Lecture 36:
Linux Security Module

Professor Adam Bates
CS 461 / ECE 422
Fall 2019

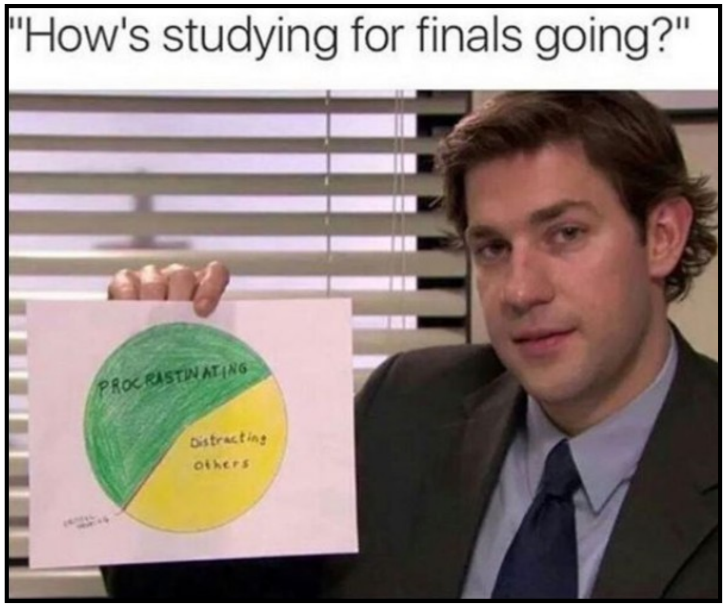
Goals for Today

- Learning Objectives:
  - Investigate the design of a modern reference monitor
- Announcements, etc:
  - Forensics CP2 due **December 6th**
  - Final Exam — 7pm December 13th

Reminder: Please put away devices at the start of class

Final Details

- December 13th, 7-10pm
  - Here, 1404 Siebel
- Multiple choice + short answer
- **Closed book.**
- *No electronic devices permitted (or necessary)!*
- **Content**: All lectures, and MP3, MP4, MP5.
- Sample exams are not available for the final; feel free to re-review midterm sample exams



Final Details

- Changes from midterm:
  - Multiple Choice will now be scantron. Only one answer is correct per question. No points deducted for guessing.
  - More multiple choice questions, ~30+ as opposed to 20.
  - More short answer questions
  - MP questions will account for a smaller percentage of overall score.

SELinux

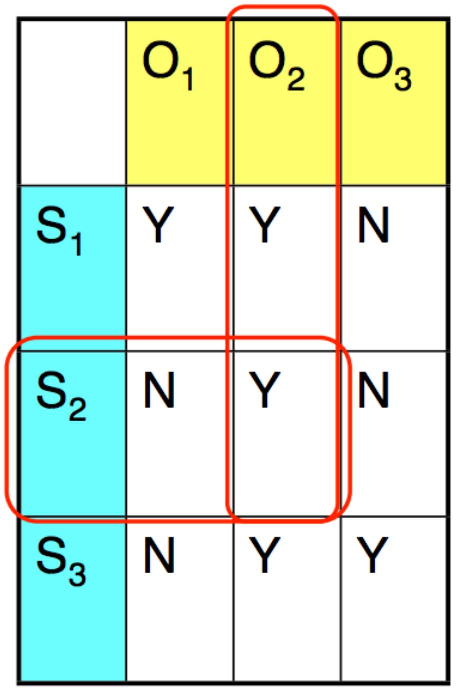
- Designed by the NSA
- A more flexible solution than MLS
- SELinux Policies are comprised of 3 components:
  - Labeling State defines security contexts for every file (object) and user (subject).
  - Protection State defines the permitted <subject,object,operation> tuples.
  - Transition State permits 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



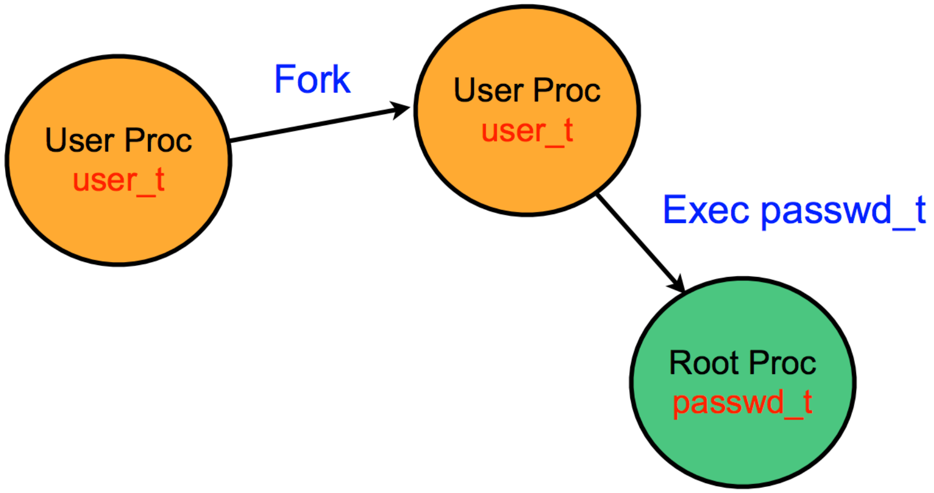
	O_1	O_2	O_3
S_1	Y	Y	N
S_2	N	Y	N
S_3	N	Y	Y

SELinux Protection State

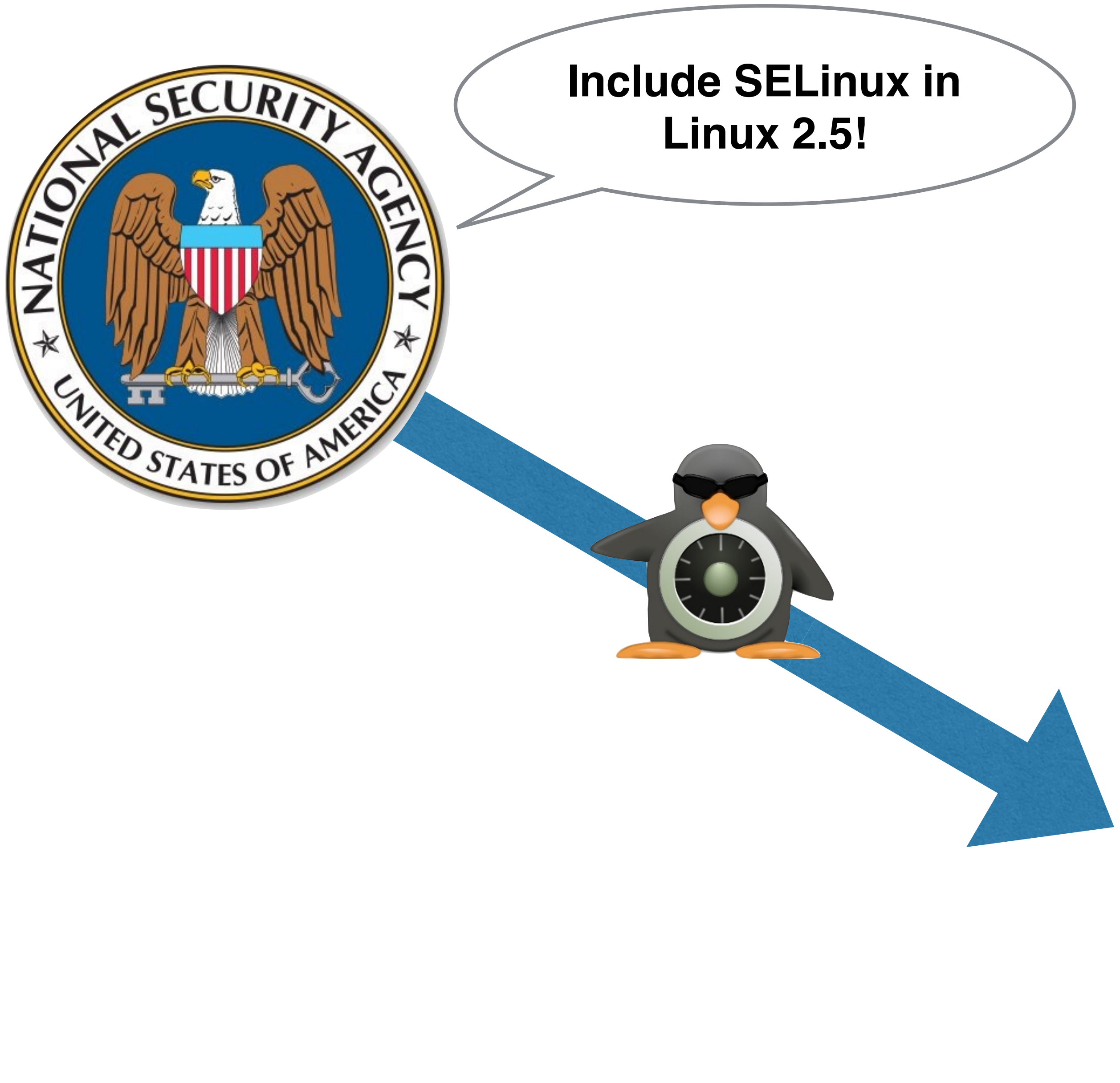
- Permissions in SELinux can be (at least partially) derived through runtime analysis.
- `audit2allow`:
  - **Step 1**: Run programs in a controlled (no attacker) environment without any enforcement.
  - **Step 2**: Audit all of the permissions used during normal operation.
  - **Step 3**: Generate policy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

SELinux Transition State

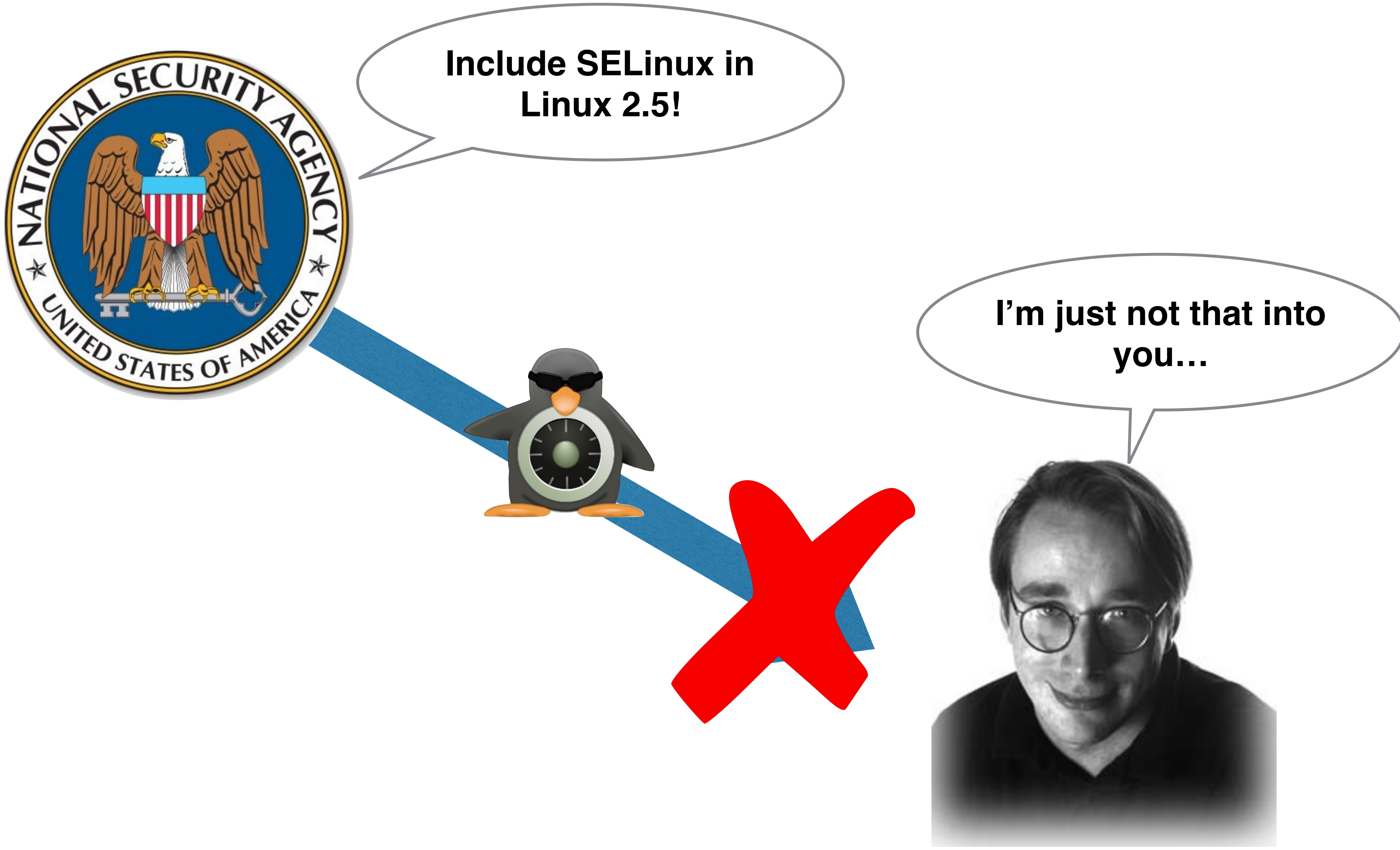
- Premise: Processes don't need to run in the same protection state all of the time.
- Borrows concepts from *Role-Based Access Control*
- Example: `passwd` starts in user context, transitions to privileged context to update /etc/passwd, transitions back to user.



@ 2001 Linux Kernel Summit…

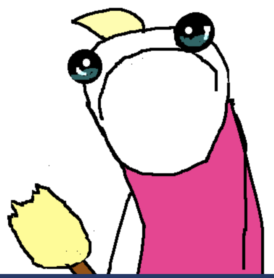


@ 2001 Linux Kernel Summit…

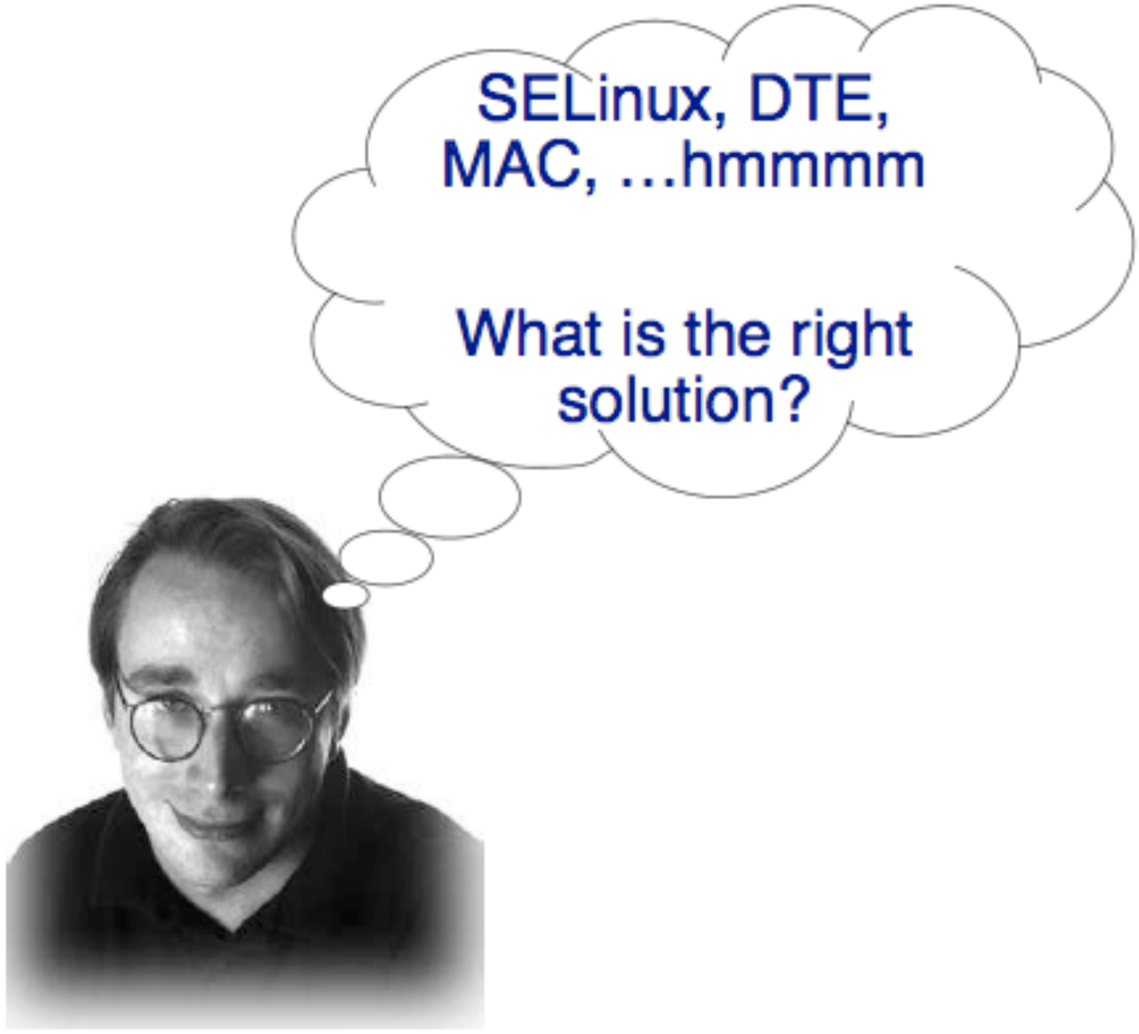


Linux Security circa 2000

- 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



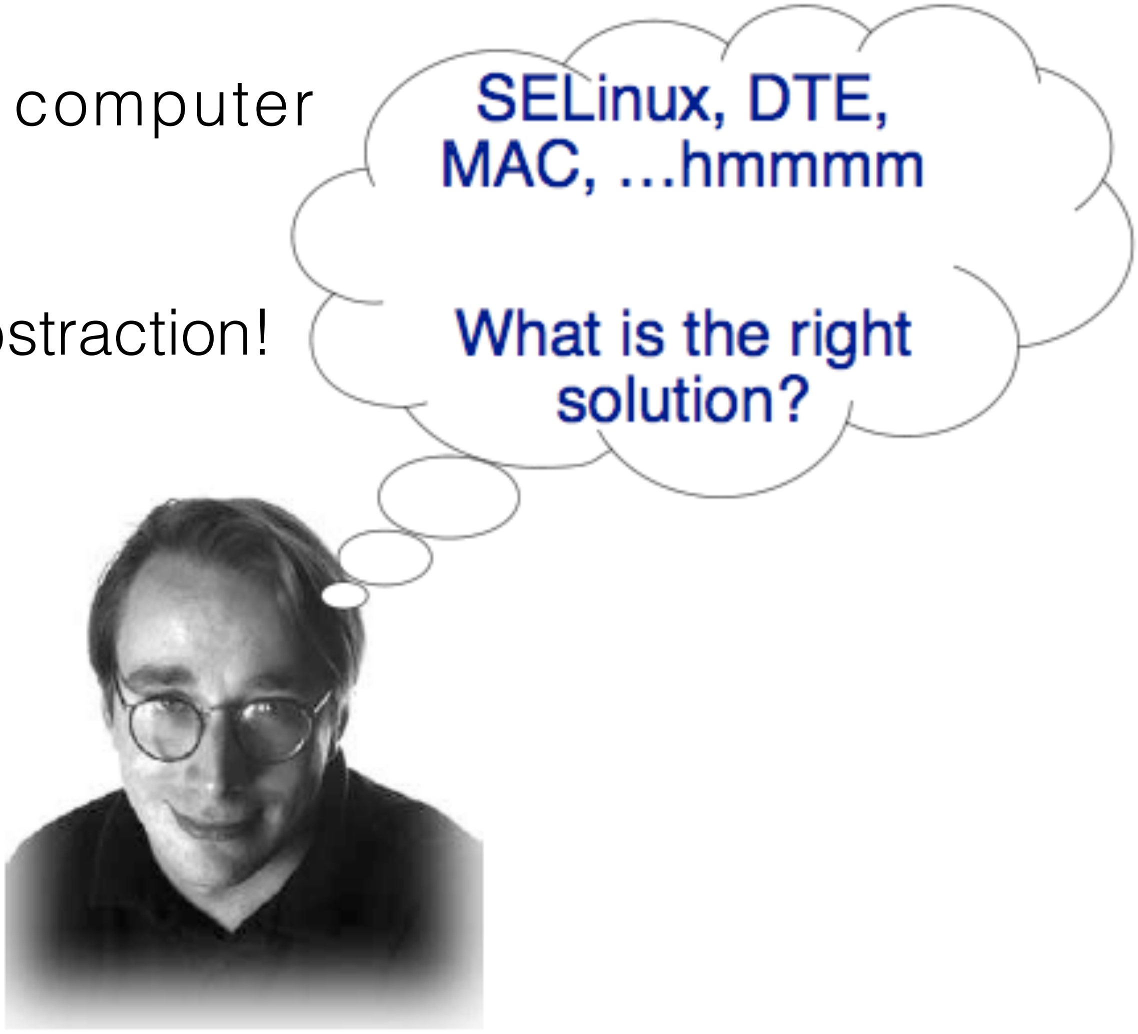
Linus's Dilemma



Linus's Dilemma

The answer to all computer science problems…

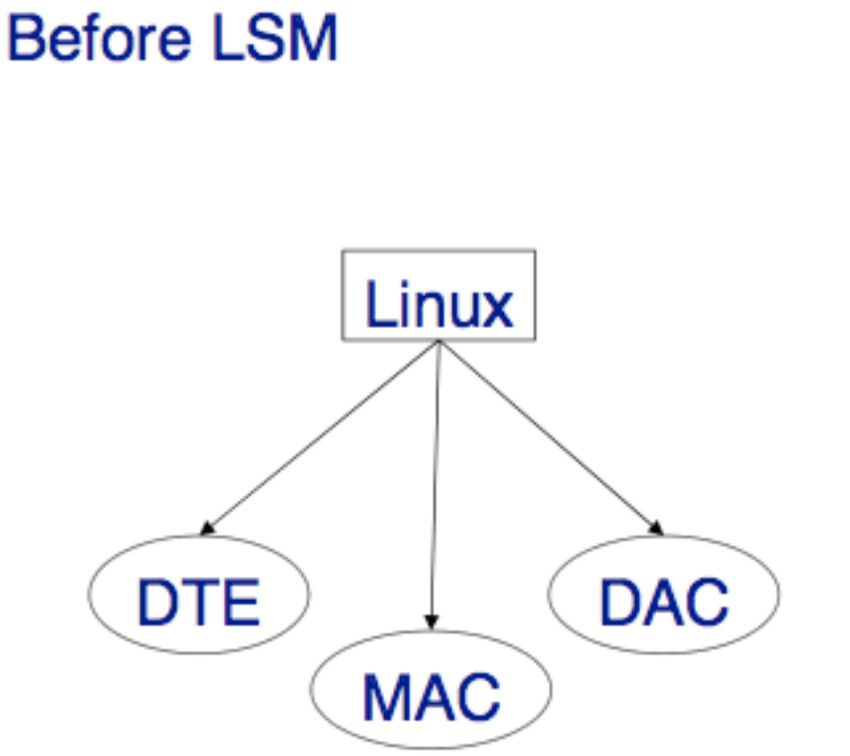
add another layer of abstraction!



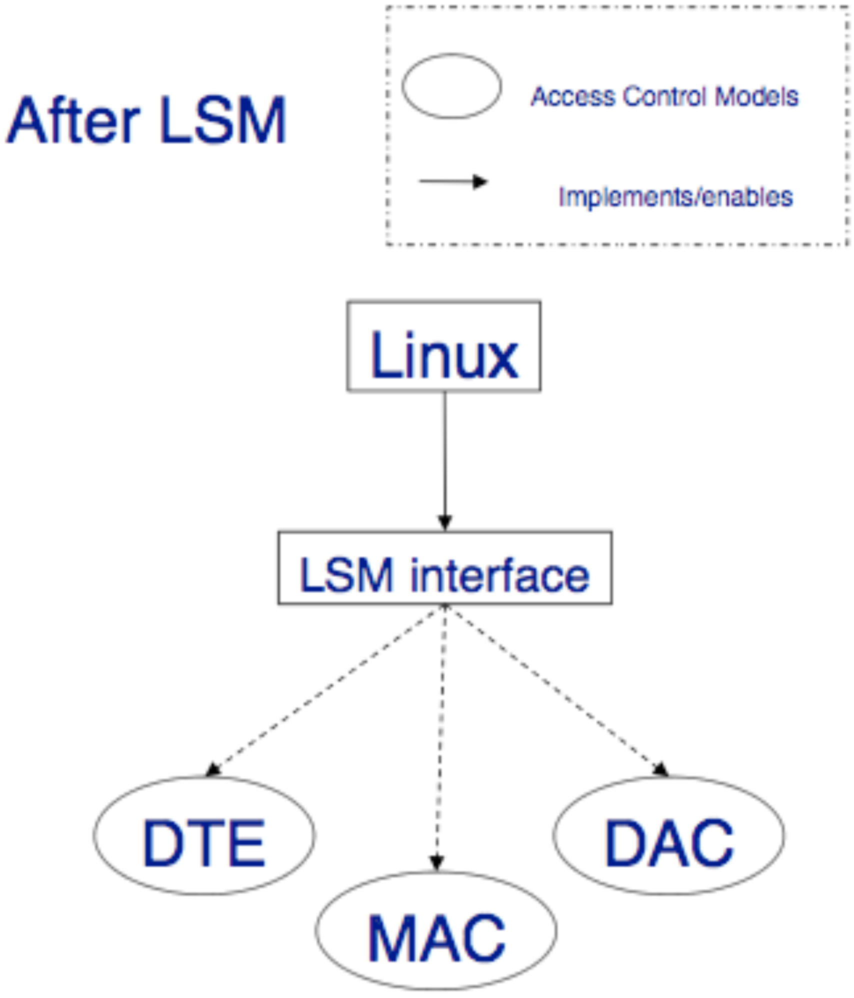
Linux Security Modules

- “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



Access control models implemented as Kernel patches



Access control models implemented as Loadable Kernel Modules

LSM Requirements

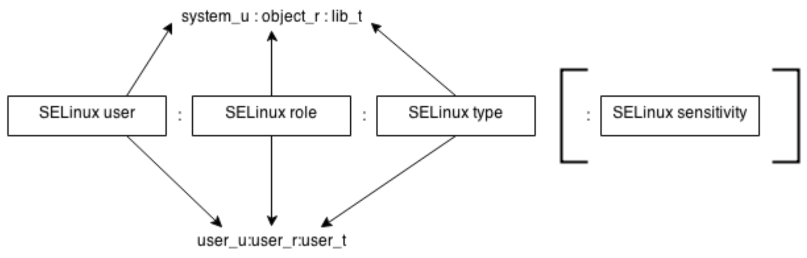
- LSM needs to reach a balance between kernel developer and security developers requirements. LSM needs to unify the functional needs of as many security projects as possible, while minimizing the impact on the Linux kernel.
  - Truly generic
  - conceptually simple
  - minimally invasive
  - Efficient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

LSM Architecture

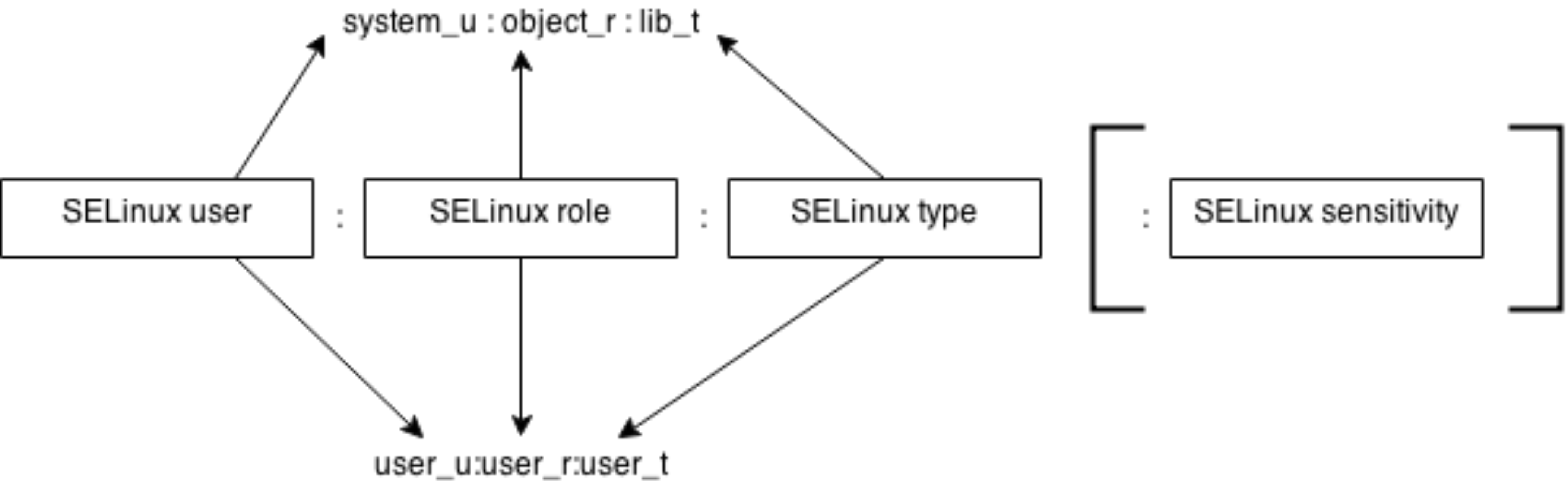
- Linux Kernel modified in 5 ways:
  - Opaque security fields added to certain kernel data structures $\dashrightarrow$ “controlled data types”
  - Security hook function calls inserted at various points with the kernel code $\dashrightarrow$ “security-sensitive operations”
  - A generic security system 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?



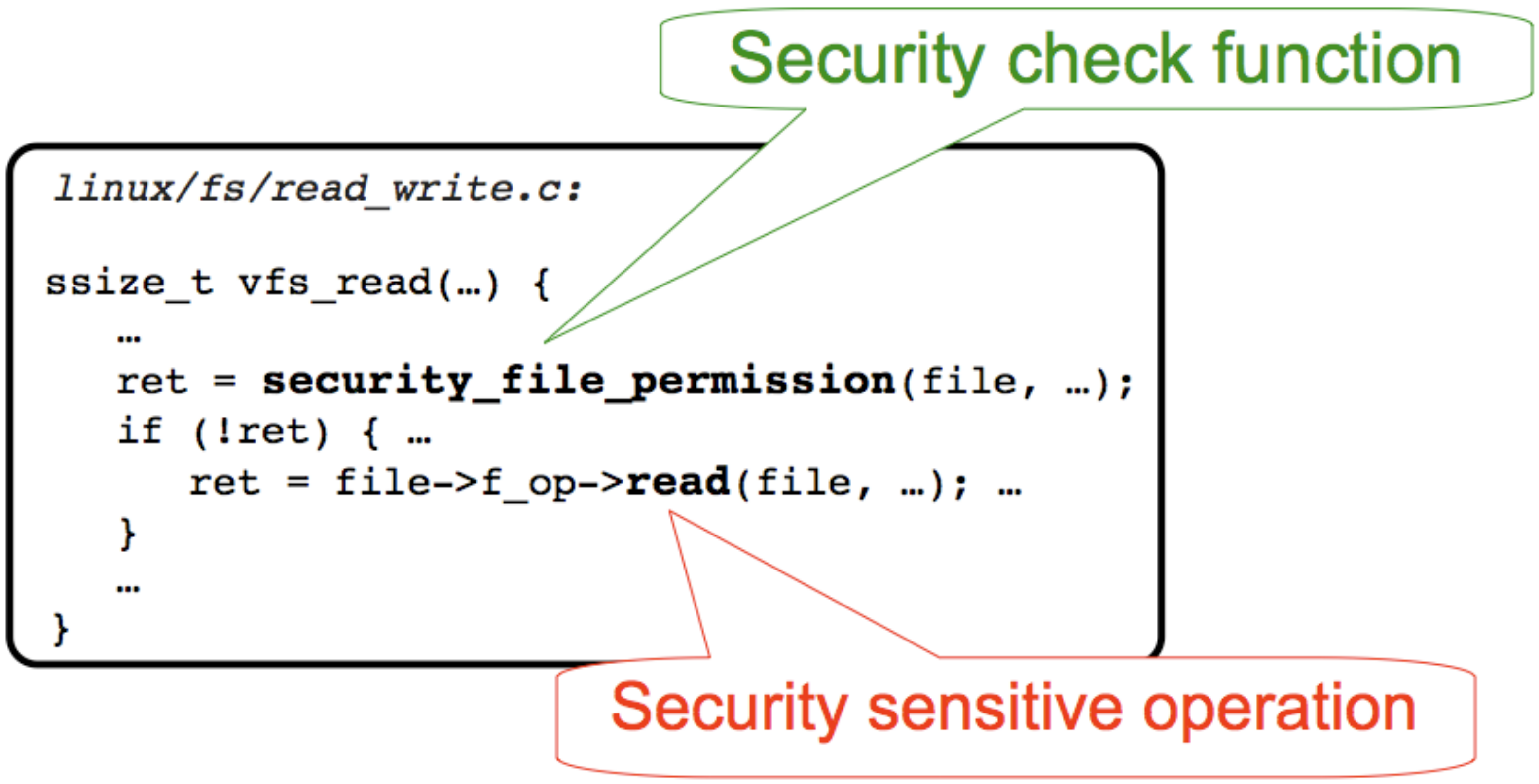
Opaque Security Fields



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



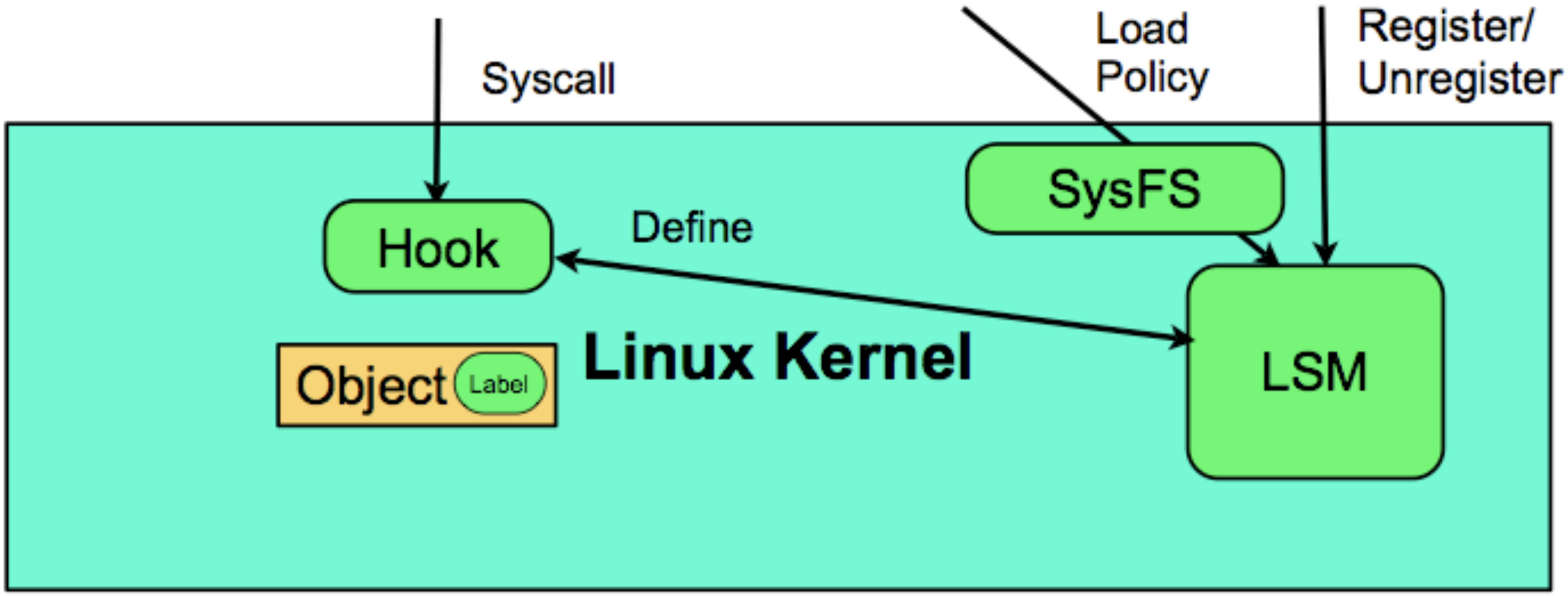
```
linux/fs/read_write.c:

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

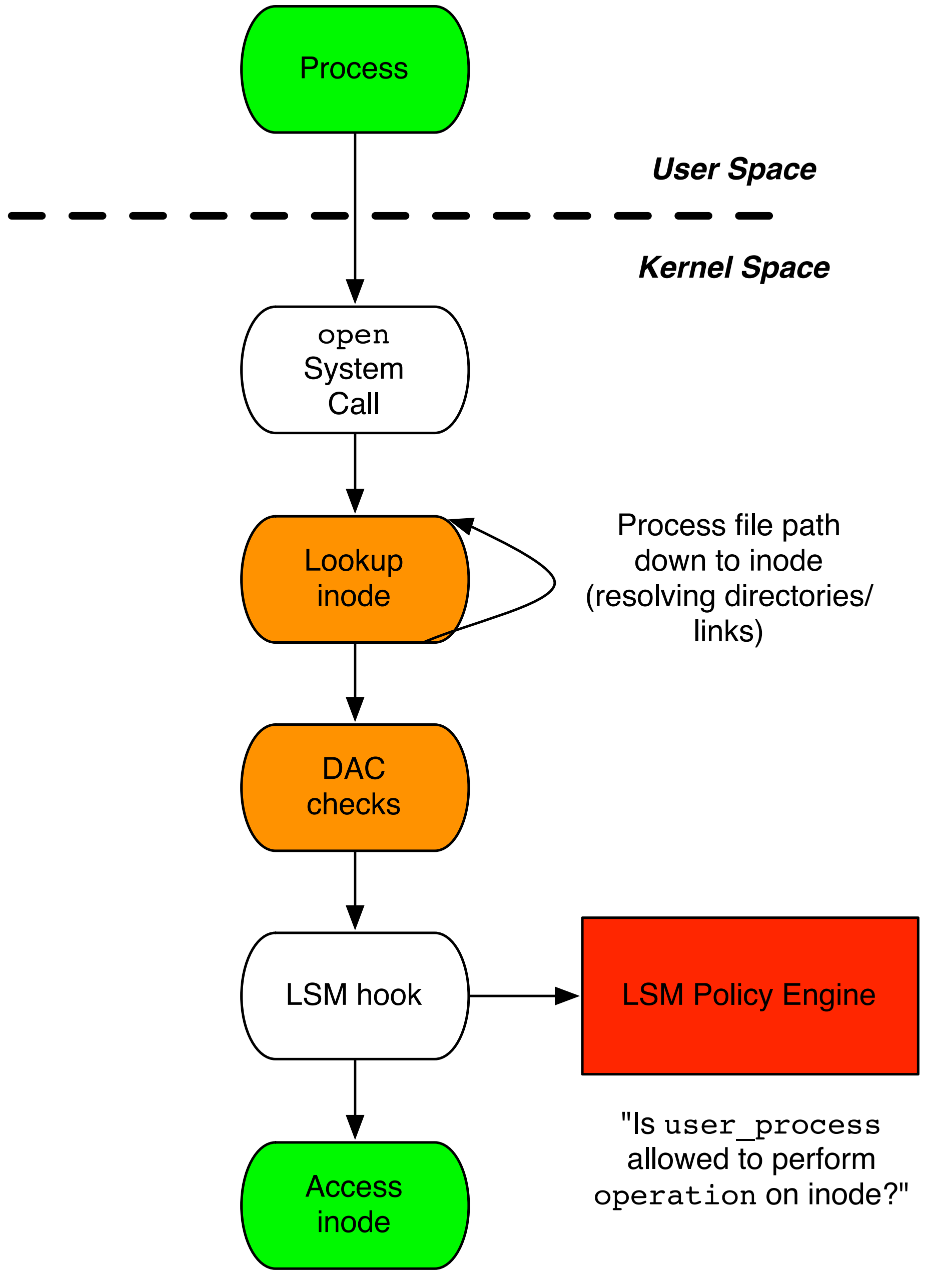
Security Hook Details

- 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


Process
User Space
Kernel Space
open System Call
Lookup inode
Process file path down to inode (resolving directories/ links)
DAC checks
LSM hook
LSM Policy Engine
"Is user_process allowed to perform operation on inode?"
Access inode

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

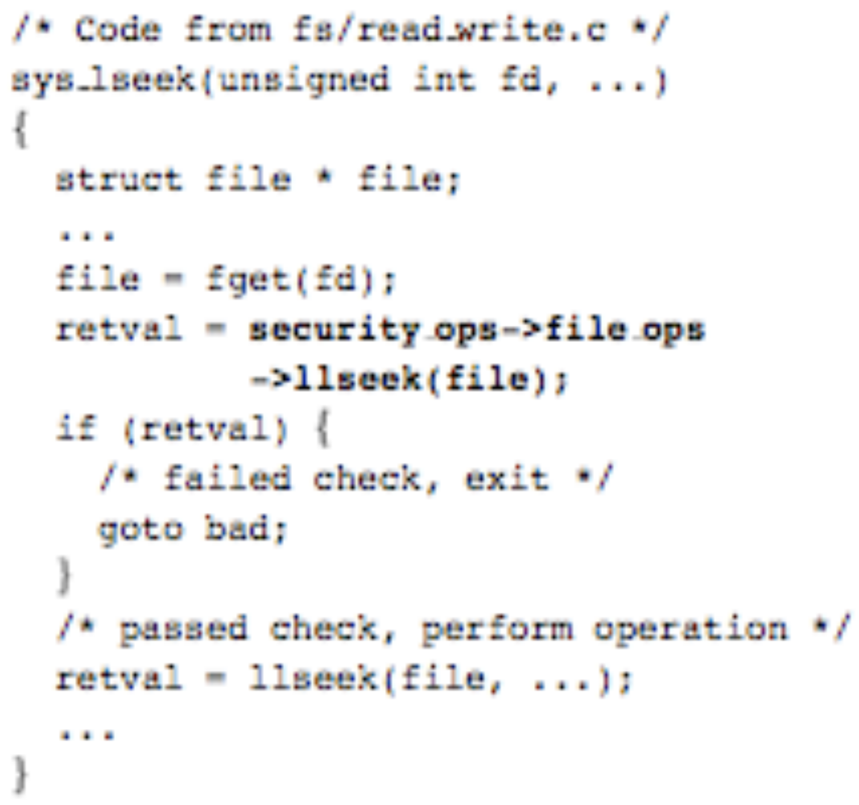
- Available in Linux 2.6
  - Packet-level controls upstreamed in 2.6.16
- Modules
  - POSIX Capabilities module
  - SELinux module
  - Domain and Type Enforcement
  - Openwall, includes grsecurity function
  - LIDS
  - AppArmor
- Not everyone is in favor
  - How does LSM impact system hardening?

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

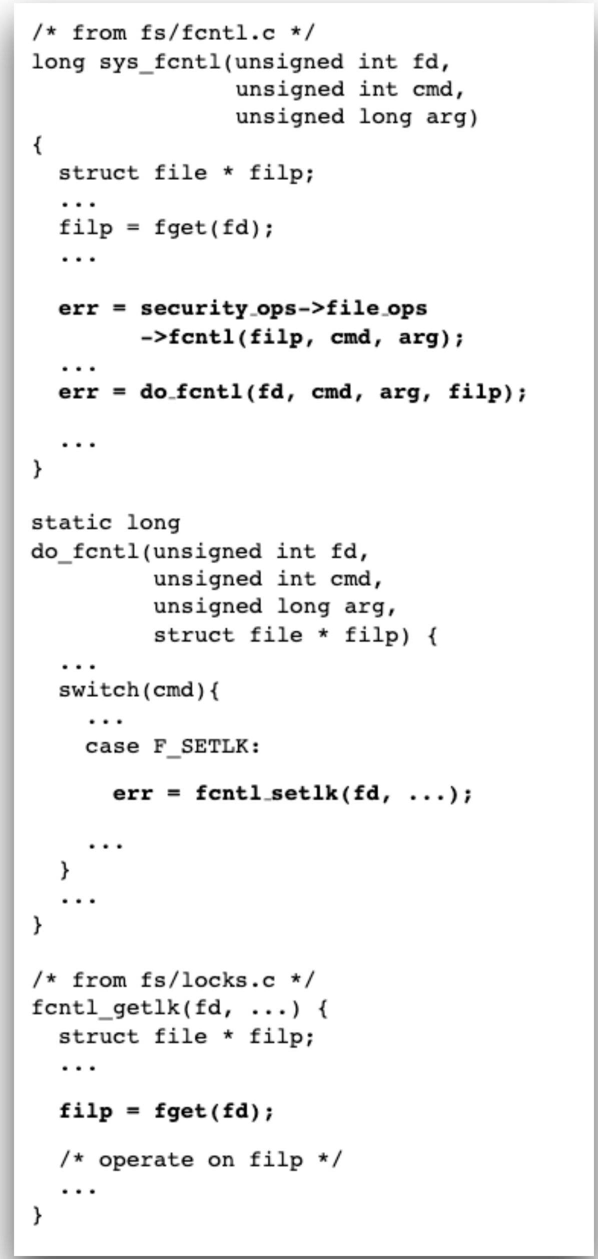
- Static analysis by Zhang, Edwards, and Jaeger [USENIX Security 2002]
  - Based on a tool called CQUAL
- Approach
  - Objects of particular types can be in two states
    - Checked, Unchecked
  - All objects in a "controlled operation" must be checked
    - Structure member access on objects



```
/* 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 */
  ...
}
```

LSM Analysis

- Runtime analysis of Edwards, Zhang, and Jaeger [ACM CCS 2002]
  - Built a runtime kernel monitor
  - Logs structure member accesses
  - Rules describe expected consistency
- Good for finding missing hooks where one is specified
  - Six cases were found

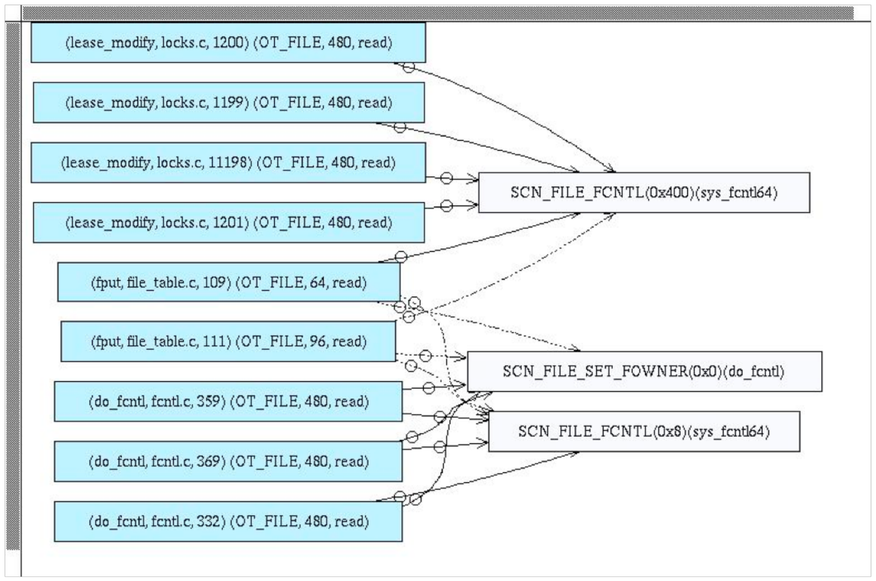
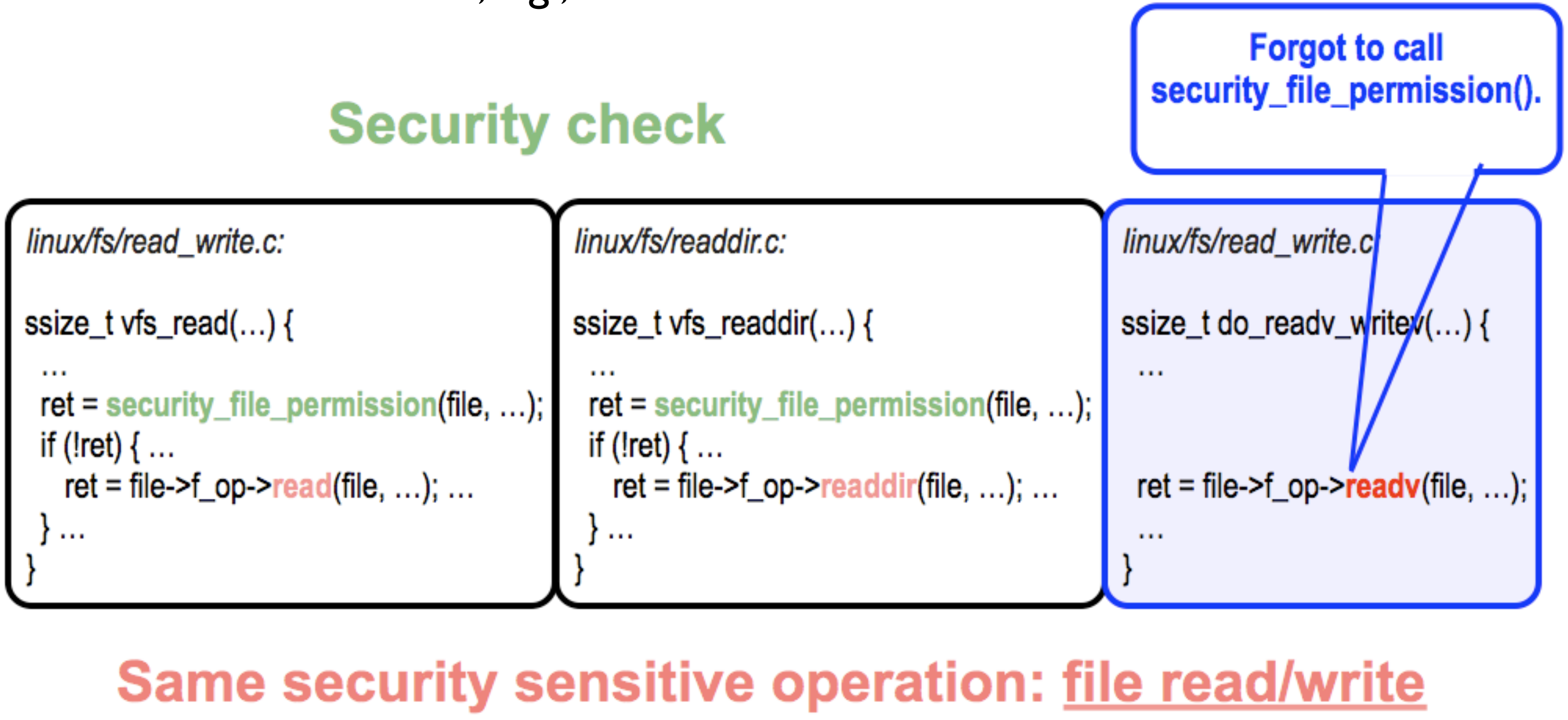

(lease_modify, locks.c, 1200) (OT_FILE, 480, read)
(lease_modify, locks.c, 1199) (OT_FILE, 480, read)
(lease_modify, locks.c, 11198) (OT_FILE, 480, read)
(lease_modify, locks.c, 1201) (OT_FILE, 480, read)
(fput, file_table.c, 109) (OT_FILE, 64, read)
(fput, file_table.c, 111) (OT_FILE, 96, read)
(do_fcntl, fcntl.c, 359) (OT_FILE, 480, read)
(do_fcntl, fcntl.c, 369) (OT_FILE, 480, read)
(do_fcntl, fcntl.c, 332) (OT_FILE, 480, read)
SCN_FILE_FCNTL(0x400)(sys_fcntl64)
SCN_FILE_SET_FOWNER(0x0)(do_fcntl)
SCN_FILE_FCNTL(0x8)(sys_fcntl64)

Figure 5: Authorization graph for `fcntl` calls for `F_SETLEASE` (controlled operations in `lease_modify` and `fput`) and `F_SETOWN` (controlled operations in `do_fcntl` and `put`). When command is `F_SETOWN` both `FCNTL` and `SET_OWNER` are authorized, but only `FCNTL` is authorized for `F_SETLEASE`.

- 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?


Security check
Forgot to call security_file_permission().
Same security sensitive operation: file read/write

```
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, ...);
  ...
}
```

Multics vs. LSM/SELinux

- LSM/SELinux is predominately software based; Multics designed for specialized hardware that could mediate access to memory segments beneath software layer.

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!!