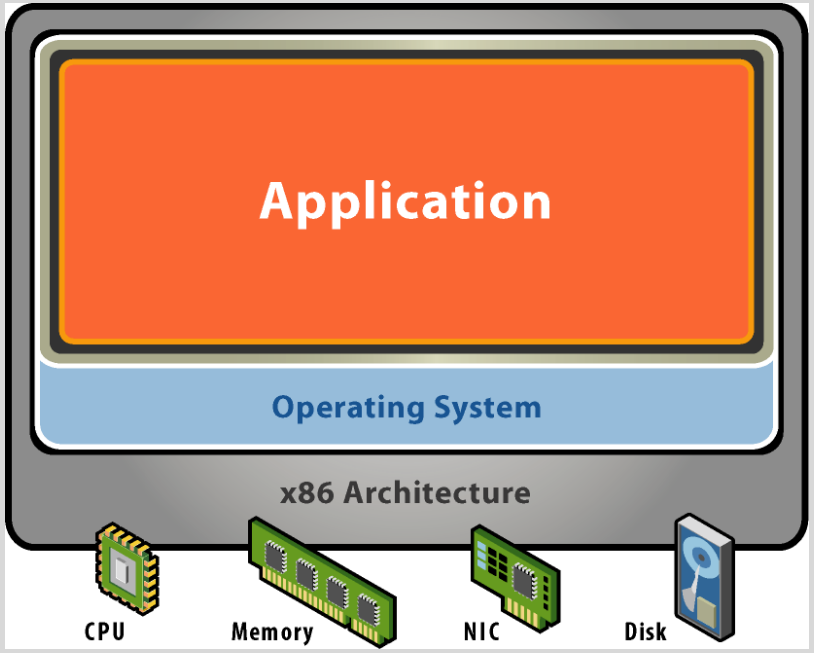
Renderer's only access to the network is via its parent browser process; file system access is restricted



Restricted Security Context

- Chrome calls **CreateRestrictedToken** to create a token that has a subset of the user's privileges
- Assigns the token the user and group **S-1-0-0 Nobody**, removes access to nearly every system resource
- As long as the disk root directories have non-null security, no files (even with null ACLs) can be accessed
- No network access

Physical Machine



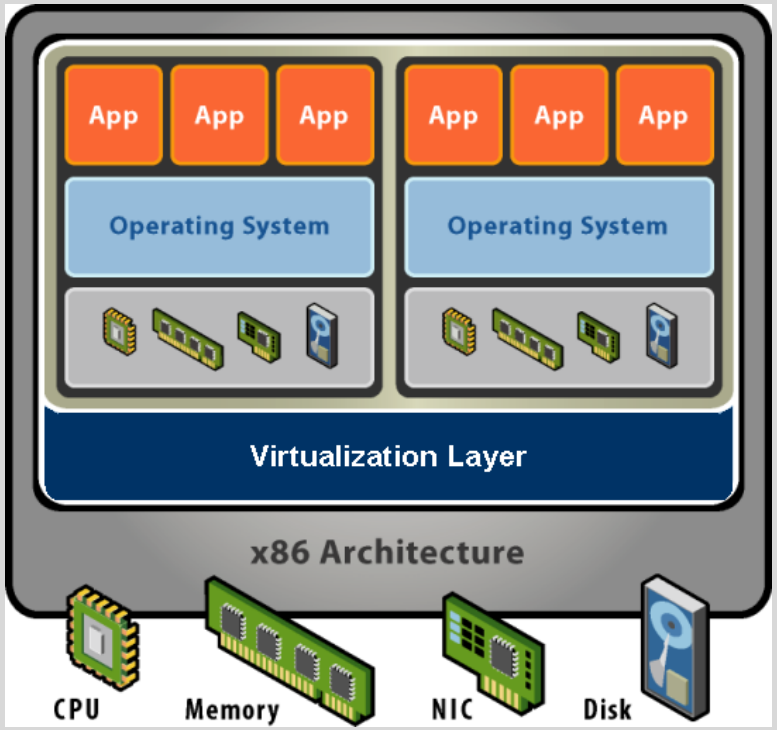
Physical hardware

- Processors, memory, chipset, I/O devices, etc.
- Resources often grossly underutilized

Software

- Tightly coupled to physical hardware
- Single active OS instance
- OS controls hardware

Virtual Machine



Software abstraction

- Behaves like hardware
- Encapsulates all OS and application state

Virtualization layer

- Extra level of indirection
- Decouples hardware, OS
- Enforces isolation
- Multiplexes physical hardware across VMs

Virtualization Properties

Isolation of faults and performance

Encapsulation of entire VM state

- Enables snapshots and cloning of VMs

Portability

- Independent of physical hardware
- Enables migration of live, running VMs

Interposition

- Transformations on instructions, memory, I/O
- Enables transparent resource overcommitment, encryption, compression, replication …

Virtualization Use Cases

- Legacy support
- Development
- Server consolidation
- Sandboxing / containment
- Cloud computing infrastructure-as-a-Service

Studying Malware with VMs

Researchers use VMs to study malware

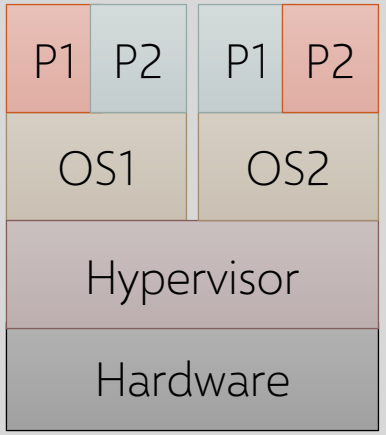
- Example: crawl Web, load pages in a browser running in a VM, look for pages that damage VM

Example of VM sandboxing

- Hypervisor must confine malicious code

How would you evade analysis as a malware writer?

- Split personalities: web page can detect it is running in a VM by using timing variations in writing to screen…malware in web page becomes benign when in a VM, evades detection



HYPERVISOR DETECTION

aka “red pill” techniques

Red Pill Techniques

- VM platforms often emulate simple hardware
  - VMWare emulates an ancient i440bx chipset… but report 8GB RAM, dual CPUs, etc.
- Hypervisor introduces time latency variances
  - Memory cache behavior differs in presence of hypervisor
  - Results in relative time variations for any two operations
- Hypervisor shares the TLB with Guest OS
  - Guest OS can detect reduced TLB size
- Many more methods

Garfinkel et al. "Compatibility is Not Transparency: VMM Detection Myths and Reality"

Hypervisor 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 hypervisor protect itself and is not buggy

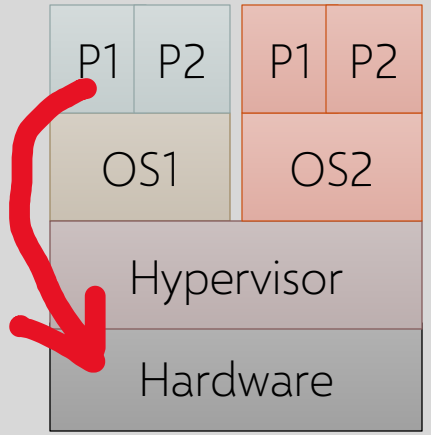
… (some) hypervisors are much simpler than a full OS

Violating Confinement

Escape-from-VM

- Vulnerability in VMM or host OS (e.g., Dom0)

Memory management flaws in VMM



Zero-Day Exploit Published for VM Escape Flaw in VirtualBox

by Lucian Constantin on November 8, 2018

A security researcher disclosed a yet unpatched zero-day vulnerability in the popular VirtualBox virtualization software that can be exploited from a guest operating system to break out of the virtual machine and gain access to the host OS.

Violating Isolation

Covert channels between VMs circumvent access controls

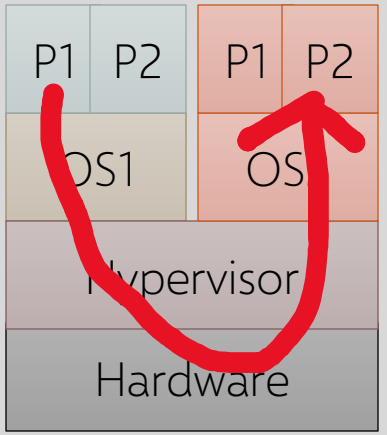
- Bugs in VMM
- Side-effects of resource usage

Degradation-of-service attacks

- Guests might maliciously contend for resources
- Xen scheduler vulnerability

Side channels

Spy on other guest via shared resources



Much more about this later