# Lecture 36: Reference Monitors

Professor Adam Bates
CS 461 / ECE 422
Fall 2019

# Goals for Today

- Learning Objectives:
  - Understand the history of Reference Monitor *implementations*
  - Explore security primitives introduced in the Multics OS
  - Investigate the design of a modern reference monitor
- Announcements, etc:
  - Forensics CP2 due **December 6th**
  - Enjoy the break! : )

**Reminder**: Please put away devices at the start of class

# Protection System

- Consider how to secure a system. A protection system is made up of not only the protection state (e.g., access control matrix), but also the **administrative operations** to modify the protection state and a **reference monitor**.
- A reference monitor can enforce the protection state if its *implementation* meets the following three guarantees:
    1. Tamperproof
    2. Complete Mediation
    3. Simple Enough to Verify

# Access Controls Discussed

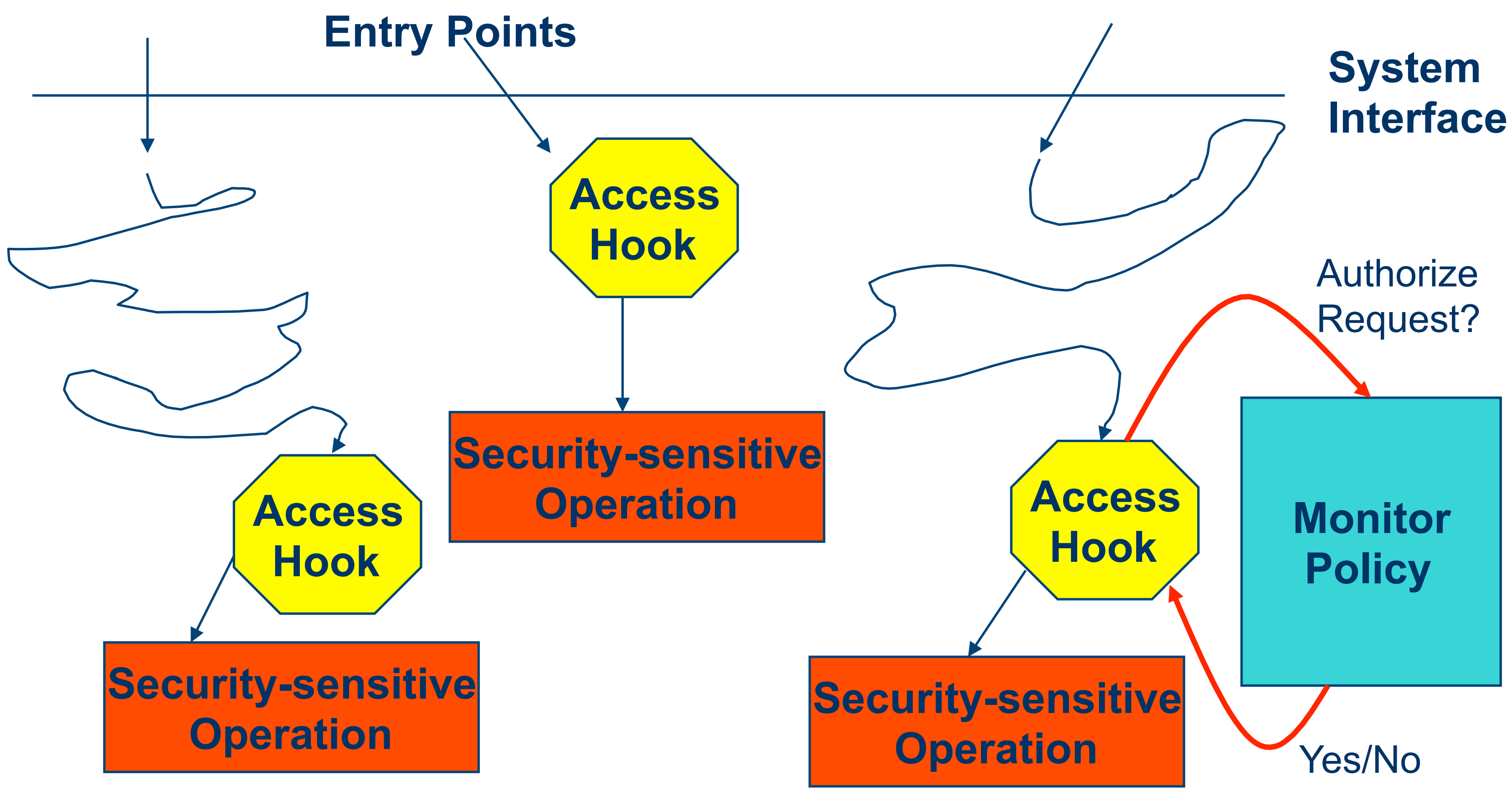
- We've talked about…
- Discretionary Access Control
- Mandatory Access Controls
  - Bell-Lapadula Confidentiality
  - Biba Integrity
  - Low-Water Mark Integrity
  - Clark Wilson Integrity

# All the Access Controls

- Basic Access Matrix
  - UNIX, ACL, various capability systems
- Aggregated Access Matrix
  - TE, RBAC, groups and attributes, parameterized
- Plus Domain Transitions
  - DTE, SELinux, Java
- Lattice Access Control Models
  - Bell-LaPadula, Biba, Denning
- Predicate Models
  - ASL, OASIS, domain-specific models, many others
- Safety Models
  - Take-grant, Schematic Protection Model, Typed Access Matrix

# Reference Monitor

Cool. But how do we implement these models in an operating system?

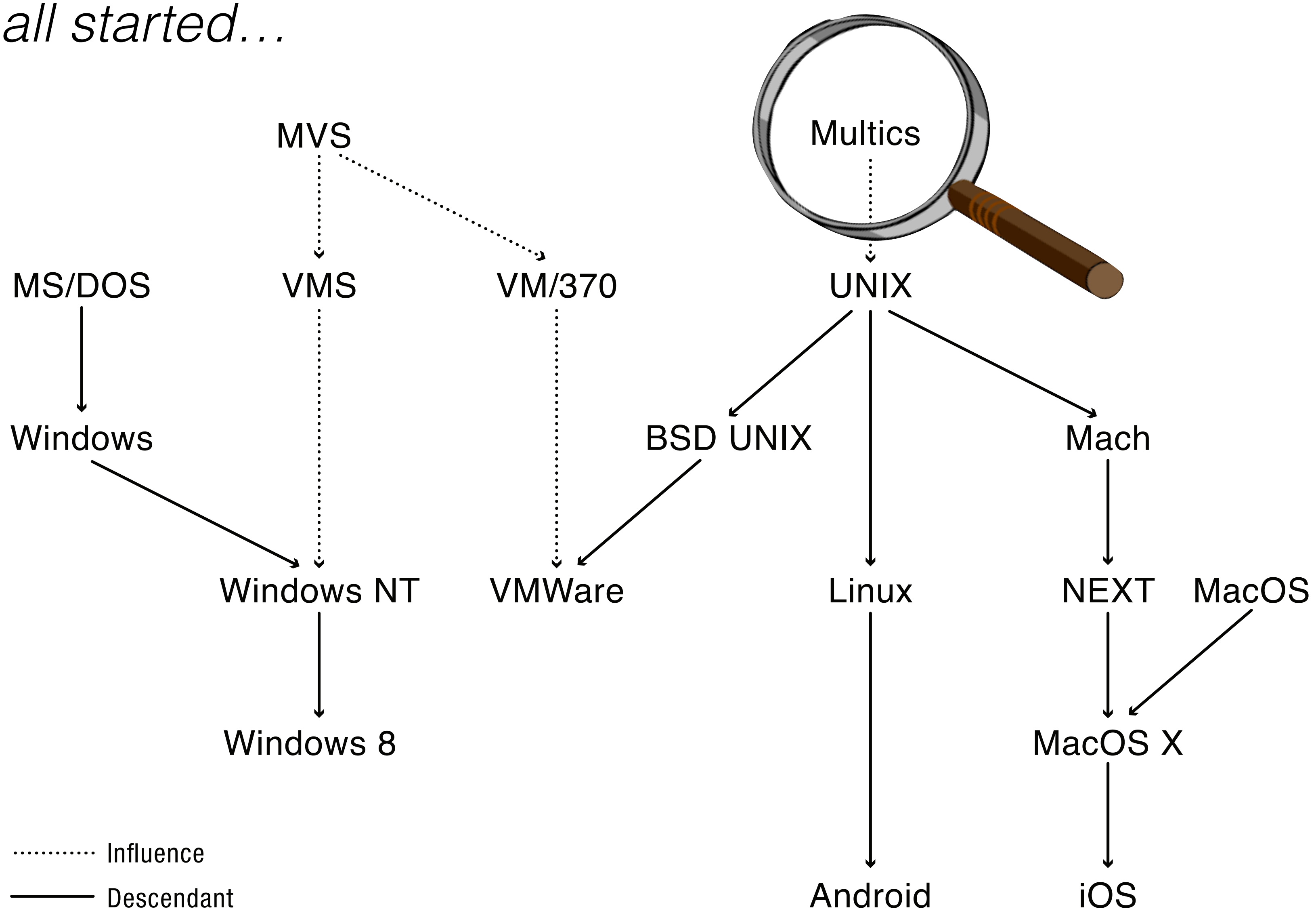


# Reference Monitor

- Where to make access control decisions? (Mediation)
- Which access control decisions to make? (Authorization)
- Decision function: Compute decision based on request and the active security policy

# Flashback

*To understand how to build a reference monitor, let's go back to where it all started…*



# Multics

- Multiprocessing system -- developed many OS concepts
  - Including security
  - Begun in 1965
  - Development continued into the mid-70s, used until 2000
- Initial partners: MIT, Bell Labs, GE/Honeywell
- Other innovations: hierarchical filesystems, dynamic linking
- Subsequent proprietary system, SCOMP, became the basis for secure operating systems design

# Multics Goals

- Secrecy
  - Multilevel Secrecy (MLS)
- Integrity
  - "Rings" of Protection
- Reference Monitoring
  - Mediate segment accesses and ring crossings

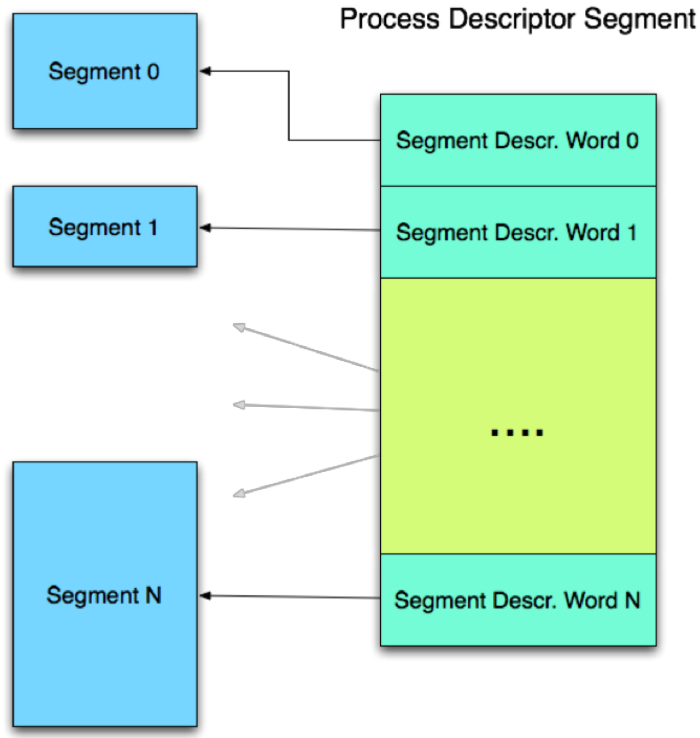
- <u>Resulting system is considered a high point in secure system design!</u>

# Multics Basics

- *Processes* are programs that are executing within Multics (seems obvious now ...)
  - Protection domain is a list of segments
  - Stored in the process descriptor segment
- *Segments* are stored value regions that are accessible by processes, e.g., memory regions, secondary storage
  - Segments can be organized into hierarchies
  - Local segments (memory addresses) are accessed directly
  - Non-local segments are addressed by hierarchy
    - /tape/drive1/top10k
    - /etc/conf/http.conf
    - This is the genesis of the modern hierarchical filesystem!

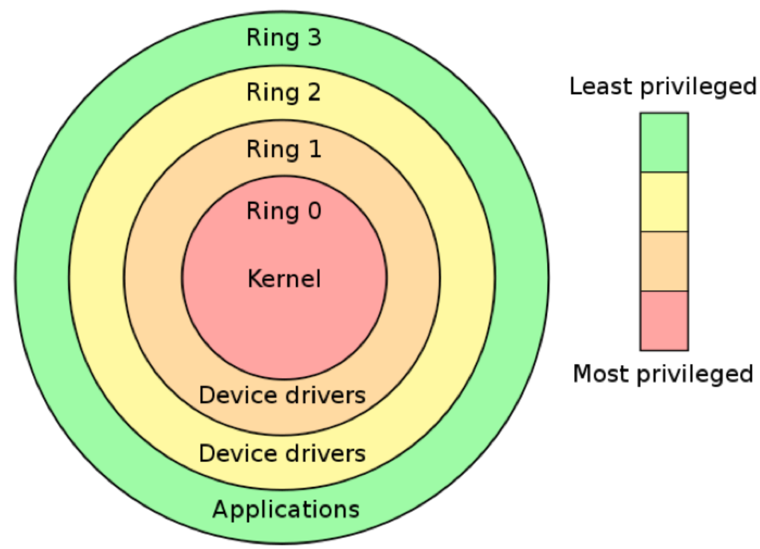
# Segment Management

- PDS acts like segment working set for process
  - Segments are addressed by name (path)
  - If authorized, added to PDS
  - Multics security is defined with respect to segments
- The *supervisor* (kernel) makes decisions and adds to PDS
  - supervisor is isolated by *protection rings*



# Protection Rings

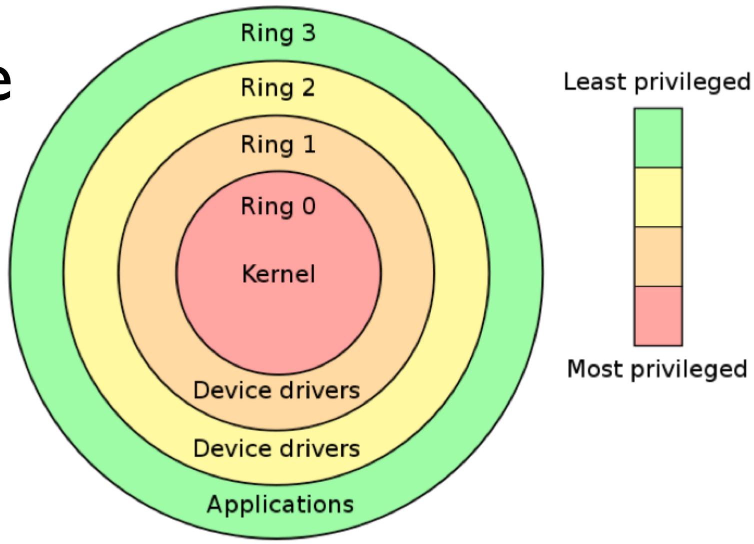
- Successively less-privileged “domains”
- Modern CPUs support 4 rings
  - Use 2 mainly: Kernel and user
- Intel x86 rings
  - Ring 0 has kernel
  - Ring 3 has application code




- Example: Multics (64 rings in theory, 8 in practice)

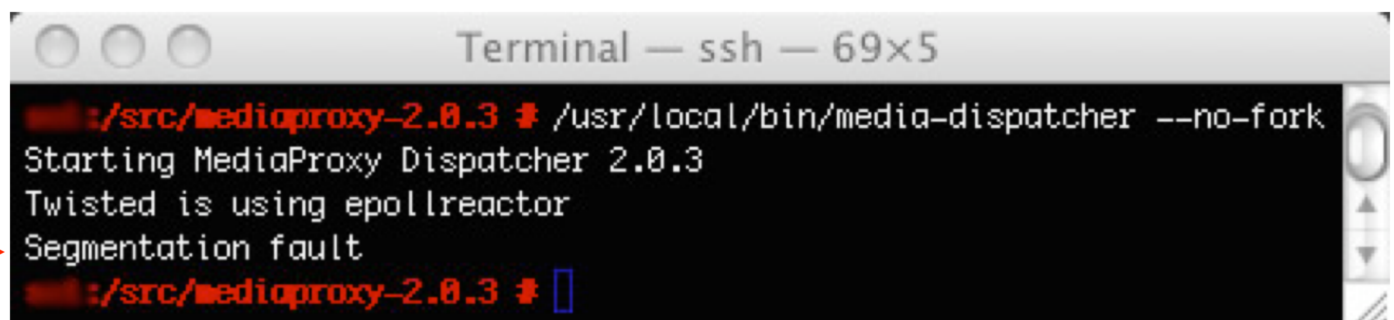
# What are Protection Rings?

- Coarse-grained, Hardware Protection Mechanism
- Boundary between Levels of Authority
  - Most privileged -- ring 0
  - Monotonically less privileged above
- Fundamental Purpose
  - Protect system integrity
    - Protect kernel from services
    - Protect services from apps
    - So on...



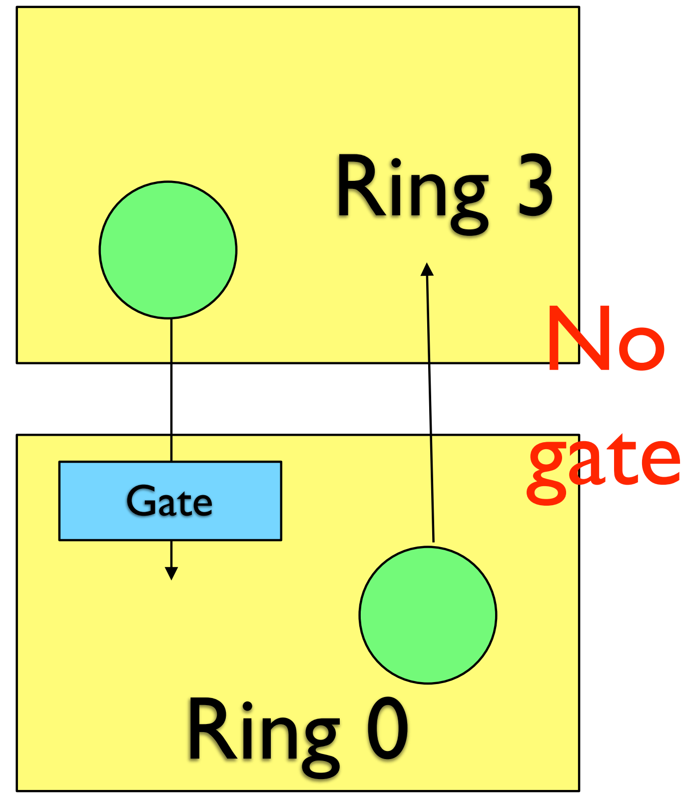
# Intel Protection Ring Rules

- Each Memory Segment has a privilege level (ring number)
- The CPU has a Current Protection Level (CPL)
  - Level of the segment where instructions are being read
- Program can read/write in segments of higher level than CPL
  - kernel can read/write user space
  - user cannot read/write kernel
    - why not?



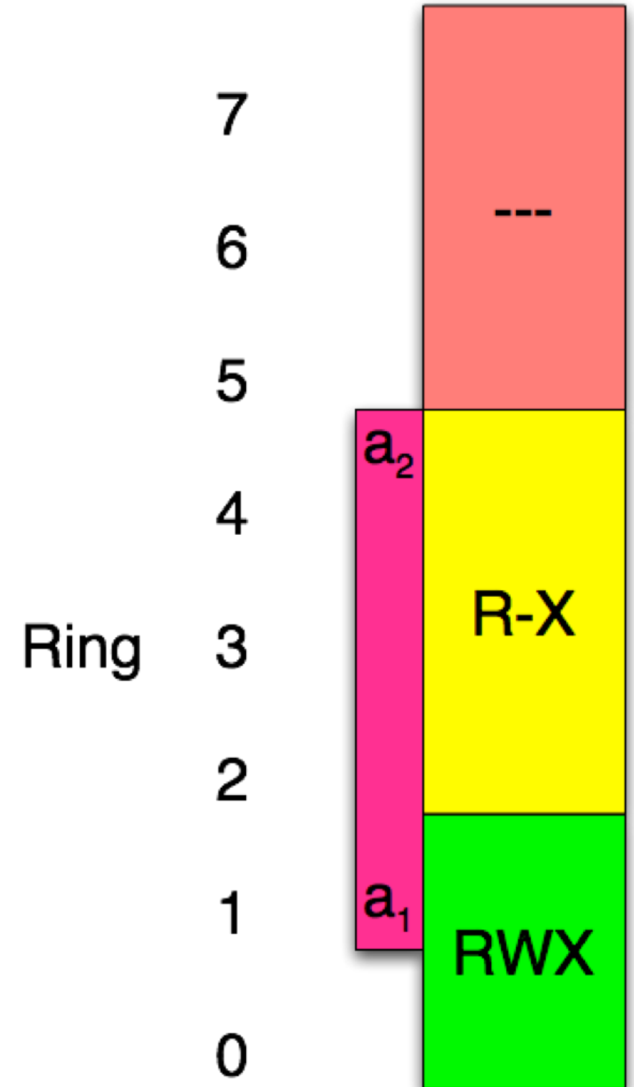
# Protection Ring Rules

- Program cannot call code of *higher privilege* directly
  - Gate is a special memory address where lower-privilege code can call higher
  - Enables OS to control where applications call it
- Known by another name today…?



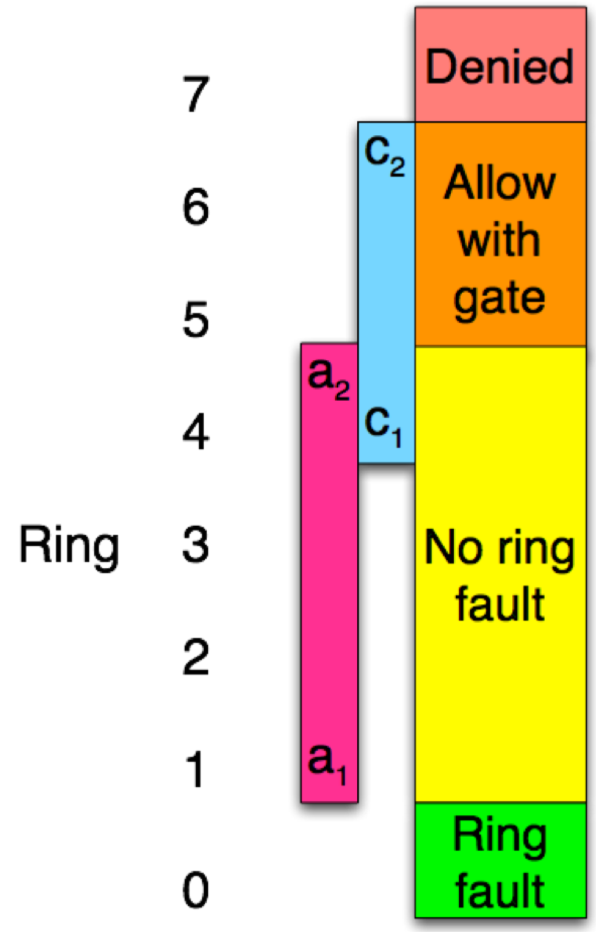
# Multics Runtime Interprettion

- Kernel resides in ring 0
- Process runs in a ring r
  - Access based on current ring
- Process accesses data (segment)
  - Each data segment has an *access bracket*: (a1, a2)
    - a1 <= a2
  - Describes read and write access to segment
    - r is the current ring
    - r <= a1: access permitted
    - a1 < r <= a2: r and x permitted; w denied
    - a2 < r: all access denied



# Multics Runtime Interprettion

- Also different procedure segments
  - with *call brackets*: (c1, c2), c1 <= c2
  - and access brackets (a1, a2)
  - The following must be true (a2 == c1)
  - Rights to execute code in a new procedure segment
    - r < a1: access permitted with ring-crossing fault
    - a1 <= r <= a2 = c1: access permitted and no fault
    - a2 < r <= c2: access permitted through a valid gate
    - c2 < r: access denied
- What's it mean?
  - case 1: ring-crossing fault changes procedure's ring
    - increases from r to a1
  - case 2: keep same ring number
  - case 3: gate checks args, decreases ring number
- Target code segment defines the new ring



# Examples

- Process in ring 3 accesses data segment
  - access bracket: (2, 4)
  - What operations can be performed?
- Process in ring 5 accesses same data segment
  - What operations can be performed?
- Process in ring 5 accesses procedure segment
  - access bracket (2, 4)
  - call bracket (4, 6)
  - Can call be made?

# Multics Segments

- *Named segments* are protected by access control lists and MLS protections
  - Hierarchically arranged
  - Precursor to hierarchical file systems
- *Memory segment* access is controlled by hardware monitor
  - Multics hardware retrieves *segment descriptor word* (SDW)
  - Like a file descriptor
  - Based on rights in the SDW determines whether can access segment
- *Master mode* (like root) can override protections

- Access a directory or SDW on each instruction!

# Fastforward

**The year is 2001…**

# 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 <subject,object,operation> 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 are 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

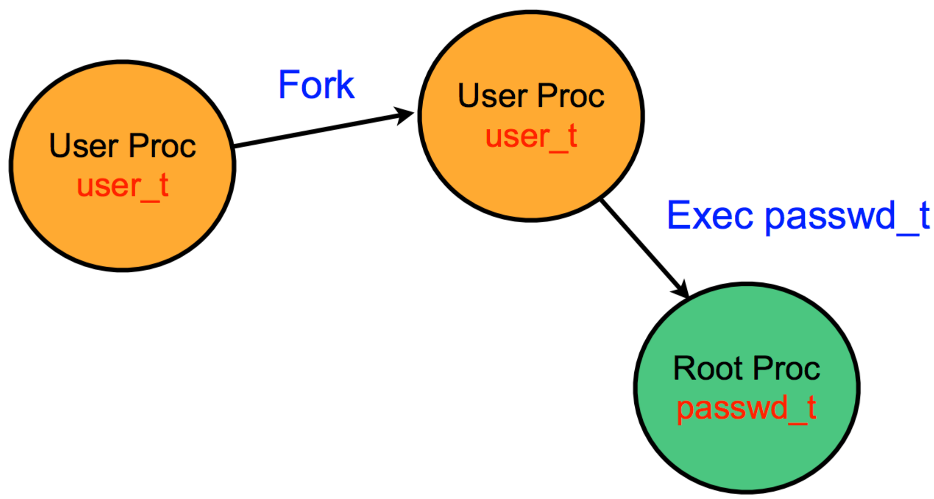
| | $O_1$ | $O_2$ | $O_3$ |
|---|---|---|---|
| $S_1$ | Y | Y | N |
| $S_2$ | N | Y | N |
| $S_3$ | N | Y | Y |

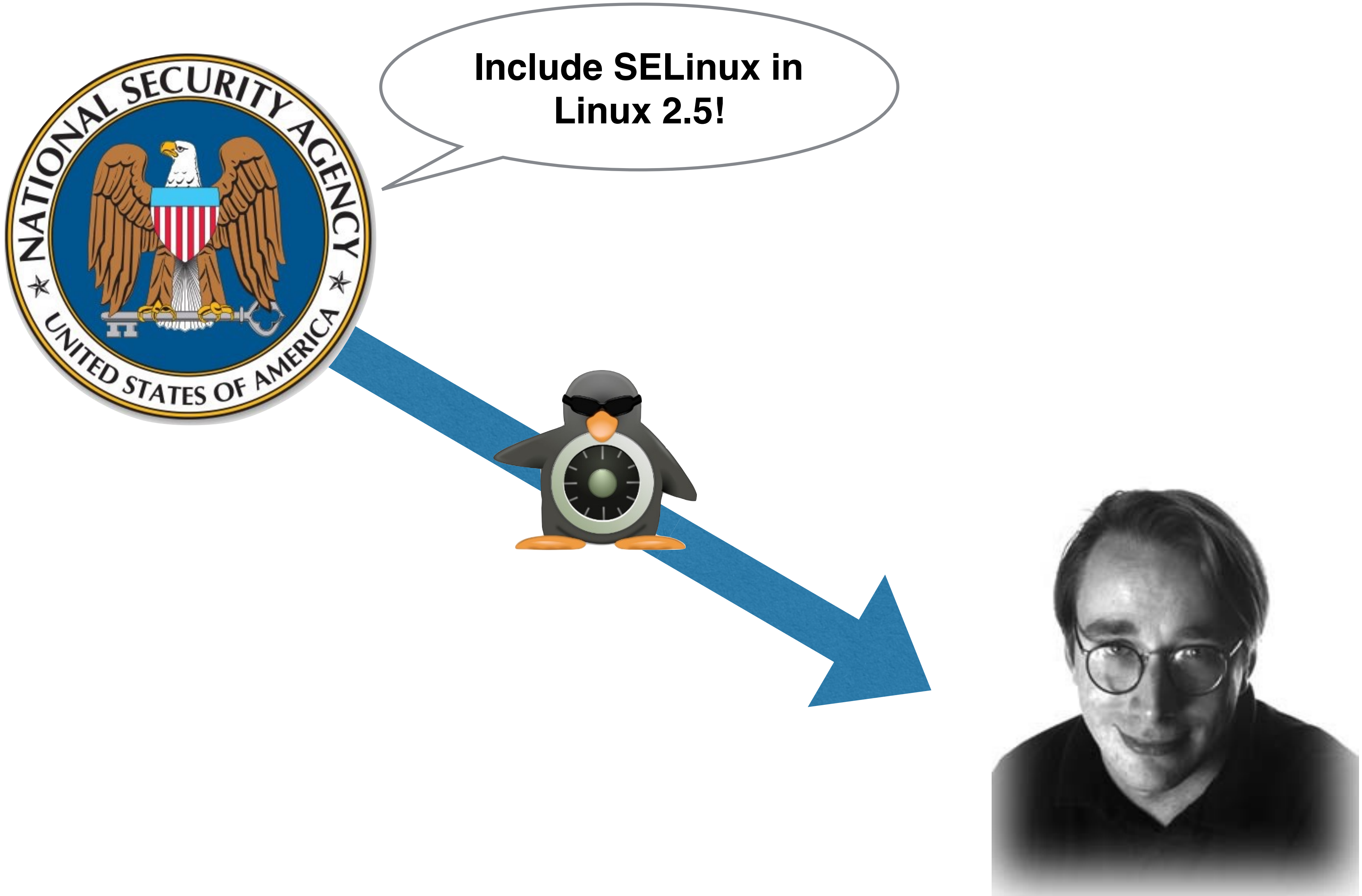
# SELinux Protection State

- 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

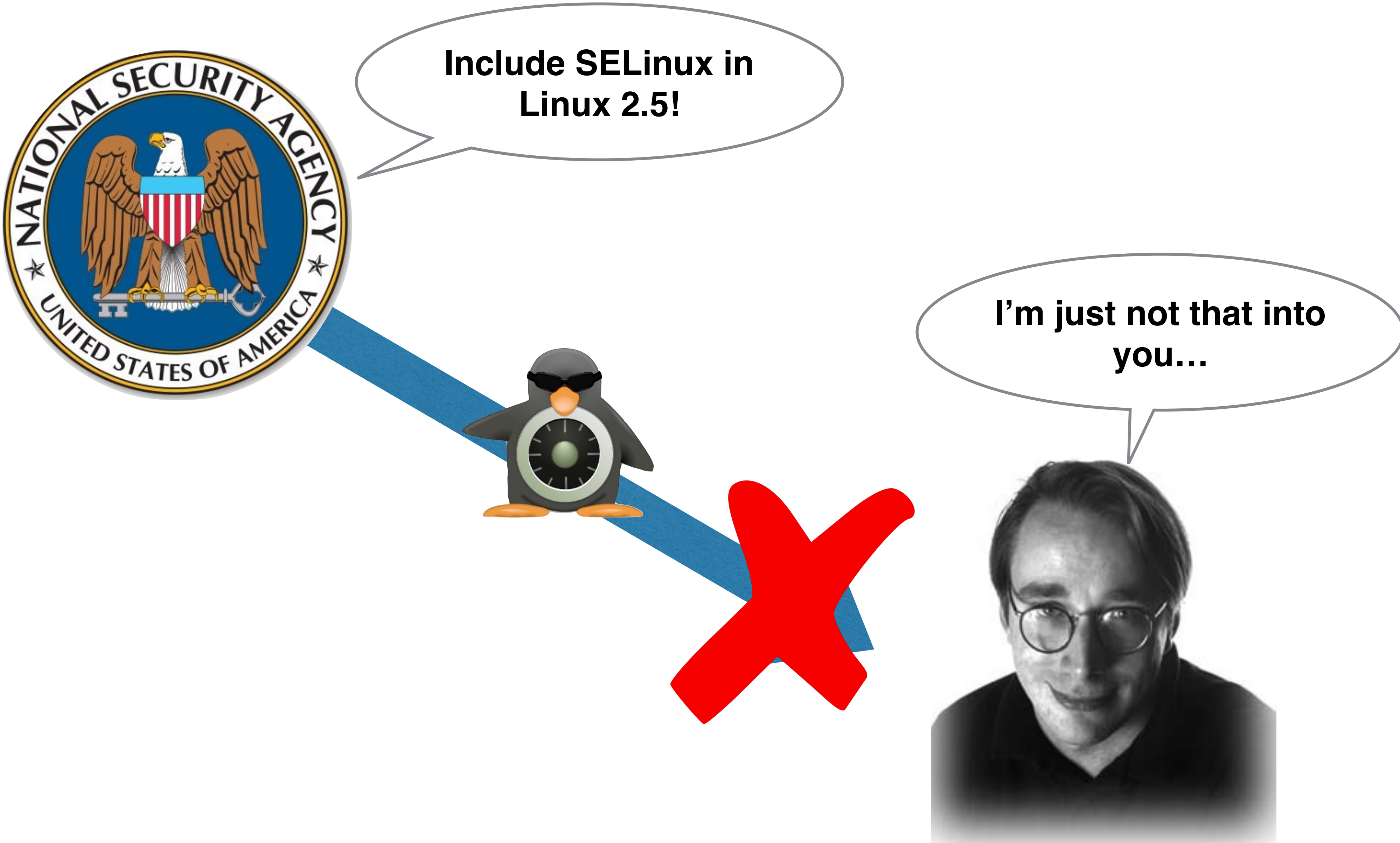
- Premise: Processes don't need to run in the same protection state all of the time.
- Borrows concepts from *Role-Based Access Control*
- Example: `passwd` starts in user context, transitions to privileged context to update /etc/passwd, transitions back to user.





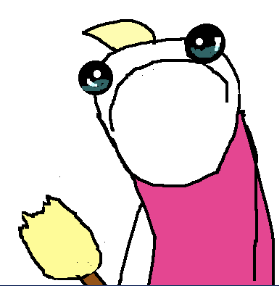
Include SELinux in Linux 2.5!
NATIONAL SECURITY AGENCY
UNITED STATES OF AMERICA

# @ 2001 Linux Kernel Summit…

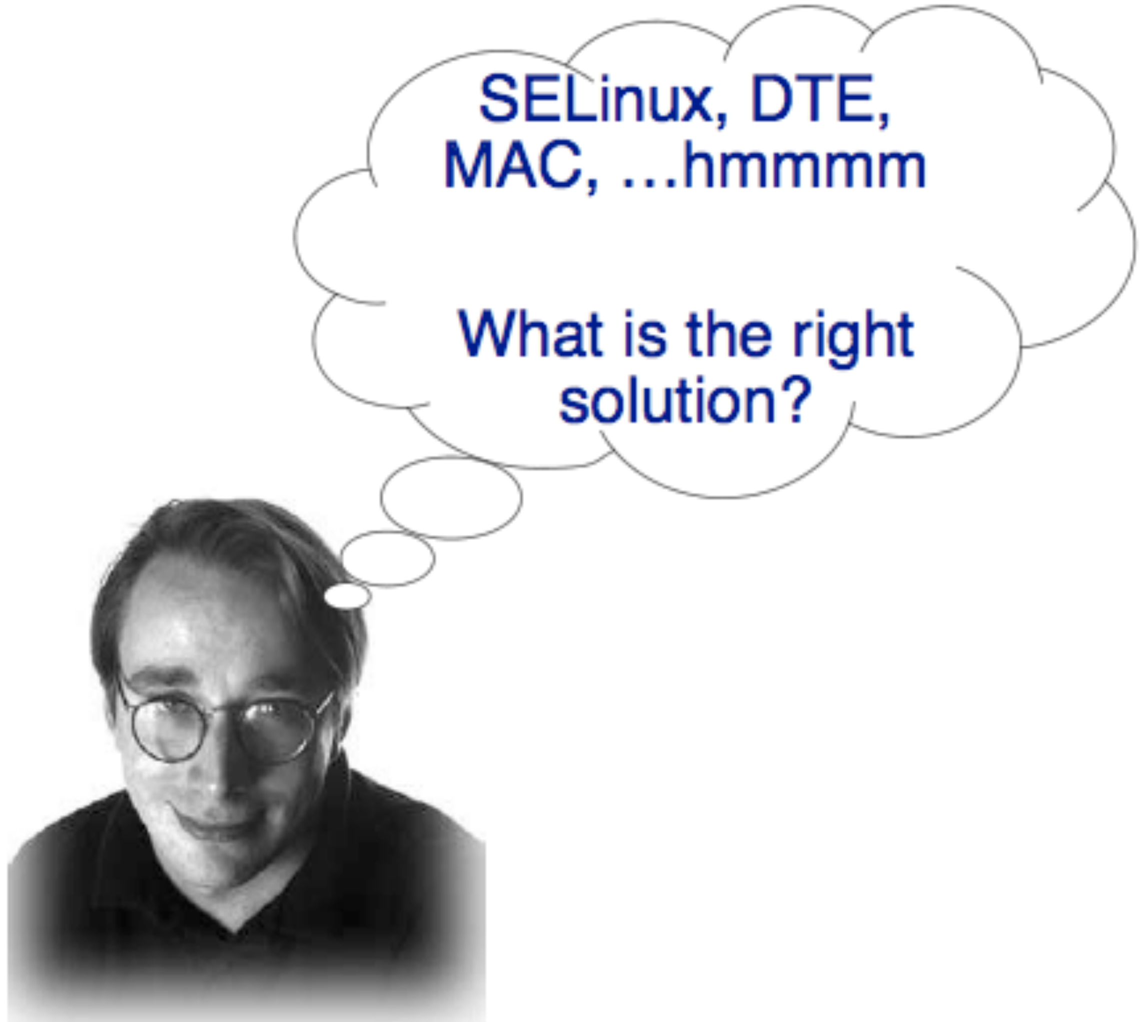


# Linux Security circa 2000

- 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



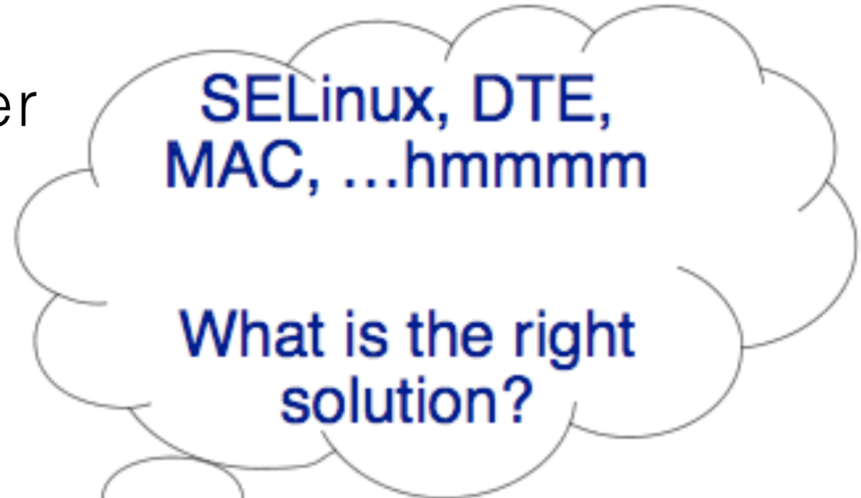
# Linus's Dilemma



# Linus's Dilemma

The answer to all computer science problems…

add another layer of abstraction!

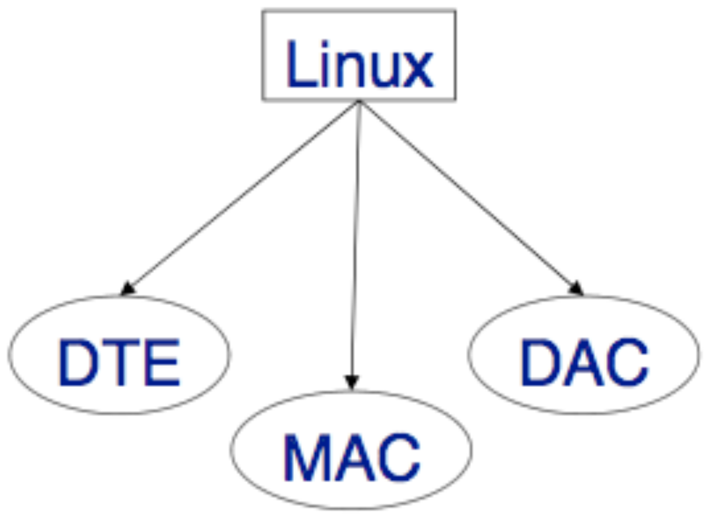



# Linux Security Modules

- "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

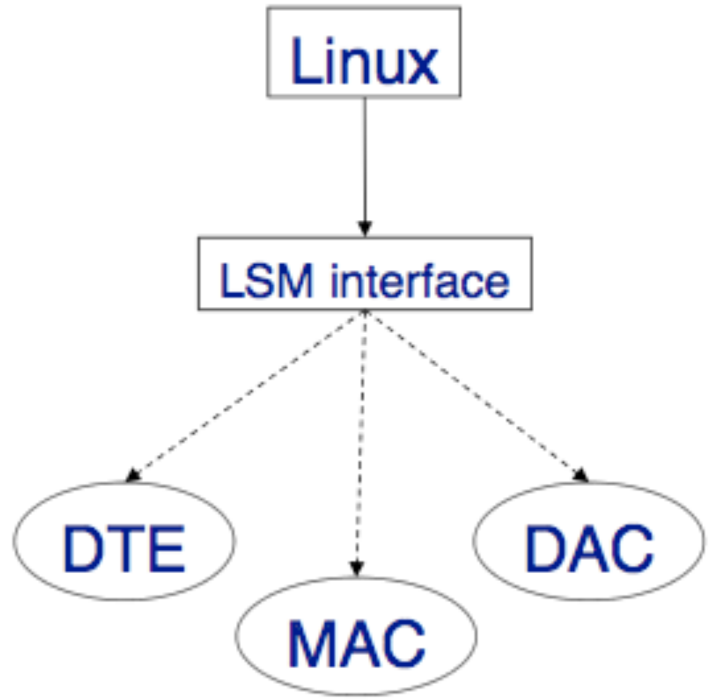
# Linux Security Modules

## Before LSM




Access control models implemented as Kernel patches

## After LSM


Access control models implemented as Loadable Kernel Modules

# LSM Requirements

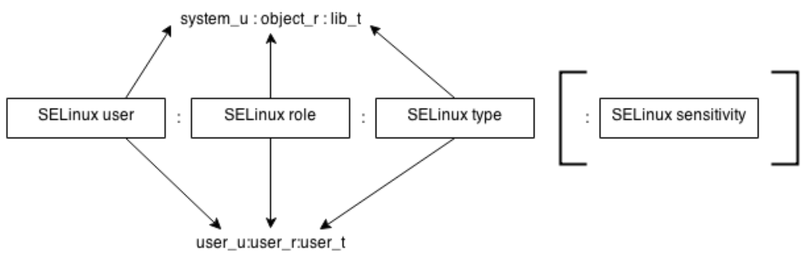
- LSM needs to reach a balance between kernel developer and security developers requirements. LSM needs to unify the functional needs of as many security projects as possible, while minimizing the impact on the Linux kernel.
  - Truly generic
  - conceptually simple
  - minimally invasive
  - Efficient
  - 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 unregistered 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

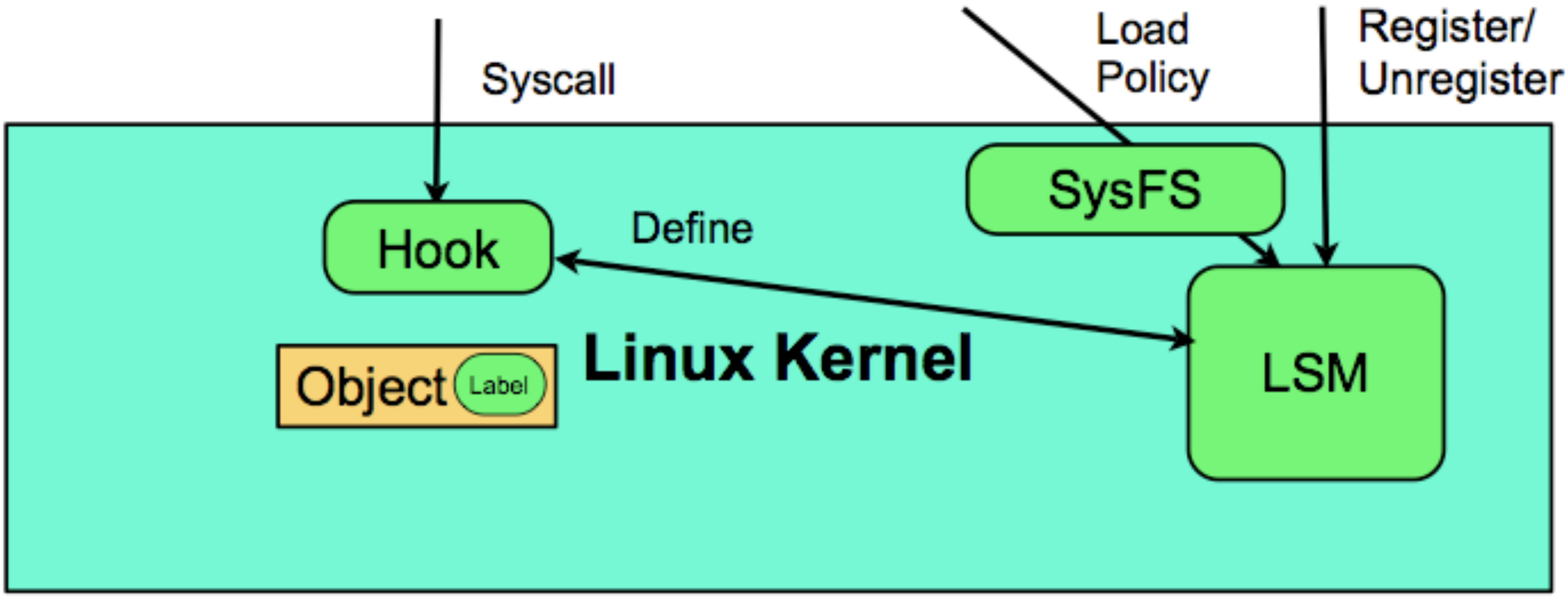
```
linux/fs/read_write.c:

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

Security sensitive operation

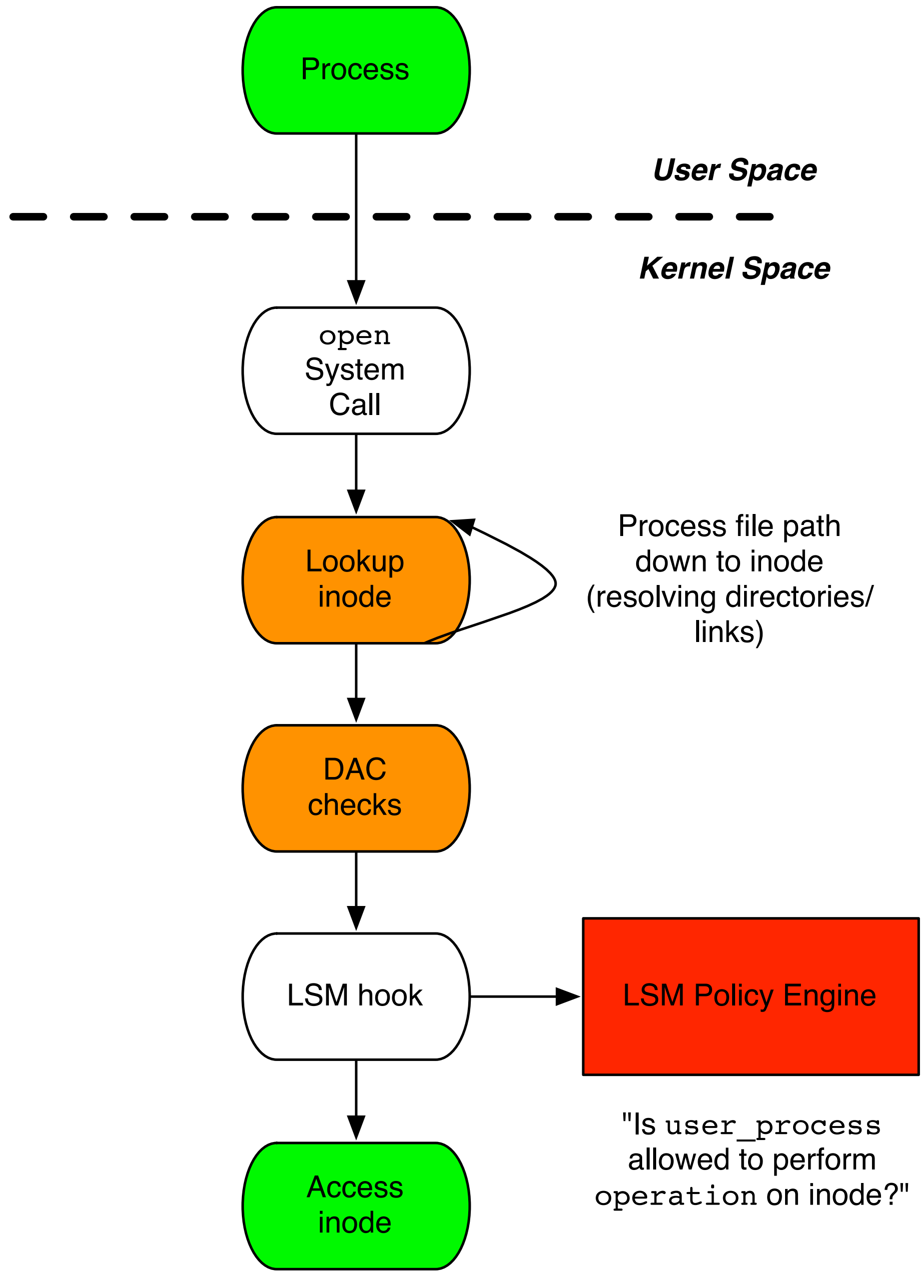
# Security Hook Details

- 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



# LSM Performance

- Microbenchmark: LMBench
  - Compare standard Linux Kernel 2.5.15 with Linux Kernel with LSM patch and a default capabilities module
  - Worst case overhead is 5.1%
- Macrobenchmark: Kernel Compilation
  - Worst case 0.3%
- Macrobenchmark: Webstone
  - With Netfilter hooks 5-7%
  - Uni-Processor 16%
  - SMP 21% overhead

# LSM Use

- Available in Linux 2.6
  - Packet-level controls upstreamed in 2.6.16
- Modules
  - POSIX Capabilities module
  - SELinux module
  - Domain and Type Enforcement
  - Openwall, includes grsecurity function
  - LIDS
  - AppArmor
- Not everyone is in favor
  - How does LSM impact system hardening?

# LSM Analysis

- LSM is mainly responsible for complete mediation
  - What was the basis for choosing security-sensitive operations?
  - Pretty ad hoc…
- How did that work out?

# LSM Analysis

- Static analysis by Zhang, Edwards, and Jaeger [USENIX Security 2002]
  - Based on a tool called CQUAL
- Approach
  - Objects of particular types can be in two states
    - Checked, Unchecked
  - All objects in a "controlled operation" must be checked
    - Structure member access on objects

```
/* Code from fs/read_write.c */
sys_lseek(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

- Static analysis by Zhang, Edwards, and Jaeger [USENIX Security 2002]
  - Based on a tool called CQUAL
- Found TOCTTOU bugs:
  1. Authorize filp in sys_fcntl…
  2. … but pass fd again to fcntl_getlk
- Takeaways? Hook Placement (i.e., **<u>Complete Mediation</u>**) 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 */
  ...
}
```

- Automatically inferring security specifications from code – Tan, Zhang, Ma, Xiong, Zhou [USENIX Security 2008]
  - Derive security specification from code analysis
  - Check for violations, e.g., are all read calls behind the correct authorization hook?

**Forgot to call security_file_permission().**

**Security check**

```
linux/fs/read_write.c:

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

```
linux/fs/readdir.c:

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

```
linux/fs/read_write.c:

ssize_t do_readv_writev(...) {
  ...


  ret = file->f_op->readv(file, ...);
  ...
}
```

**Same security sensitive operation: file read/write**

# Conclusions

- Access Control is supported in operating systems through the "Reference Monitor" concept
- LSM is a framework for designing reference monitors
- Today, many security modules exist
  - Must define authorization hooks to mediate access
  - Must expose a policy framework for specifying which accesses to authorize
- Policy Challenges
  - Is language expressive enough to capture the goals of the user?
  - Is language simple/intuitive enough for ease of use?
    - Policy misconfiguration breaks security!!