

Isolation & Authorization

University of Illinois

ECE 422/CS 461

Goals

- By the end of this chapter you should:
 - Identify Lampson’s “basic dilemma” and “gold standard” of security
 - Define Isolation, its properties, and why is isolation important
 - Understand at what levels isolation can be implemented
 - Understand Authorization and how different access control mechanisms work

Isolation

- *“The foundation of security, unquestionably, is isolation. If you don’t have isolation there’s no hope for security.”* – Butler Lampson, SOSPP’15
- Isolation: The host prevents one execution environment from affecting another.
- A related concept is Confinement: The host prevents one execution environment from affecting the rest of the system.
- Caveat: Must trust the host!

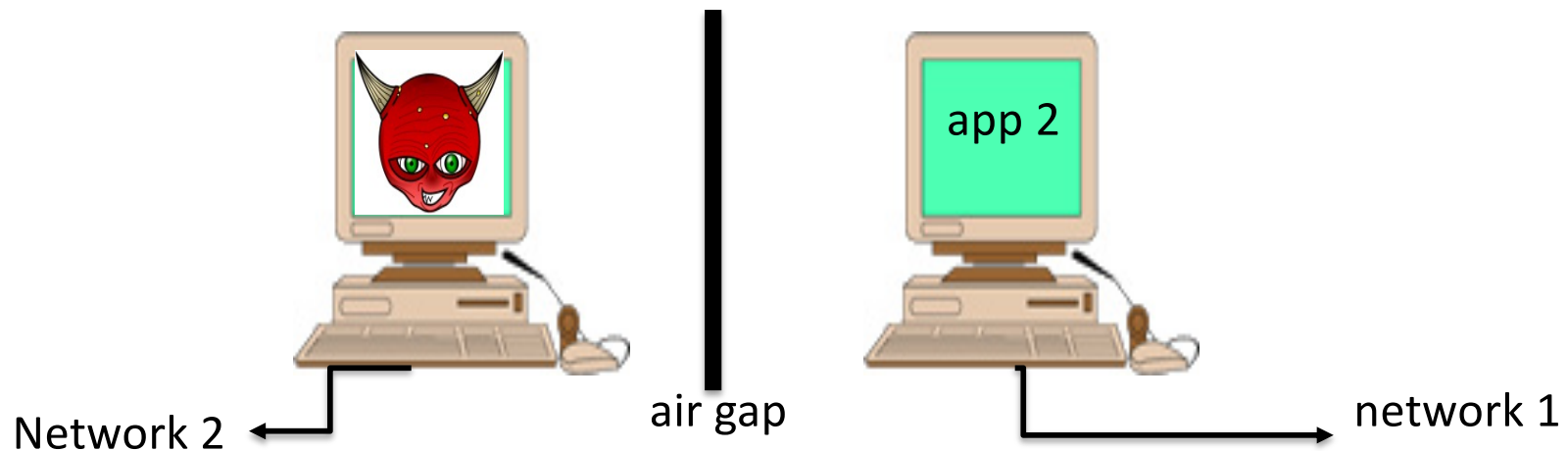
Isolation Mechanisms

- Isolation can be implemented in many ways (and is associated with many bugs)

Mechanism	Host	Relevant Citation/Year
Air Gaps	Physics	[Tempest ~1955]
Processes	OS	[CTSS 1962]
Virtual Machines	Hypervisor	[CP/40 1966]
Modules/Objects	Language/Runtime	[Clu 1974]
Limited comm. (wires or crypto)	Network	[DESNIC 1985]
Software Fault Isolation	Process	[Wahbe et al., 1993]
Java (JS) Sandboxing	JVM/JS Engine	[Java 1995]

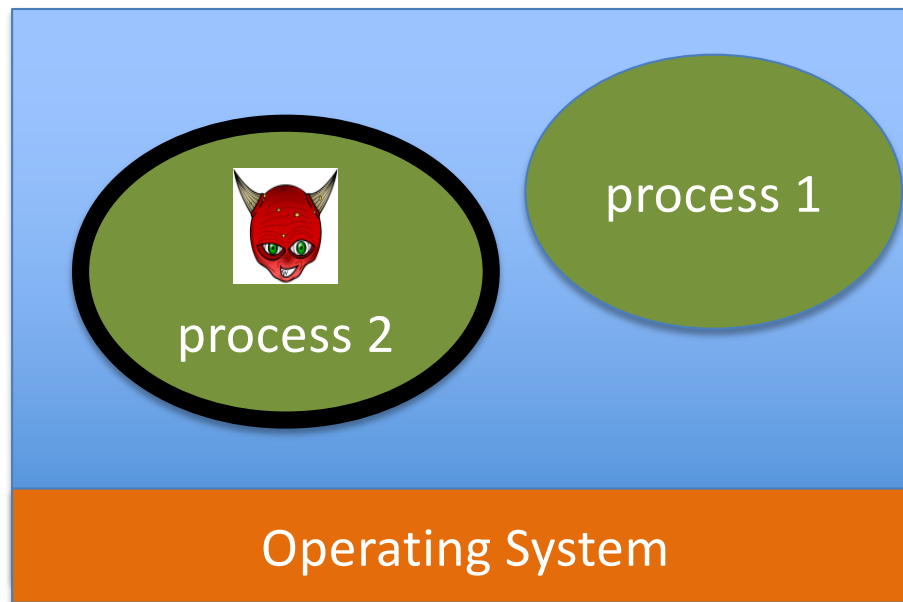
Isolation via Physics

- Air gap the hosts
- Still relevant today, particular in security-critical environments (see, e.g., Stuxnet 2005)



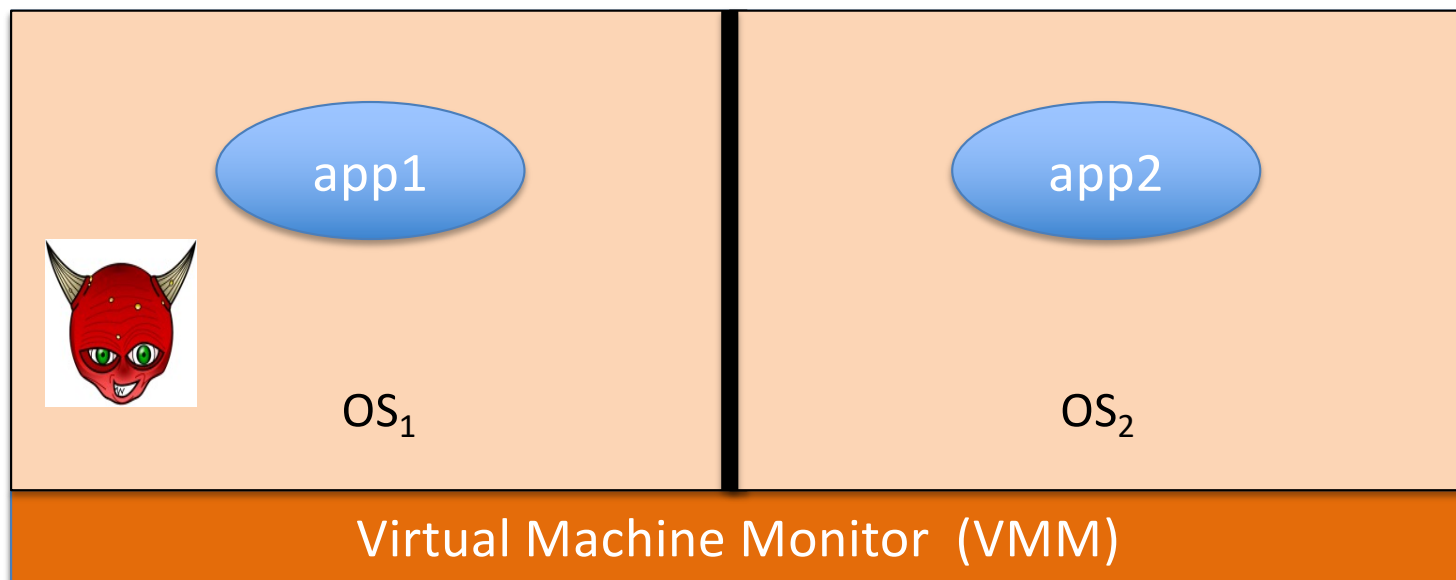
Isolation via OS

- Process spaces are all about isolation.
- Facilitates via system call interposition, virtual memory, protection rings, etc.



Isolation via Hypervisor

- Hypervisor (a.k.a. Virtual Machine Monitor) hosts multiple operating systems on a single machine.
- An old idea, essential to cloud computing today (e.g., Xen, KVM, VMWare, etc.)
- Compare to process isolation?



Why are VMs so popular now?

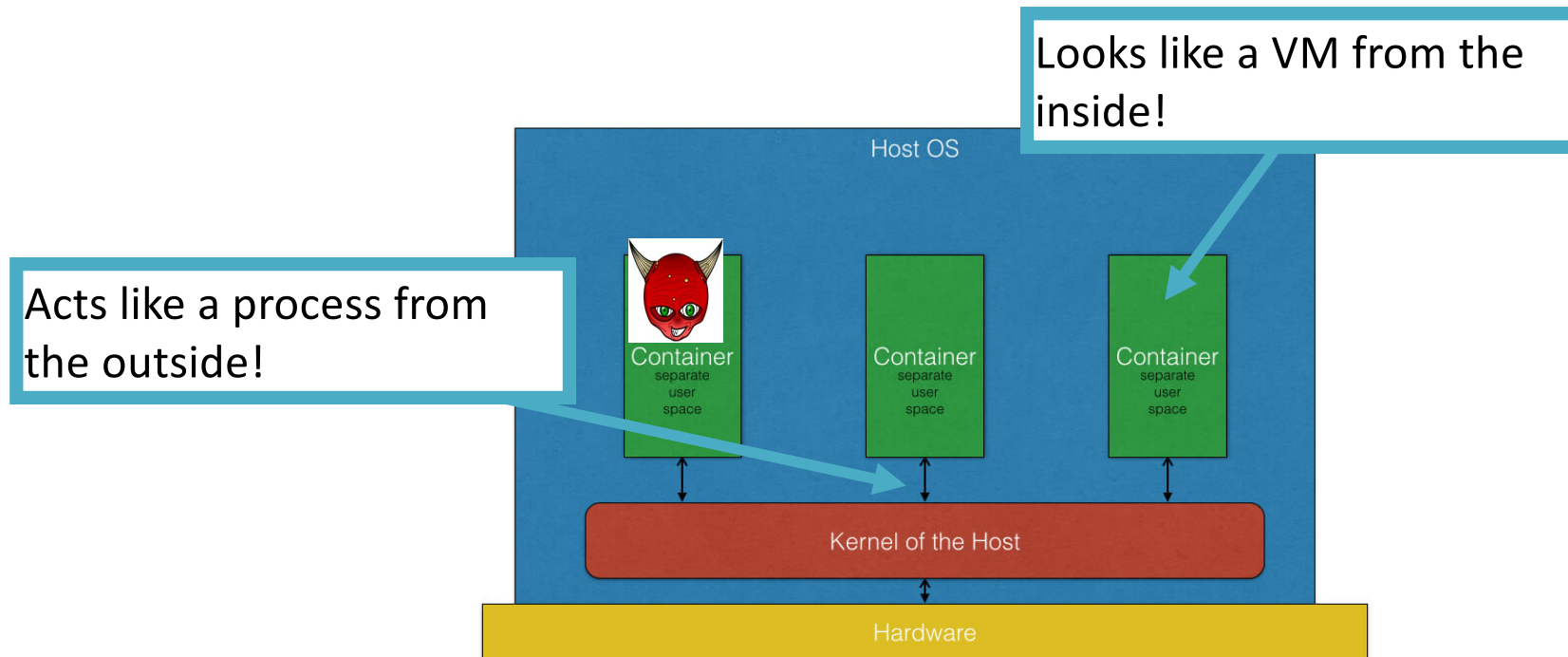
- VMs in the 1960' s:
 - Few computers, lots of users
 - VMs allow many users to share a single computer
- VMs 1970' s – 2000: non-existent
- VMs since 2000:
 - Too many computers, too few users
 - Print server, Mail server, Web server, File server, Database, ...
 - Wasteful to run each service on different 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

Isolation via Containers

- A hybrid of Process- and VM-based isolation
- Kernel virtualizes an execution environment within a separate *namespace* of the OS.
- Compare to hypervisor/VM-based isolation?



An old idea: chroot


- Often used for “guest” accounts on ftp sites
- To use: (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

An old idea: chroot

- 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
- *Sound familiar?* 
- note: simple chroot jail does not limit network access

Threats to Isolation

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

Container Security?

“Containers do not contain.” - Dan Walsh (SELinux contributor)

- In a nutshell, it's real hard to prove that every feature of the operating system is namespaced.
 - /sys? /proc? /dev? LKMs? kernel keyrings?
- Root access to any of these enables pwning the host
- Solution? Layer on access controls. SELinux and other systems now support for namespace labeling.
 - Easier to express a correct security policy over namespaces than, say, individual processes.

Threats to Isolation

- There will always be counterexamples, but...
we're actually reasonably good at isolation!
- So what's the problem?
 - **Isolation vs. Sharing**
 - “The basic dilemma of security.” – Lampson
 - Every user wants a private machine isolated from others, but they also want to share data, programs, and resources!!
 - Especially true today (e.g., cloud)

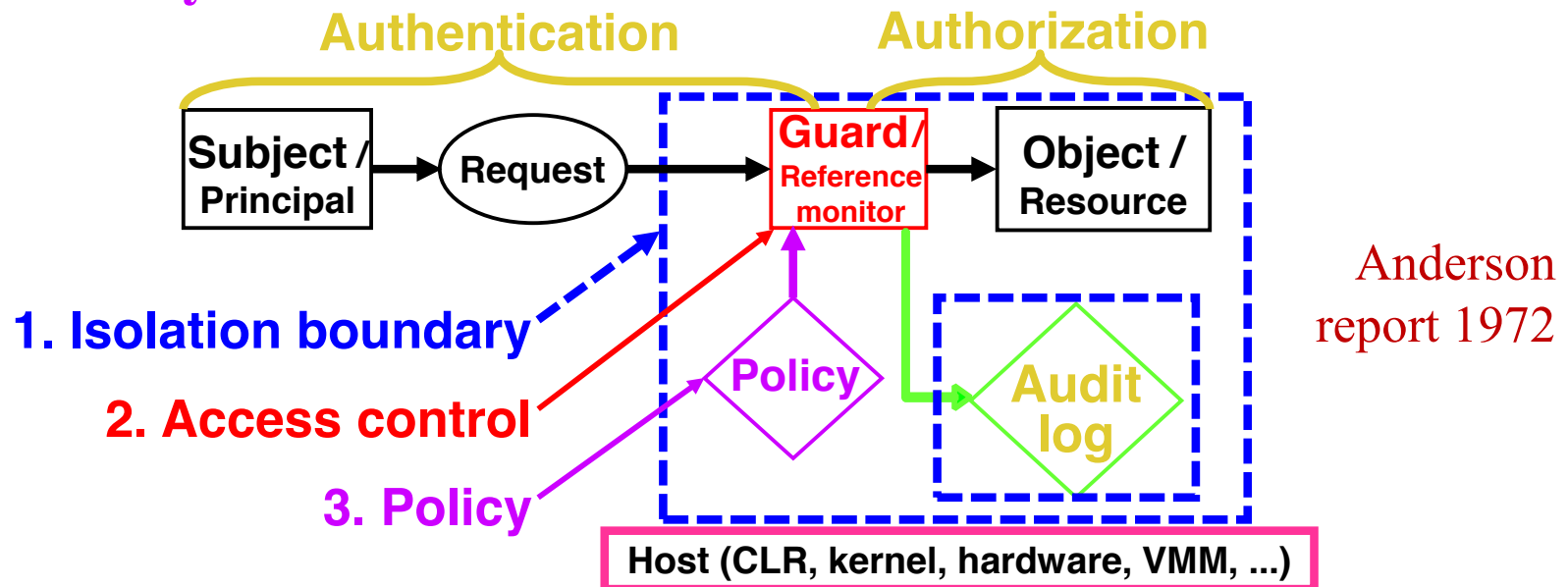
Access Control

The **Gold** Standard...

- **Authenticate** principles: Who made a request?
 - People, but also channels/servers/programs
- **Authorize** access: Who is trusted with a resource?
 - *Group* principles and resources to simplify management
- **Audit** requests: Who did what when?

Access Control

1. **Isolation boundary** limits attacks to channels (no bugs)
2. **Access Control** for channel traffic
3. **Policy** sets the rules




DAC vs MAC

- I. Discretionary access control: “owners” of an object define its policy.
 - Users have discretion over who has access to what objects and when (i.e., users are trusted)
 - Canonical example: the UNIX filesystem (RWX assigned by file owners)
 2. Mandatory access control: the environment enforces a static policy
 - Access control policy is defined by the administrators and the system environment, user has no control over rights
 - Canonical example: Process labeling (system assigns labels for process and objects)
- **Advantages/Disadvantages?**

Discretionary Access Control

Access Mask defines permissions for User, Group, and Other



```
batesa@sp19-cs423-adm:/dev$ ls -alh /dev
total 0
drwxr-xr-x 19 root root      4.0K Mar  6 06:48 .
drwxr-xr-x 23 root root      321 Mar  6 06:48 ..
crw-rw-rw- 1 root root    10, 175 Feb 10 14:31 agpgart
crw-rw-rw- 1 root root    10, 235 Feb 10 14:31 autofs
drwxr-xr-x 2 root root      380 Feb 10 14:31 block
drwxr-xr-x 2 root root       80 Feb 10 14:31 bsg
crw-rw-rw- 1 root disk    10, 234 Feb 10 14:31 btrfs-control
lrwxrwxrwx 1 root root         3 Feb 10 14:31 cdrom -> sr0
lrwxrwxrwx 1 root root         3 Feb 10 14:31 cdrw -> sr0
drwxr-xr-x 2 root root     3.3K Mar  6 06:48 char
crw-rw-rw- 1 root root       5,   1 Feb 10 14:31 console
lrwxrwxrwx 1 root root        11 Feb 10 14:31 core -> /proc/kcore
drwxr-xr-x 6 root root      120 Mar  6 06:48 cpu
crw-rw-rw- 1 root root    10,  59 Feb 10 14:31 cpu_dma_latency
crw-rw-rw- 1 root root    10, 203 Feb 10 14:31 cuse
drwxr-xr-x 6 root root      120 Feb 10 14:31 disk
brw-rw-rw- 1 root disk   253,   0 Feb 10 14:31 dm-0
brw-rw-rw- 1 root disk   253,   1 Feb 10 14:31 dm-1
brw-rw-rw- 1 root disk   253,   2 Feb 10 14:31 dm-2
brw-rw-rw- 1 root disk   253,   3 Feb 10 14:31 dm-3
drwxr-xr-x 2 root root       80 Feb 10 14:31 dri
lrwxrwxrwx 1 root root         3 Feb 10 14:31 dvd -> sr0
crw-rw-rw- 1 root root    10,  61 Feb 10 14:31 ecryptfs
crw-rw-rw- 1 root video   29,   0 Feb 10 14:31 fb0
lrwxrwxrwx 1 root root        13 Feb 10 14:31 fd -> /proc/self/fd
brw-rw-rw- 1 root disk     2,   0 Feb 10 14:31 fd0
```

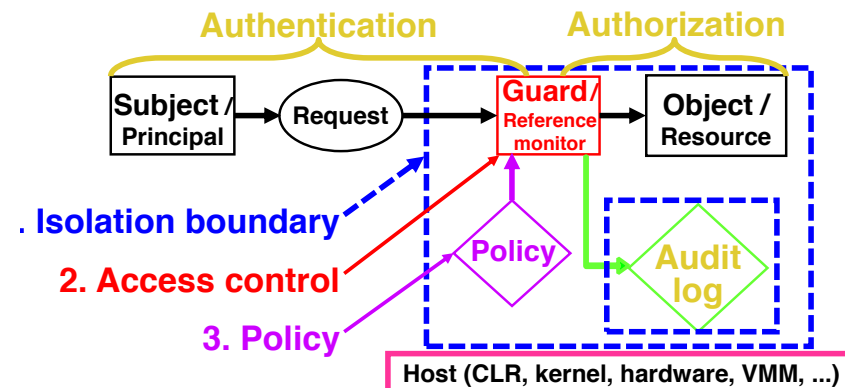
```
chmod u=rwx,g=rx,o=r myfile
chmod 754 myfile
```

<- Same thing

4 stands for "read",
2 stands for "write",
1 stands for "execute", and
0 stands for "no permission."

Reference Monitor

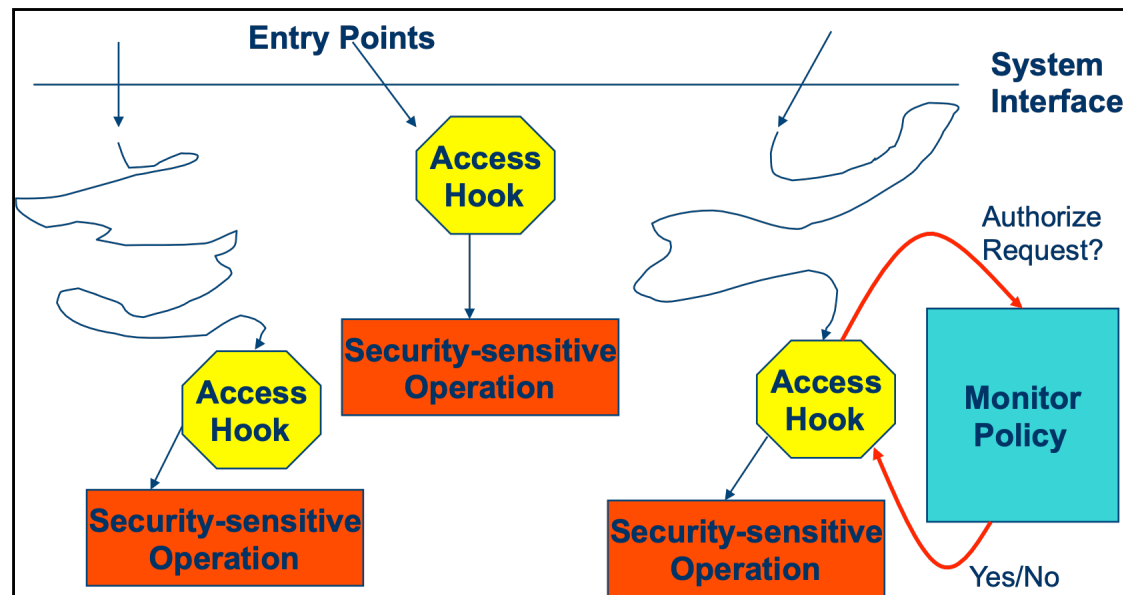
- Mediates requests from applications
 - Implements protection policy
 - Enforces isolation and confinement
- Must always be invoked:
 - Every application request must be mediated
- Tamperproof:
 - Reference monitor cannot be killed
 - ... or if killed, then monitored process is killed too
- Small enough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



Reference Monitor Concept

A reference monitor can enforce a system's protection state if it meets the following three guarantees:

1. Tamperproof
2. Complete Mediation
3. Simple Enough to Verify



Access Control Policy

- The security policy describes the protection state of the system.
- Access Control Matrix
 - Columns are objects, rows are subjects, entries represent rights
 - To determine if S_i has right to access O_i , find appropriate entry
 - There is a matrix for each set of rights

	O_1	O_2	O_3
S_1	Y	Y	N
S_2	N	Y	N
S_3	N	Y	Y

Access Control Policy

- Does the following protection state ensure the secrecy of J's private key file, object O1?

	O ₁	O ₂	O ₃
J	R	RW	RW
S ₂	-	R	RW
S ₃	-	R	RW

Access Control Policy

- Does the following protection state protect the integrity of J's public key file, object O2?

	O ₁	O ₂	O ₃
J	R	RW	RW
S ₂	-	R	RW
S ₃	-	R	RW

Least Privilege

- The principle of least privilege states that a system should only provide those rights needed to perform the processes function, and no more.
- **Implication 1:** You want to reduce the protection state to the smallest possible set of objects
- **Implication 2:** You want to assign the minimal set of rights to each subject
- **Caveat:** You need to provide enough rights and a large enough protection domain to get the job done.

Least Privilege

- Goal: Limit permissions to those required and no more. Consider three processes for user J.

	O₁	O₂	O₃
J₁	R	RW	RW
J₂	-	R	RW
J₃	-	R	RW

Least Privilege

- Goal: Limit permissions to those required and no more. Consider three processes for user J.
 - Restrict privilege of the processes J1 and J2 to prevent leaks to (or through) O3.

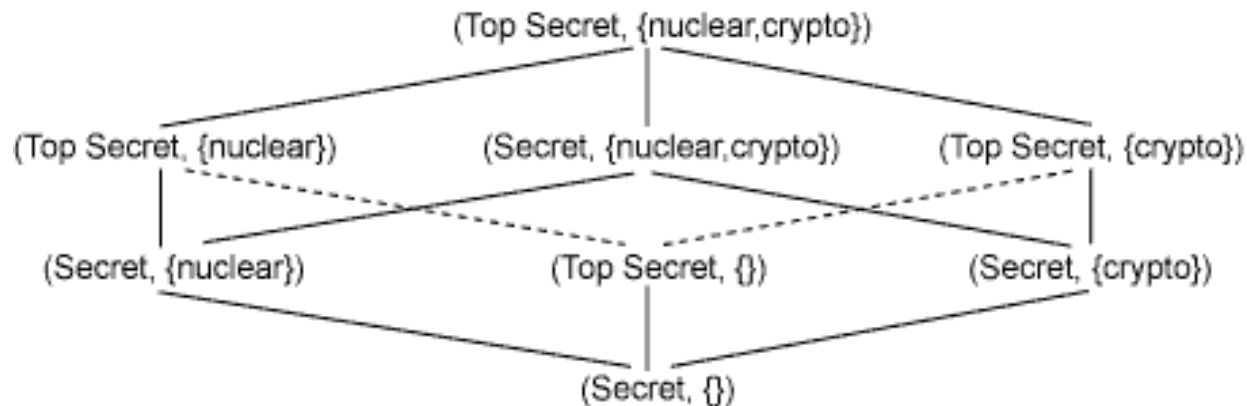
	O ₁	O ₂	O ₃
J ₁	R	RW	-
J ₂	-	R	-
J ₃	-	R	RW

Designing a (Mandatory) Security Policy

- Given a perfect **reference monitor**...
- And what we know about the principle of **least privilege**...
- What should our access control policy (protection state) look like?

Bell-LaPadula Information Flow Model

- Used by the US Military (and many others), the lattice model defines a Multi-Level Security (MLS) policy:
 - *UNCLASSIFIED < CONFIDENTIAL < SECRET < TOP SECRET*
- Categories are represented as an unbounded set:
 - *NUC(lear), INTEL(ligence), CRYPTO(graphy)...*
- State machine (Lattice) specifies permissible actions
- These levels are used for physical government of documents as well.

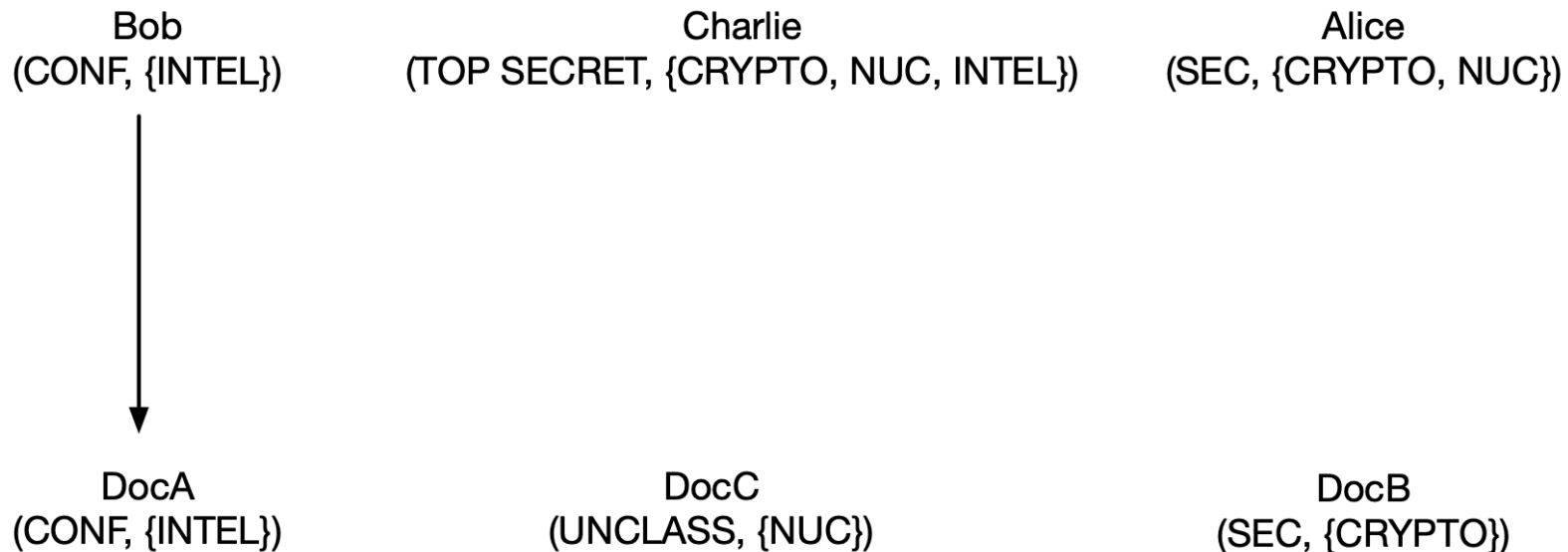


Bell-LaPadula Information Flow Model

- All subjects are assigned clearance levels and compartments:
 - Alice: (SECRET, {CRYPTO, NUC})
 - Bob: (CONFIDENTIAL, {INTEL})
 - Charlie: (TOP SECRET, {CRYPTO, NUC, INTEL})
- All objects are assigned an access class:
 - DocA: (CONFIDENTIAL, {INTEL})
 - DocB: (SECRET, {CRYPTO})
 - DocC: (UNCLASSIFIED, {NUC})

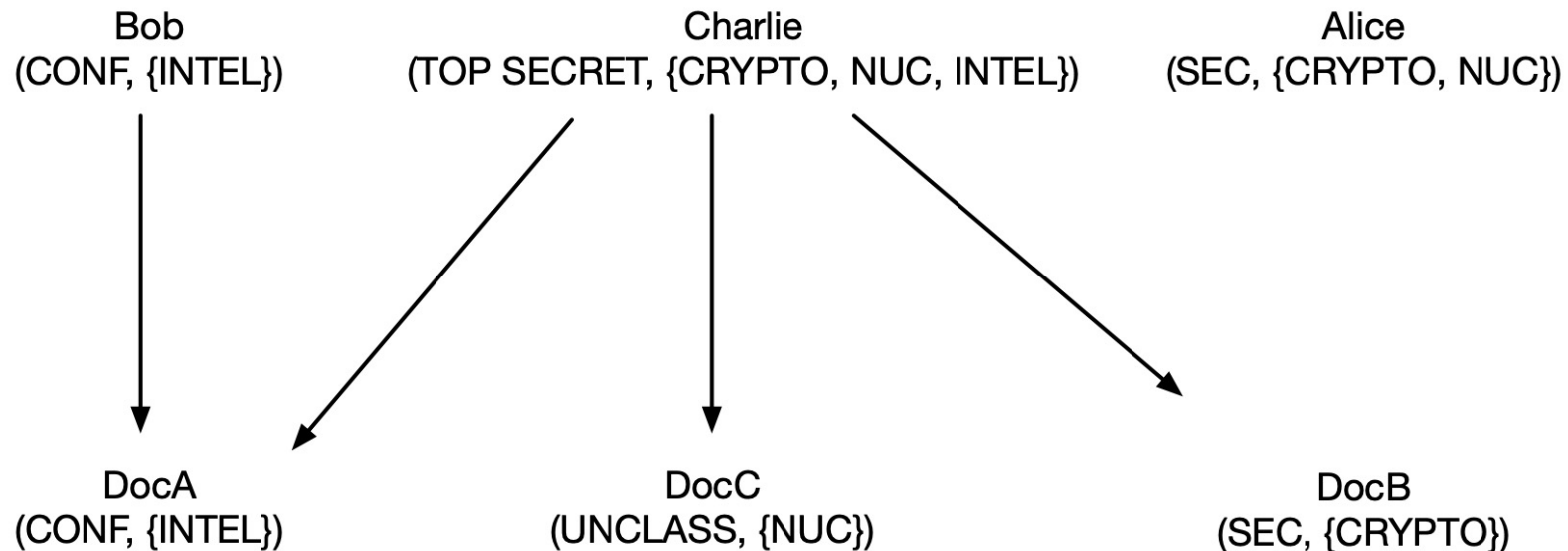
Evaluating Policy

- Access is allowed if:
 - subject clearance level \geq object sensitivity level
 - and**
 - object categories \subseteq subject categories



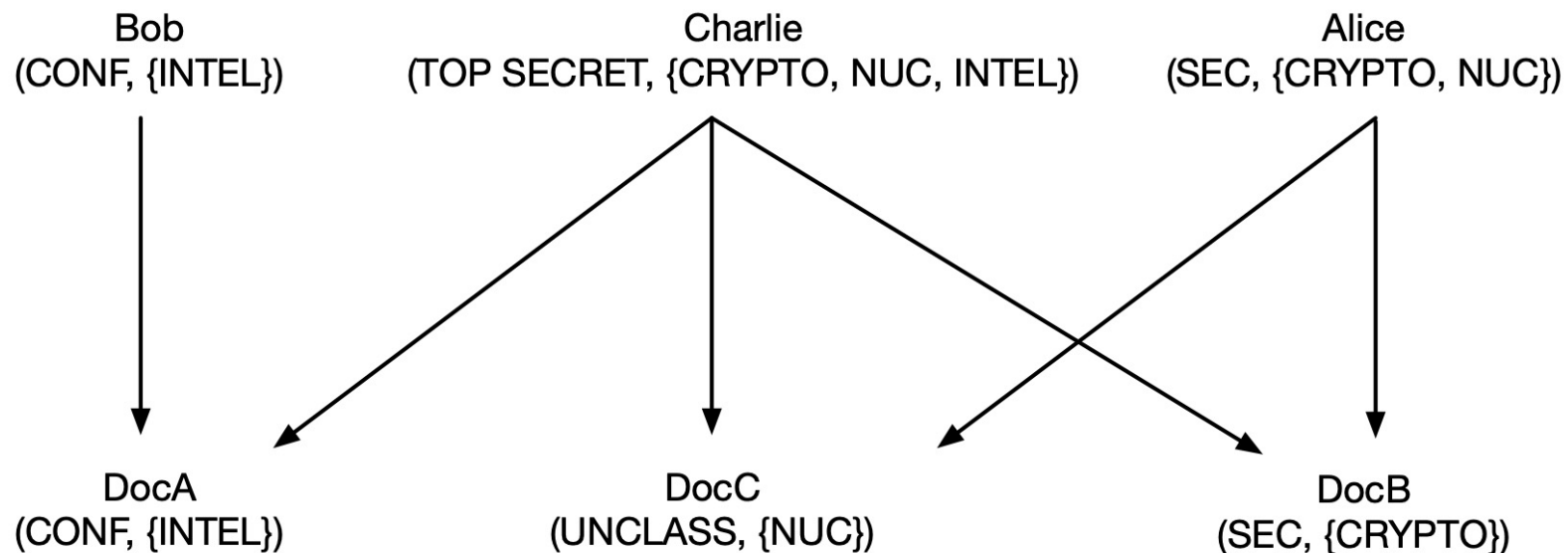
Evaluating Policy

- Access is allowed if:
 - subject clearance level \geq object sensitivity level
 - and**
 - object categories \subseteq subject categories



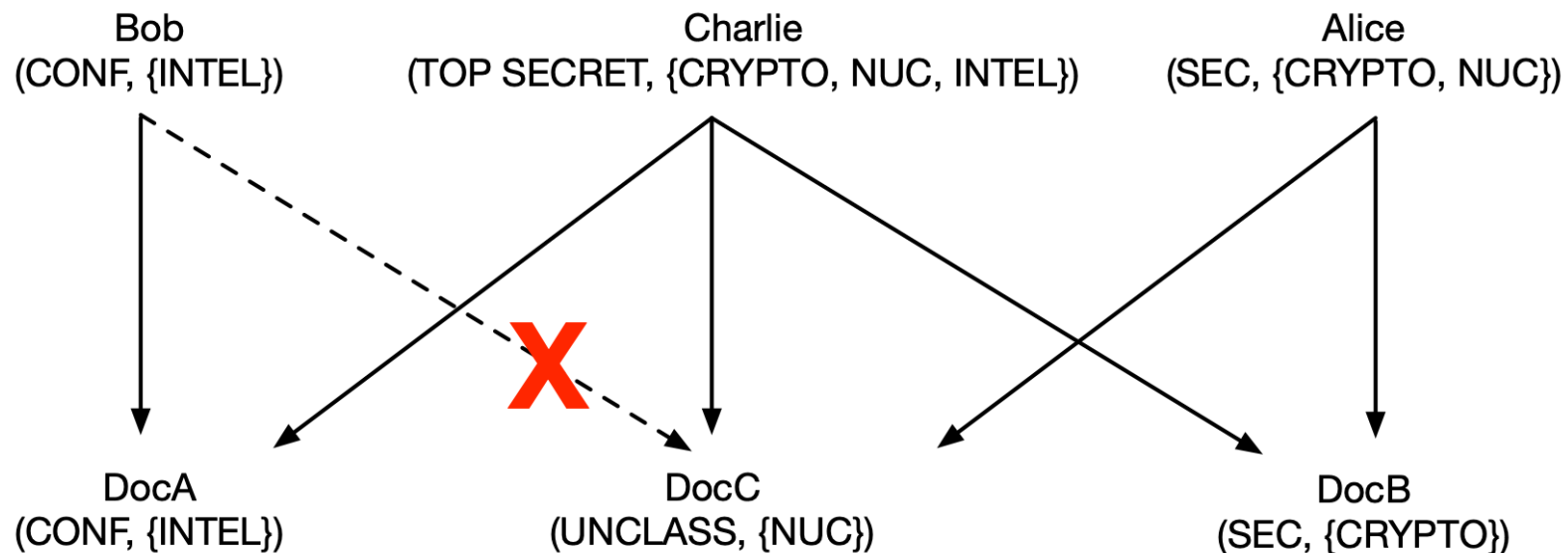
Evaluating Policy

- Access is allowed if:
 - subject clearance level \geq object sensitivity level
 - and**
 - object categories \subseteq subject categories



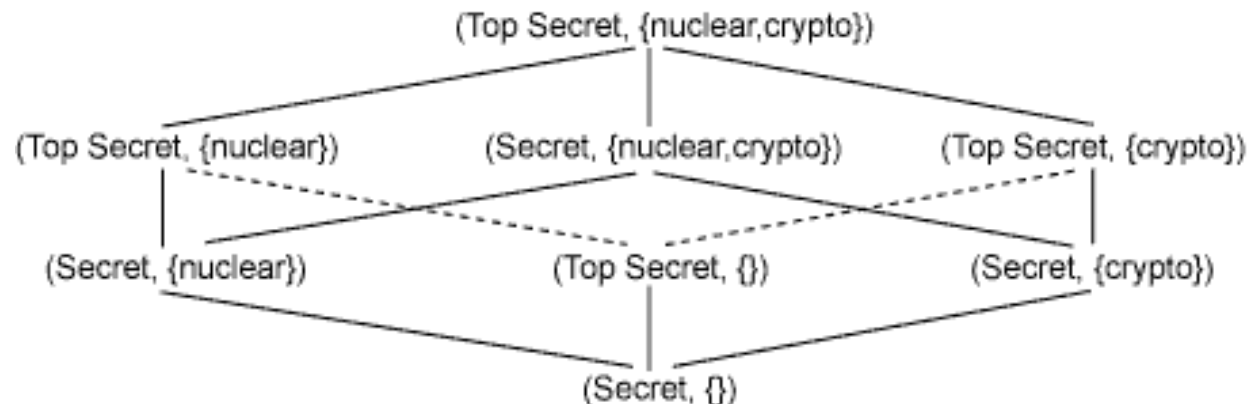
Evaluating Policy

- Access is allowed if:
 - subject clearance level \geq object sensitivity level
 - and**
 - object categories \subseteq subject categories



Bell-LaPadula Information Flow Model

- The Simple Security Property: A subject running at security level k can read only objects at its level or lower. (no read up)
- The * Property: A subject at security level k can write only objects at its level or higher (no write down)



SELinux

SELinux



- Designed by the NSA
- A more flexible solution than MLS
- SELinux Policies are comprised of 3 components:
 - Labeling State defines security contexts for every file (object) and user (subject).
 - Protection State defines the permitted $\langle \text{subject}, \text{object}, \text{operation} \rangle$ tuples.
 - Transition State permits ways for subjects and objects to move between security contexts.
- Enforcement mechanism designed to satisfy reference monitor concept

SELinux Labeling State

- Files and users on the system at boot-time must be associated with their security labels (contexts)
 - Map file paths to labels via regular expressions
 - Map users to labels by name
 - User labels pass on to their initial processes
- How are new files labeled? Processes?

SELinux Protection State

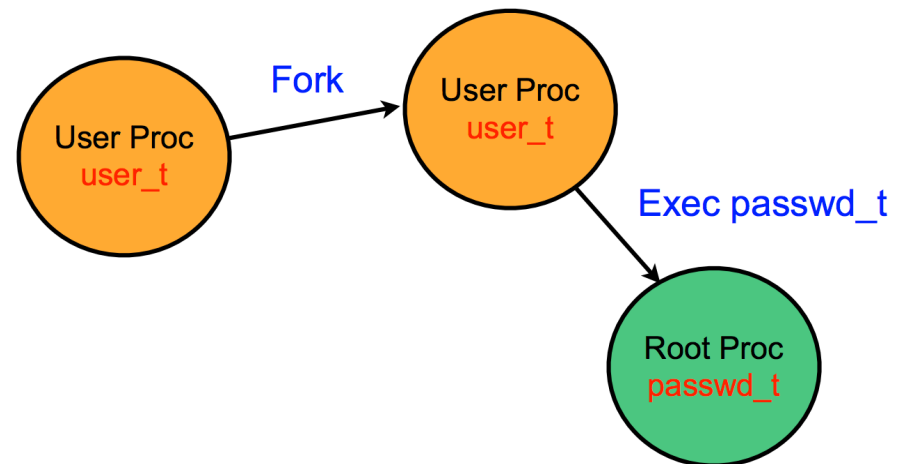
- MAC Policy based on *Type Enforcement*
 - an abstraction of the ACL
- Access Matrix Policy
 - Processes with subject label...
 - Can access file of object label
 - If operations in matrix cell allow
- Focus: Least Privilege
 - Only provide permissions necessary

SELinux Protection State

- Permissions in SELinux can be (at least partially) derived through runtime analysis.
- `audit2allow`:
 - **Step 1**: Run programs in a controlled (no attacker) environment without any enforcement.
 - **Step 2**: Audit all of the permissions used during normal operation.
 - **Step 3**: Generate policy file description
 - Assign subject and object labels associated with program
 - Encode all permissions used into access rules

SELinux Transition State

- Premise: Processes don't need to run in the same protection state all of the time.
- Borrows concepts from Role-Based Access Control
- Example: passwd starts in user context, transitions to privileged context to update /etc/passwd, transitions back to user.



@ 2001 Linux Kernel Summit...



**Include SELinux in
Linux 2.5!**



@ 2001 Linux Kernel Summit...



**Include SELinux in
Linux 2.5!**

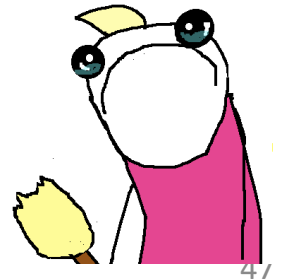


**I'm just not that into
you...**

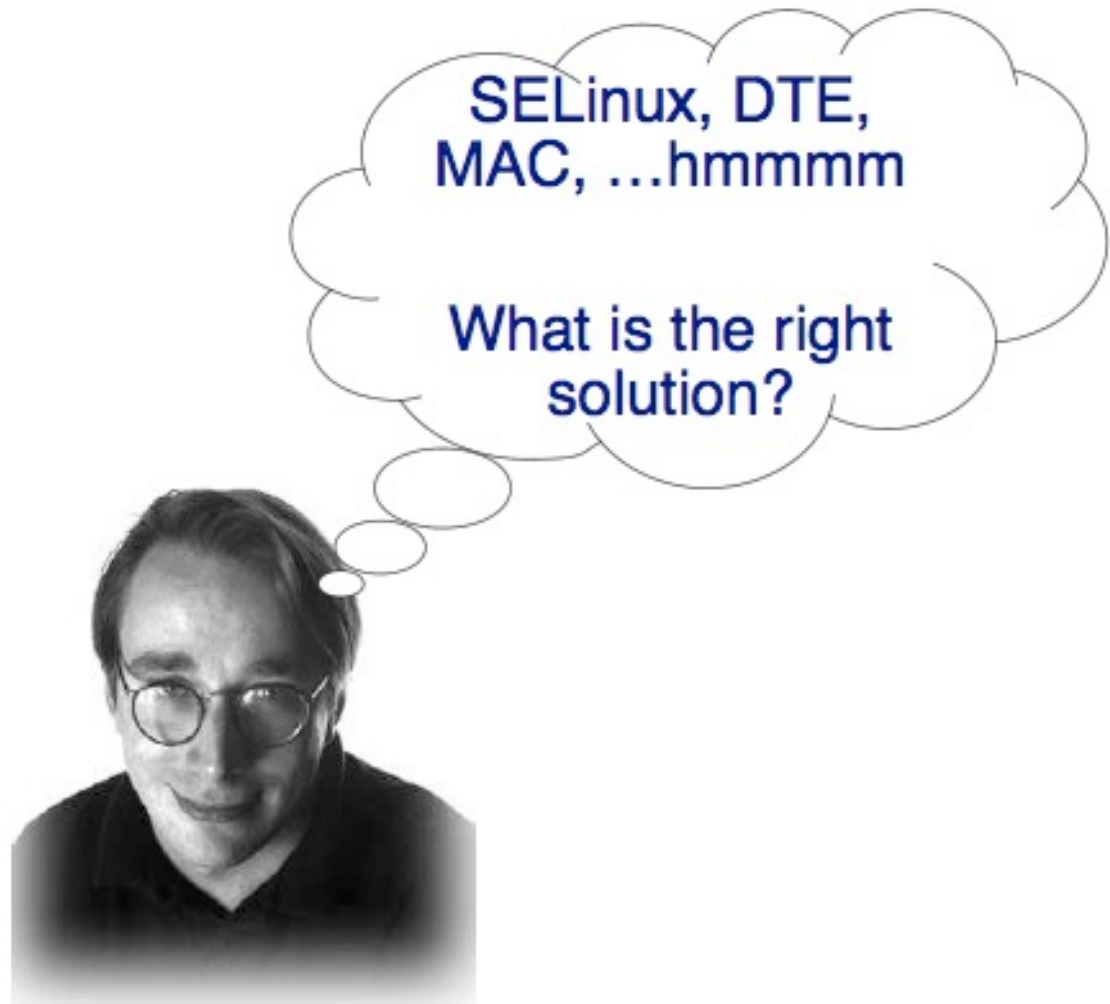


@ 2001 Linux Kernel Summit...

- Patches to the Linux kernel
 - Enforce different access control policy
 - Restrict root processes
 - Some hardening
- Argus PitBull
 - Limited permissions for root services
- RSBAC
 - MAC enforcement and virus scanning
- grsecurity
 - RBAC MAC system
 - Auditing, buffer overflow prevention, /tmp race protection, etc
- LIDS
 - MAC system for root confinement



Linus' Dilemma



Linus' Dilemma

The answer to all computer science problems...

add another layer of abstraction!



SELinux, DTE,
MAC, ...hmmmm

What is the right
solution?

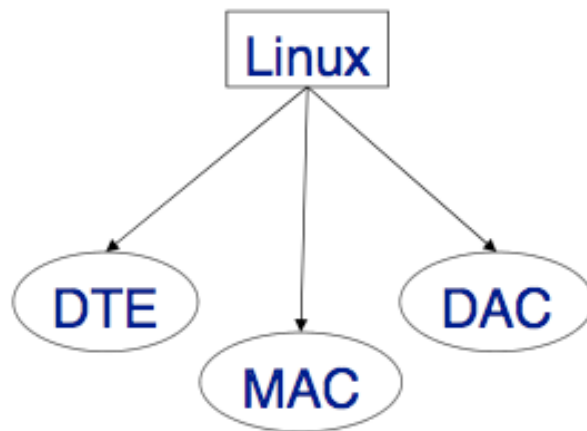
Linux Security Modules

- “to allow Linux to support a variety of security models, so that security developers don't have to have the ‘my dog's bigger than your dog’ argument, and users can choose the security model that suits their needs.”, Crispin Cowan

— <http://mail.wirex.com/pipermail/linux-security-module/2001-April/0005.html>

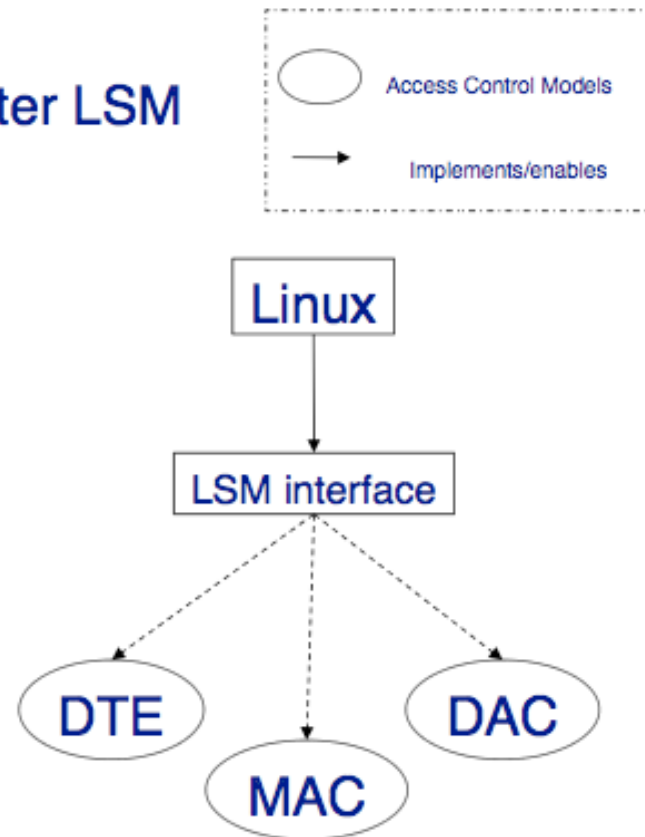
Linux Security Modules

Before LSM



Access control models implemented as
Kernel patches

After LSM


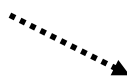


Access control models implemented as
Loadable Kernel Modules

LSM Requirements

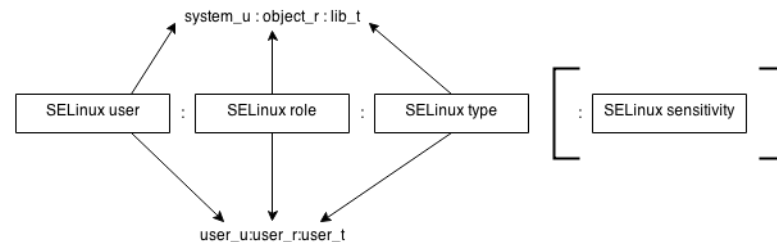
- LSM needs to reach a balance between kernel developer and security developers requirements. LSM needs to unify the functional needs of as many security projects as possible, while minimizing the impact on the Linux kernel.
 - Truly generic
 - conceptually simple
 - minimally invasive
 - Efficient
 - Support for POSIX capabilities
 - Support the implementation of as many access control models as Loadable Kernel Modules

LSM Architecture

- Linux Kernel modified in 5 ways:
 - Opaque security fields added to certain kernel data structures  “controlled data types”
 - Security hook function calls inserted at various points with the kernel code  “security-sensitive operations”
 - A generic security system call added
 - Function to allow modules to register and unregister as security modules
 - Move capabilities logic into an optional security module

Opaque Security Fields

- Enable security modules to associate security information to Kernel objects
- Implemented as void* pointers
- Completely managed by security modules
- What to do about object created before the security module is loaded?



Security Hooks

- Function calls that can be overridden by security modules to manage security fields and mediate access to Kernel objects.
- Hooks called via function pointers stored in `security->ops` table
- Hooks are primarily “restrictive”

Security Hooks

Security check function

linux/fs/read_write.c:

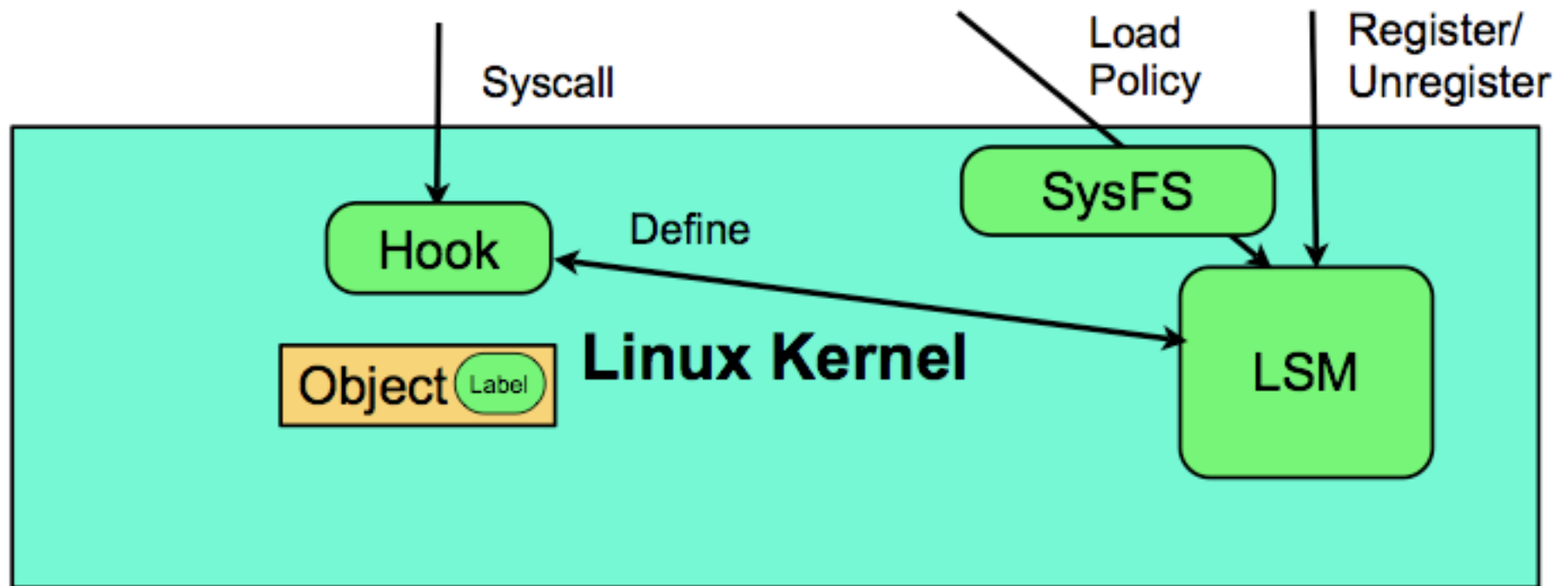
```
ssize_t vfs_read(...) {  
    ...  
    ret = security_file_permission(file, ...);  
    if (!ret) { ...  
        ret = file->f_op->read(file, ...); ...  
    }  
    ...  
}
```

Security sensitive operation

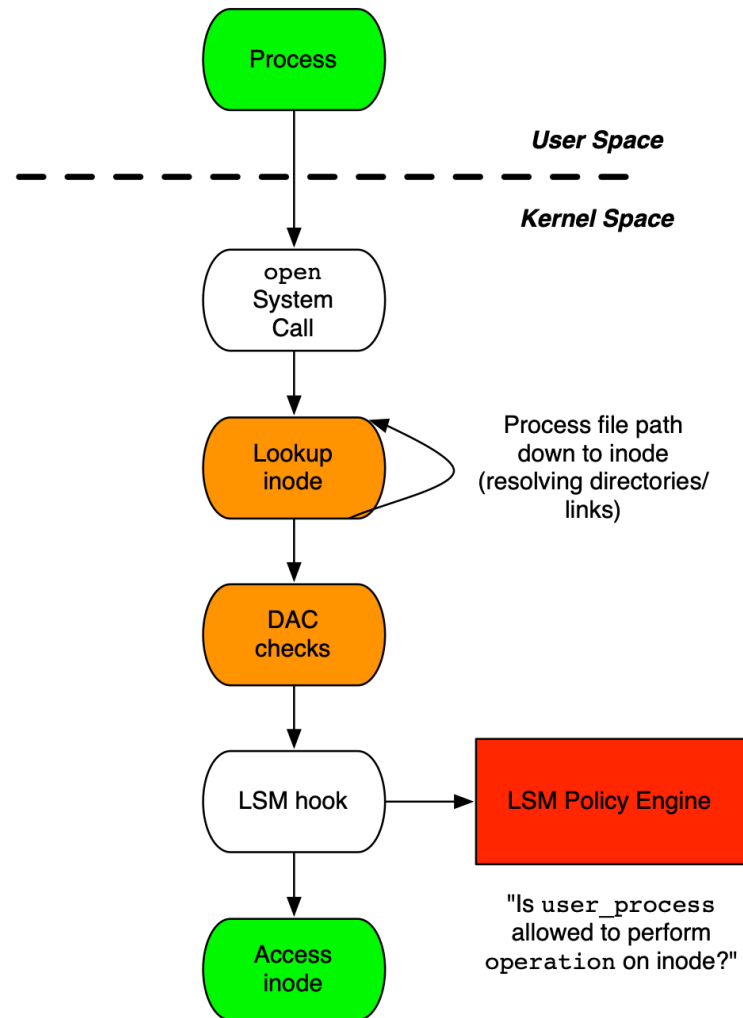
Security Hook Details

- Difference from discretionary controls
 - More object types
 - 29 different object types
 - Per packet, superblock, shared memory, processes
 - Different types of files
 - Finer-grained operations
 - File: ioctl, create, getattr, setattr, lock, append, unlink,
 - System labeling
 - Not dependent on user
 - Authorization and policy defined by module
 - Not defined by the kernel

LSM Hook Architecture



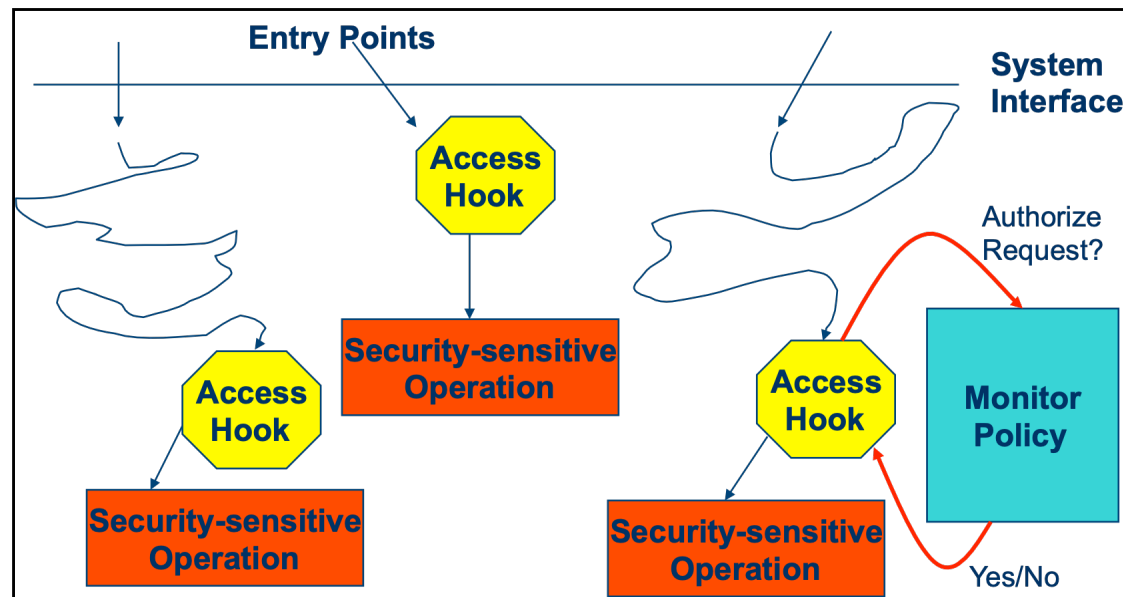
LSM Hook Architecture



Does LSM satisfy the Reference Monitor Concept?

A reference monitor can enforce a system's protection state if it meets the following three guarantees:

1. Tamperproof
2. Complete Mediation
3. Simple Enough to Verify



LSM Analysis

- Early effort to verify LSM
 - ▶ Based on a tool called CQUAL
- Approach
 - ▶ Objects of particular types can be in two states: Checked, Unchecked
 - ▶ To achieve complete mediation, all objects in a “controlled operation” must be checked!

```
/* Code from fs/read.write.c */
sys_llseek(unsigned int fd, ...)
{
    struct file * file;
    ...
    file = fget(fd);
    retval = security_ops->file.ops
        ->llseek(file);
    if (retval) {
        /* failed check, exit */
        goto bad;
    }
    /* passed check, perform operation */
    retval = llseek(file, ...);
    ...
}
```

LSM Analysis

- Time-of-Check-to-Time-of-Use (TOCTTOU) bug: A race condition vulnerability in an authorization check.
- Analysis found TOCTTOU bugs!
- In file locking procedure:
 1. Authorize filp in sys_fcntl...
 2. ... but pass fd again to fcntl_getlk
- Takeaways? Hook Placement (i.e., complete mediation) is hard

```
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
               unsigned int cmd,
               unsigned long arg)
{
    struct file * filp;
    ...
    filp = fget(fd);
    ...

    err = security_ops->file_ops
        ->fcntl(filp, cmd, arg);
    ...
    err = do_fcntl(fd, cmd, arg, filp);
    ...
}

static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
    ...
    switch(cmd){
        ...
        case F_SETLK:
            err = fcntl_setlk(fd, ...);

            ...
        }
    ...
}

/* from fs/locks.c */
fcntl_getlk(fd, ...) {
    struct file * filp;
    ...
    filp = fget(fd);
    /* operate on filp */
    ...
}
```

To Learn More ...

- Books
 - Stallings and Brown, Chapter 12
 - Pfleeger and Pfleeger, Chapter 5
 - Goodrich and Tamassia, Chapter 3
 - Bishop, Chapter 16, 26, 29
- Papers
 - App Isolation: Get the Security of Multiple Browsers with Just One - Chen
 - Native Client: A Sandbox for Portable, untrusted x86 Native Code - Yee*
 - Innovative Instructions and Software Model for Isolated Execution - McKeen
 - The Security Architecture of the Chromium Browser – Barth*
- Talks:
 - Butler Lampson, “Perspectives on Protection and Security,” SOSP’15