Next edition of OS @ Brown

Fall 2026, Instructor: Malte Schwarzkopf (me)

If you enjoyed the material and want to help make the next version of the course, **apply to TA in Fall 2026**.

Either way, I want to hear from you:

1. *what is great about the course, and*
2. *what you'd change!*

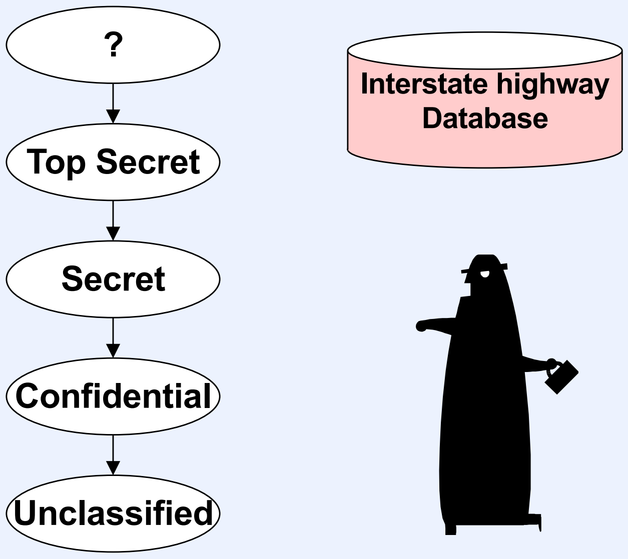
Reach out to meet or share your thoughts:

malte@brown.edu

Security Part 7

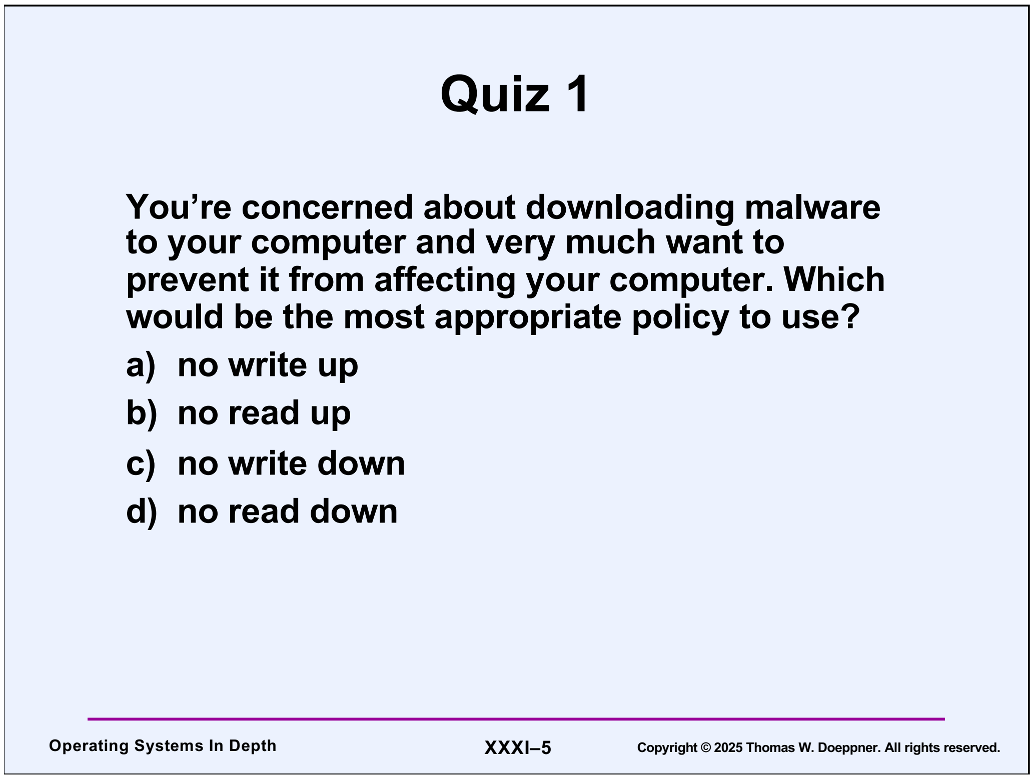
Live Anonymous Q&A:
https://tinyurl.com/cs1670feedback

Integrity


?
Top Secret
Secret
Confidential
Unclassified
Interstate highway Database

Biba Model

- Integrity is what's important
  - no-write-up
  - no-read-down



Quiz 1

You're concerned about downloading malware to your computer and very much want to prevent it from affecting your computer. Which would be the most appropriate policy to use?

a) no write up

b) no read up

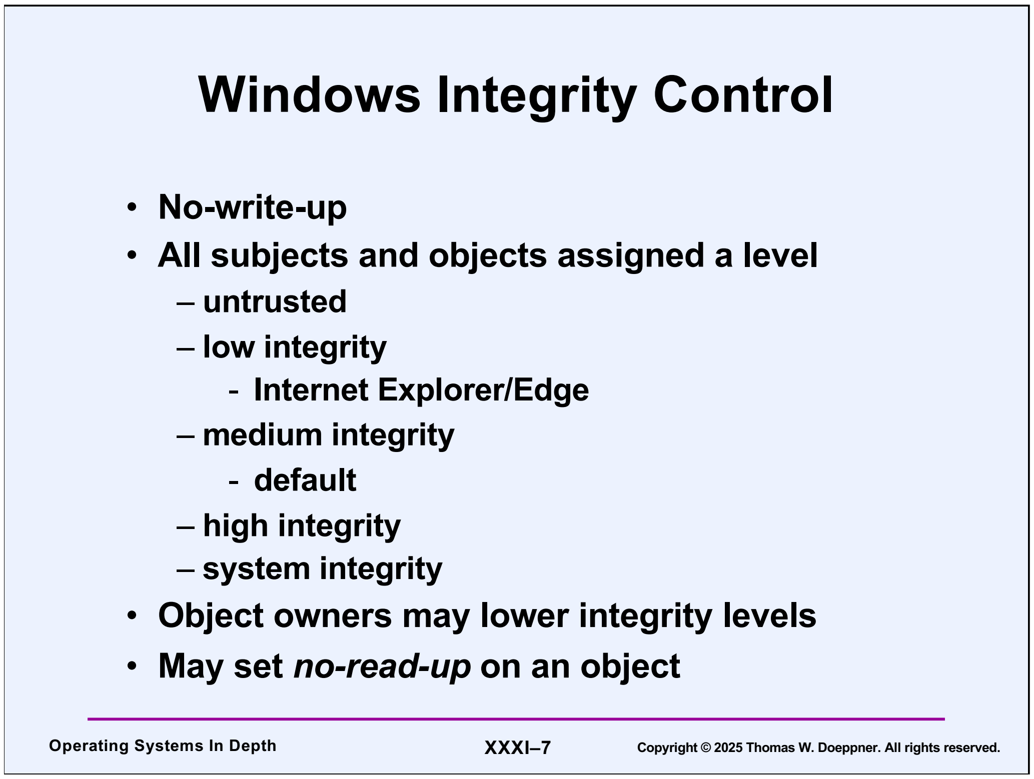
c) no write down

d) no read down

Hint: you don't want code downloaded by your web browser to replace trusted code on your computer.

Windows and MAC

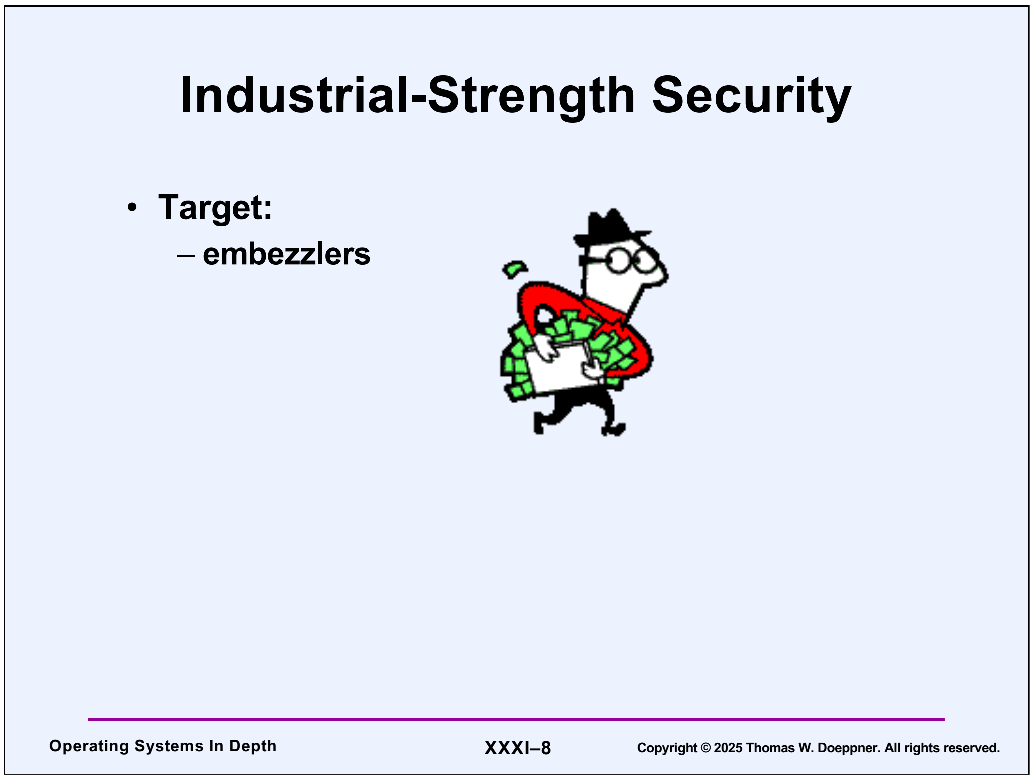
- Concerns
  - viruses
  - spyware
  - etc.
- Installation is an integrity concern
- Solution
  - adapt Biba model



Windows Integrity Control

- No-write-up
- All subjects and objects assigned a level
  - untrusted
  - low integrity
    - Internet Explorer/Edge
  - medium integrity
    - default
  - high integrity
  - system integrity
- Object owners may lower integrity levels
- May set *no-read-up* on an object

The integrity level of an object is stored in its SACL (system access control list).

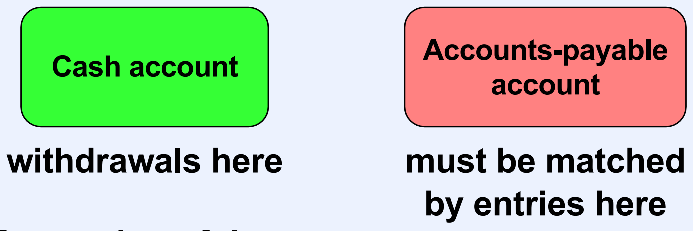


Industrial-Strength Security

- Target:
  - embezzlers

Clark-Wilson Model

- Integrity and confidentiality aren't enough
  - there must be control over how data is produced and modified
    - well formed transactions

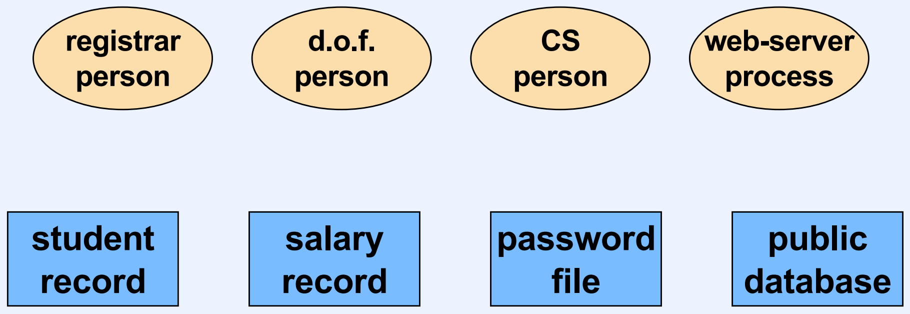


- Separation of duty
  - steps of transaction must involve multiple people

Mandatory Access Control (MAC)

Implementing MAC

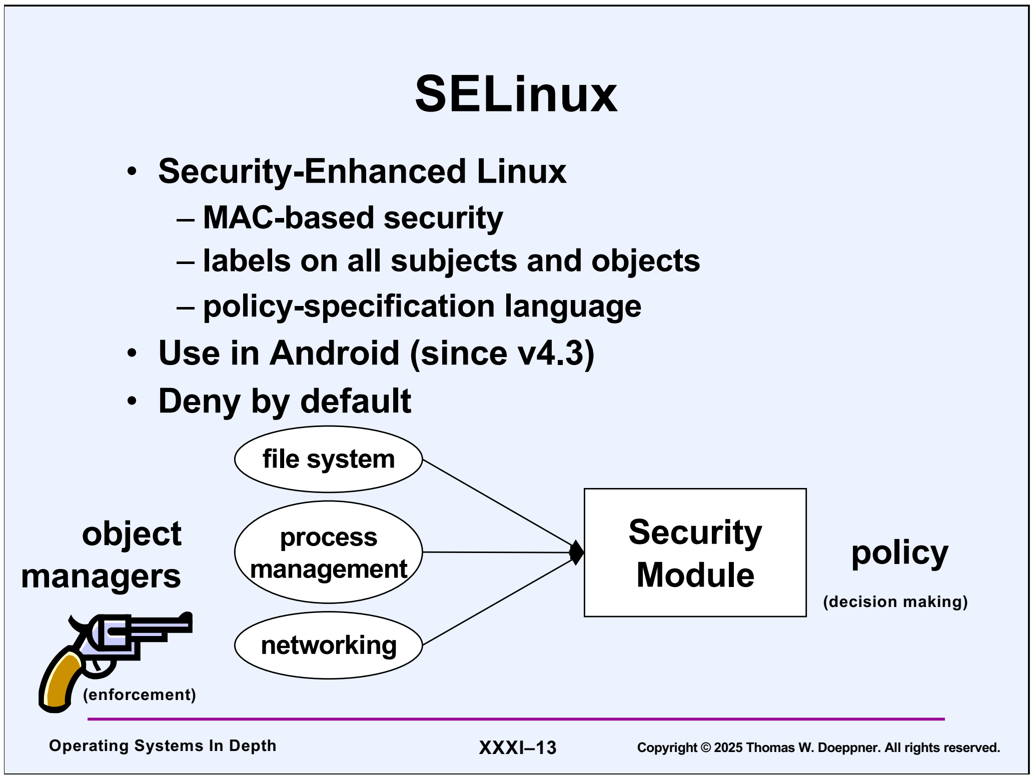
- Label subjects and objects
- Security policy makes decisions based on labels and context



Quiz 2

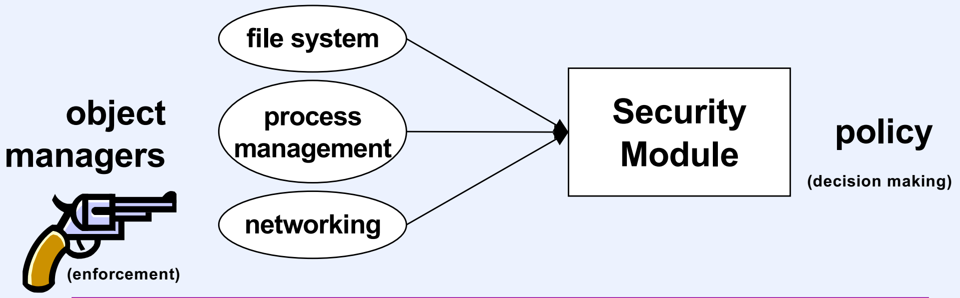
I have a file that I accidentally set as having rw permission for everyone (0666). You have a process that has opened my file rw. I discover this and immediately change the permissions to 0600 (access only by me). Can your process still read and write the file?

a) It can read and write
b) It can read, but not write
c) It can write, but not read
d) It can do neither



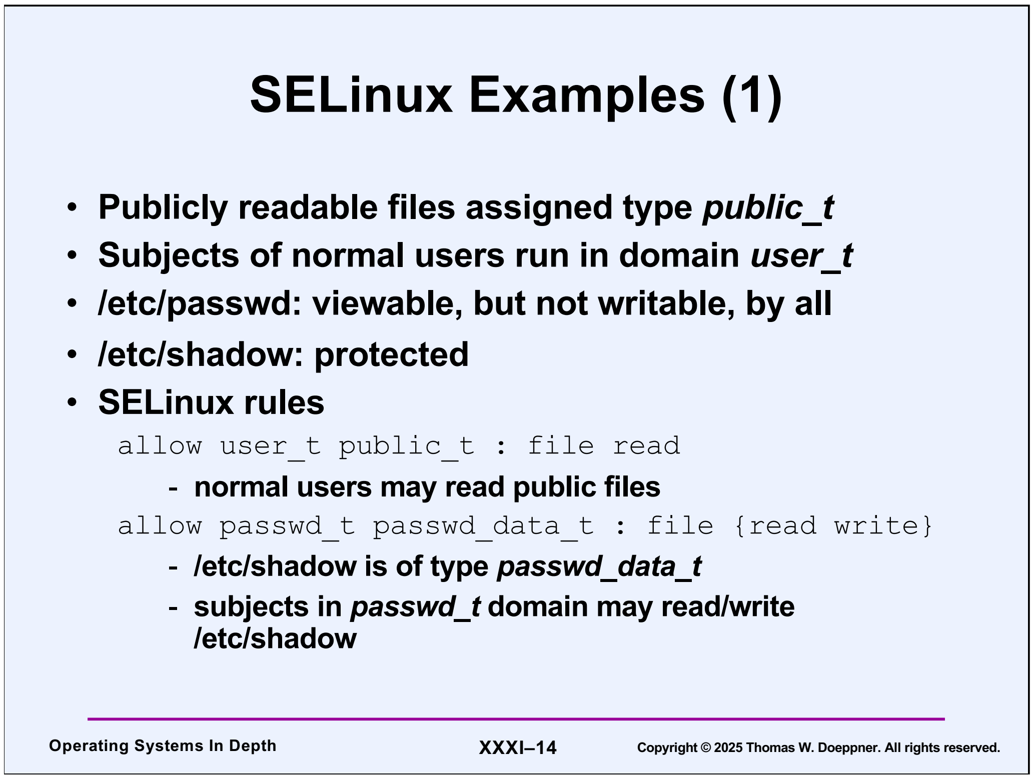
SELinux

- Security-Enhanced Linux
  - MAC-based security
  - labels on all subjects and objects
  - policy-specification language
- Use in Android (since v4.3)
- Deny by default



A description of SELinux is given in the paper “Integrating Flexible Support for Security Policies into the Linux Operating System” by Peter Loscocco and Stephen Smalley in the Proceedings of the 2001 USENIX Annual Technical Conference (FREENIX ’01), June 2001.

In SELinux, access permissions are checked not just when a file is opened, but on every access to the file.



SELinux Examples (1)

- Publicly readable files assigned type *public_t*
- Subjects of normal users run in domain *user_t*
- /etc/passwd: viewable, but not writable, by all
- /etc/shadow: protected
- SELinux rules

```
allow user_t public_t : file read
```

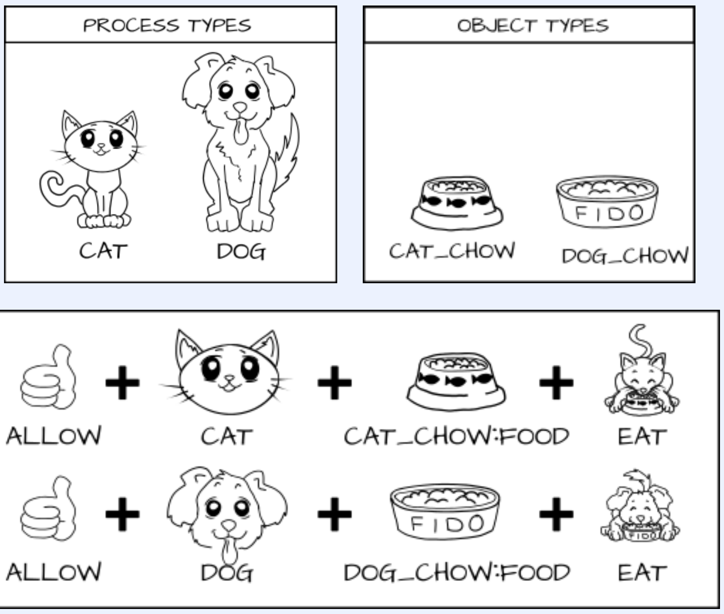
  - normal users may read public files

```
allow passwd_t passwd_data_t : file {read write}
```

  - /etc/shadow is of type *passwd_data_t*
  - subjects in *passwd_t* domain may read/write /etc/shadow

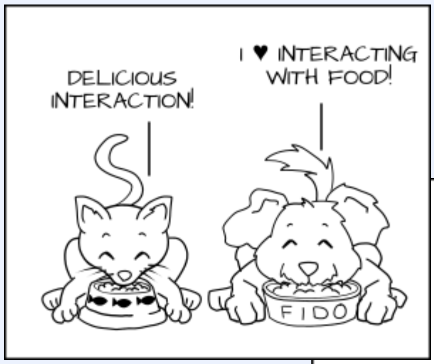
This example is covered in the textbook, starting on page 338.

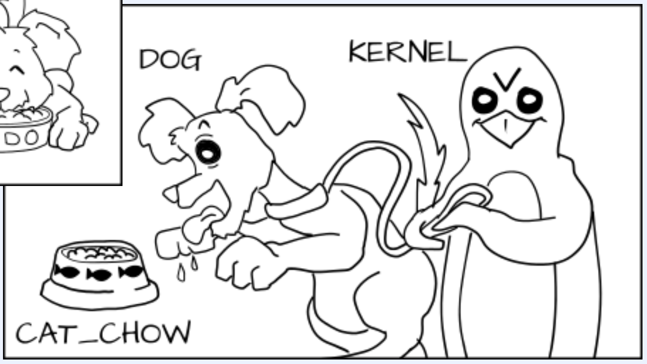
Note that, unless there are additional allow statements for **passwd_data_t**, only subjects in the **passwd_t** domain may access /etc/shadow.


PROCESS TYPES
CAT
DOG
OBJECT TYPES
FIDO
CAT_CHOW
DOG_CHOW
ALLOW + CAT + CAT_CHOW:FOOD + EAT
ALLOW + DOG + DOG_CHOW:FOOD + EAT
FIDO

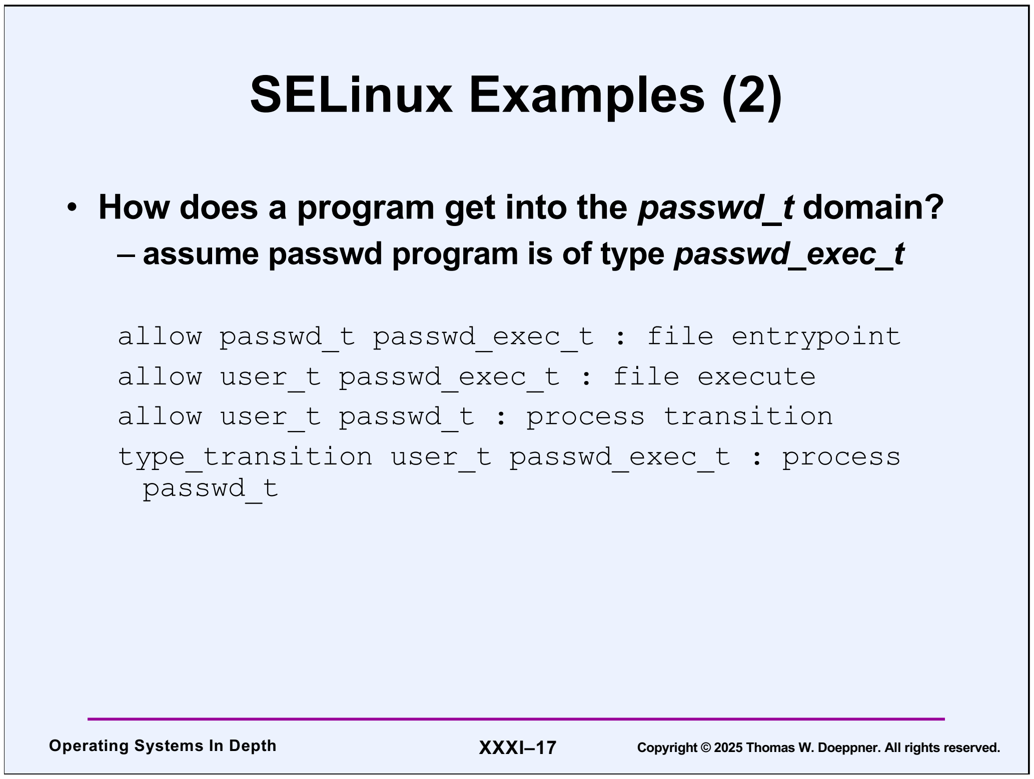
Cartoon credit: https://opensource.com/business/13/11/selinux-policy-guide

Result





Cartoon credit: https://opensource.com/business/13/11/selinux-policy-guide



SELinux Examples (2)

- **How does a program get into the *passwd_t* domain?**
  - **assume passwd program is of type *passwd_exec_t***

```
allow passwd_t passwd_exec_t : file entrypoint
allow user_t passwd_exec_t : file execute
allow user_t passwd_t : process transition
type_transition user_t passwd_exec_t : process
  passwd_t
```

The first allow statement says that the domain **passwd_t** can be entered if a program of type **passwd_exec_t** is being run.

The second allow statement says that subjects in the **user_t** domain may execute programs of type **passwd_exec_t**.

The third allow statement says that it's permissible for subjects in the **user_t** domain to transition to the **passwd_t** domain.

The last statement says that when a subject in the **user_t** domain executes a program of type **passwd_exec_t**, it should attempt to switch to the the **passwd_t** domain, assuming it's permitted to do so (which it will be given the prior three statements).

Quiz 3

We’ve seen how the setuid feature in Unix is used to allow normal users to change their passwords in /etc/shadow.

a) This approach actually isn’t secure, which is among the reasons why SELinux exists

b) The approach is secure and thus SELinux doesn’t really add any additional protection to /etc/shadow

c) The approach is secure but there are other potential /etc/shadow-related vulnerabilities that SELinux helps deal with

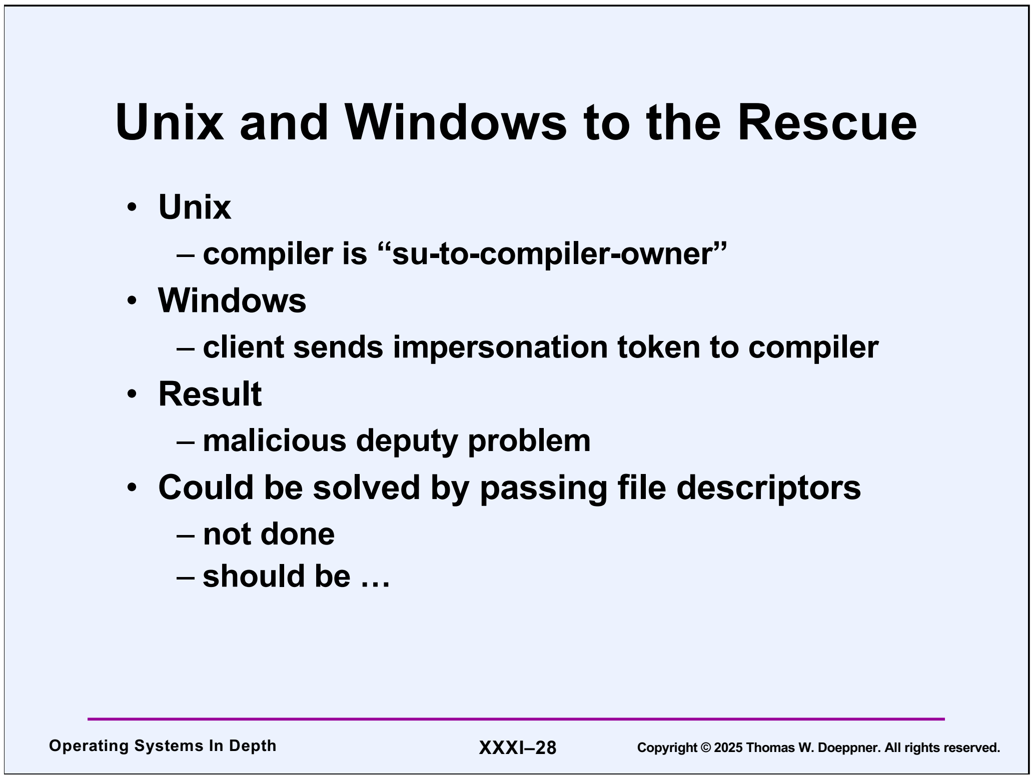
Off-the-Shelf SELinux

- Strict policy
  - normal users in *user_r* role
  - users allowed to be administrators are in *staff_r* role
    - but may run admin commands only when in *sysadm_r* role
  - policy requires > 20,000 rules
  - tough to live with
- Targeted policy
  - targets only “network-facing” applications
  - everything else in *unconfined_t* domain
  - ~11,000 rules

Capability-Based Systems

Confused-Deputy Problem

- The system has a pay-per-use compiler
  - keeps billing records in file /u/sys/comp/usage
  - puts output in file you provide
    - /u/you/comp.out
- The concept of a pay-per-use compiler annoys you
  - you send it a program to compile
  - you tell it to put your output in /u/sys/comp/usage
  - it does
    - it's confused
    - you win



Unix and Windows to the Rescue

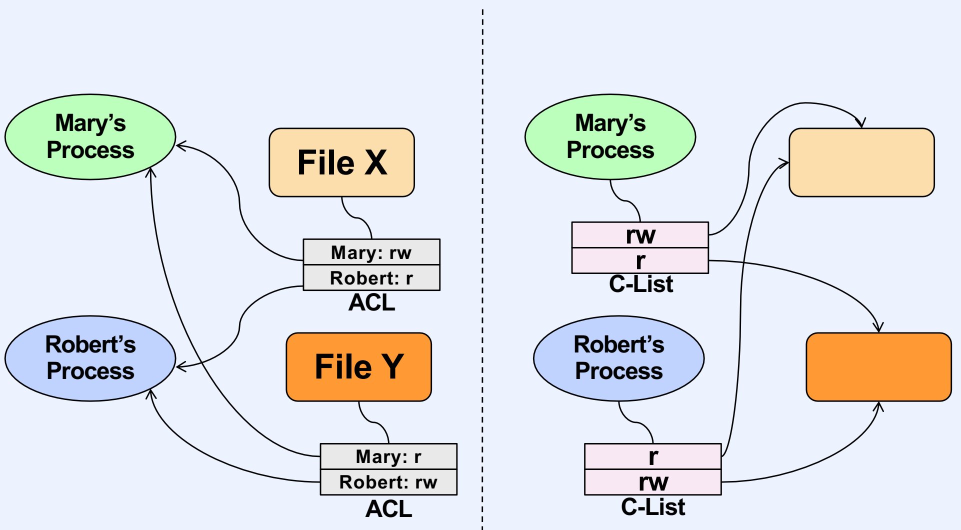
- Unix
  - compiler is “su-to-compiler-owner”
- Windows
  - client sends impersonation token to compiler
- Result
  - malicious deputy problem
- Could be solved by passing file descriptors
  - not done
  - should be …

The "malicious deputy" can access all your files.

Authority

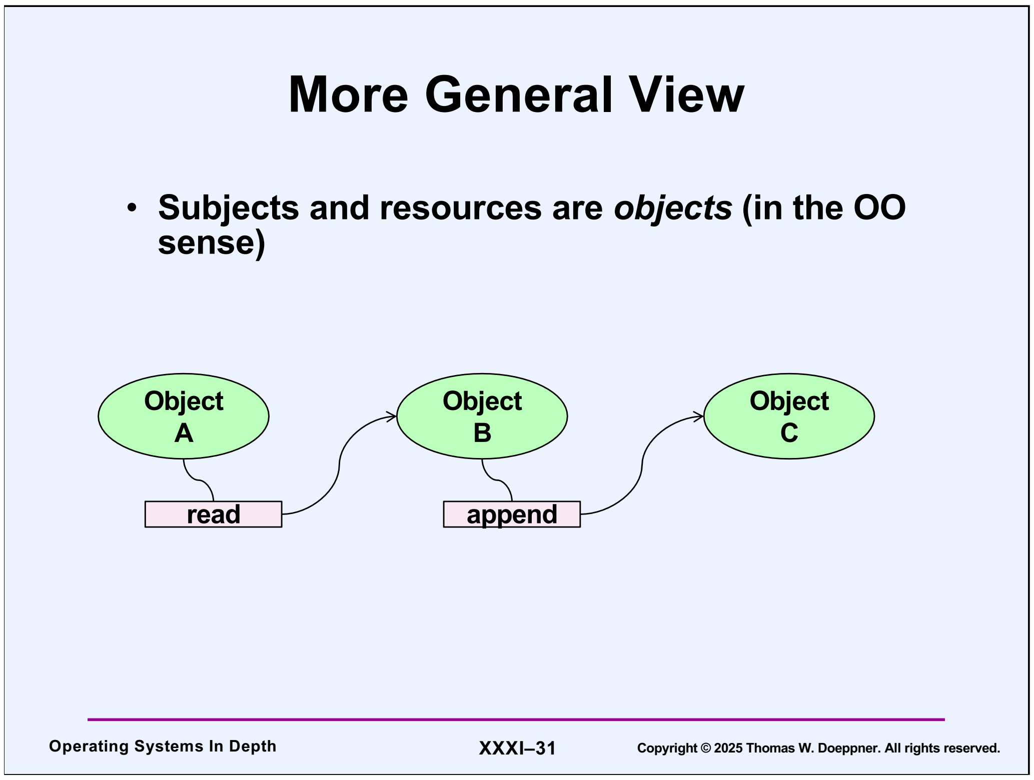
- Pure ACL-based systems
  - authority depends on subject’s user and group identities
- Pure capability-based systems
  - authority depends upon capabilities possessed by subject

ACLs vs. C-Lists



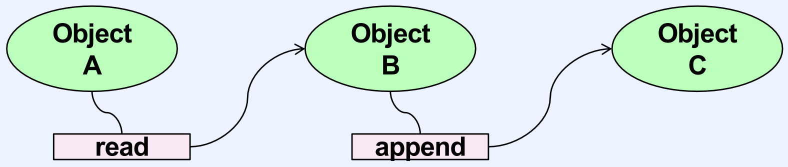
A **capability** is both a reference to a resource and an access right to that resource. Furthermore, it's unforgeable. The set of capabilities possessed by a subject is called its **C-list**. Note that with capabilities, it's not necessary for resources to have names—the capability suffices. Note that the ACLs refer to any process whose user ID is Mary or Robert, whereas a C-list merely indicates that a particular process has a capability for the indicated resource.

Note that, to avoid confusion with the objects of object-oriented programming, we're using the term "resource" in place of "object."

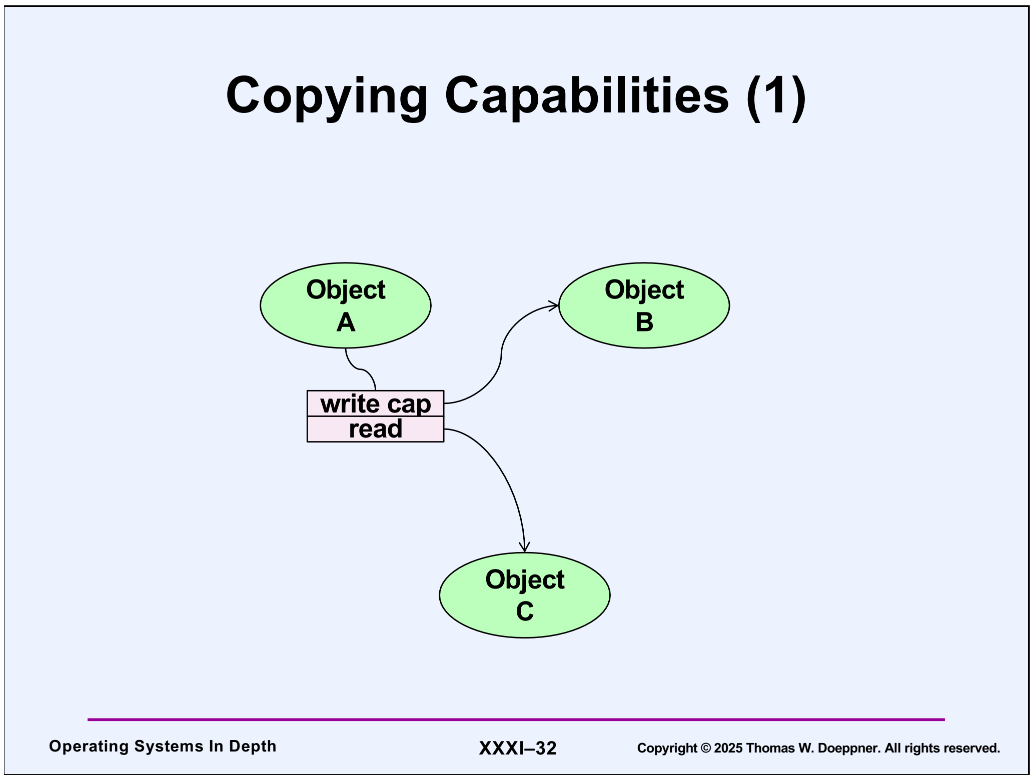


More General View

- Subjects and resources are *objects* (in the OO sense)

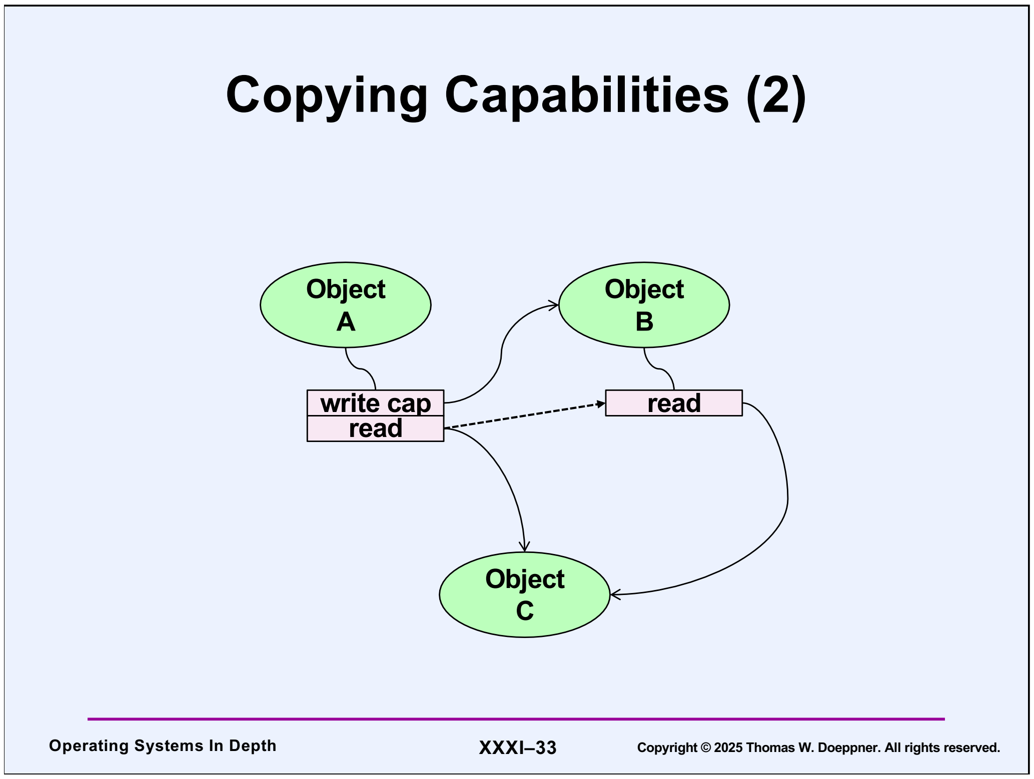


Let's think of both subjects and resources as being general objects (in the object-oriented-programming sense). A capability is a reference to an object that allows the bearer to invoke the indicated operation.

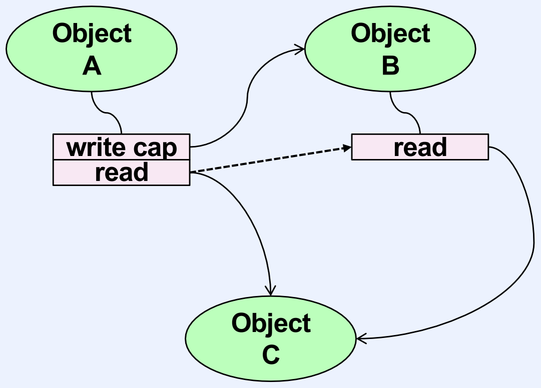


Copying Capabilities (1)

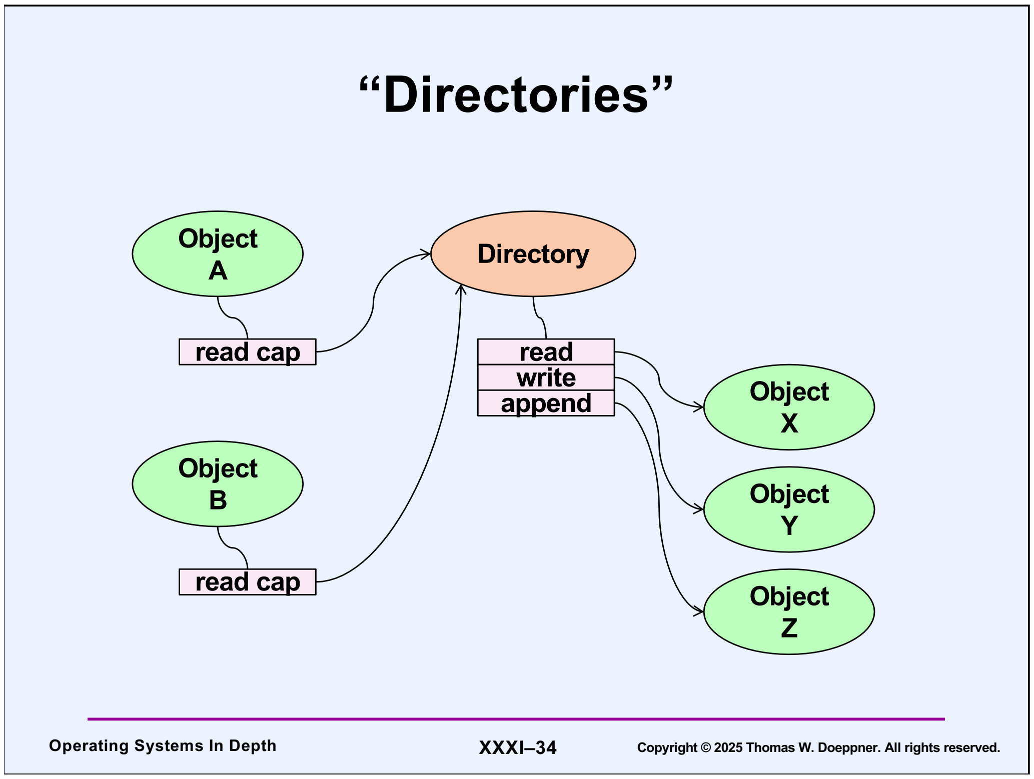
Object A has a “write-capability” capability for object B, which allows A to copy capabilities to B.



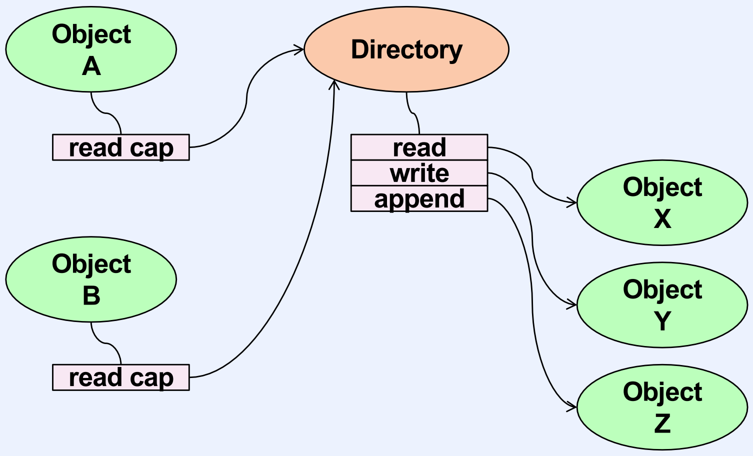
Copying Capabilities (2)



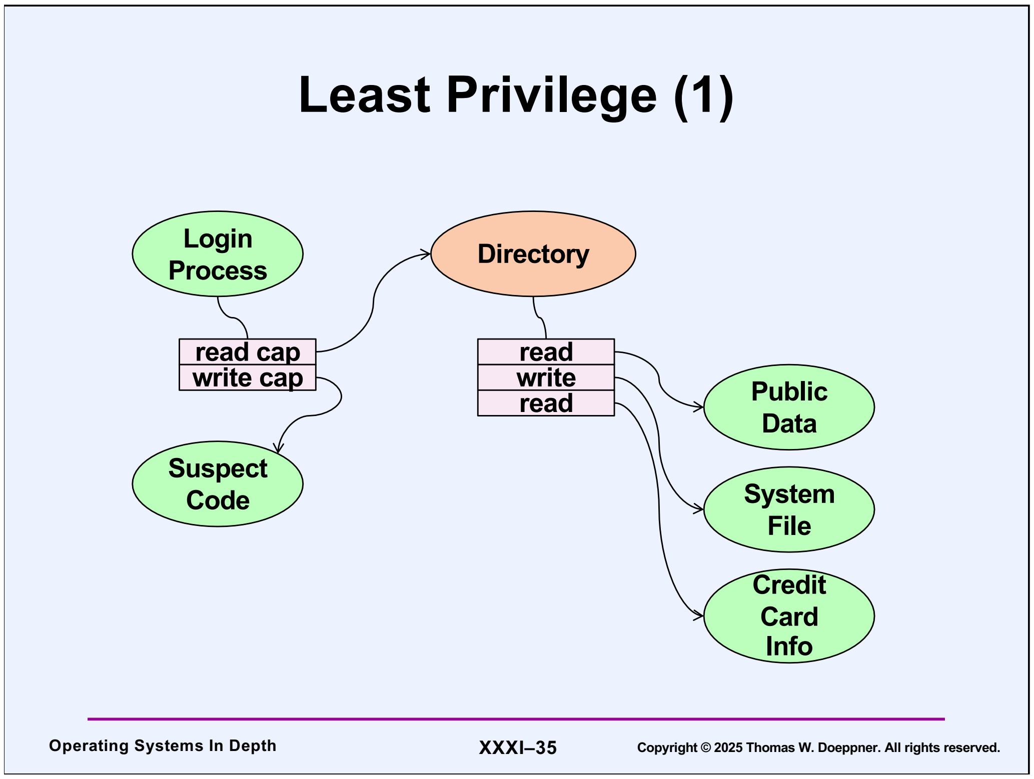
Object A has copied its “read capability for object C” to object B.



“Directories”

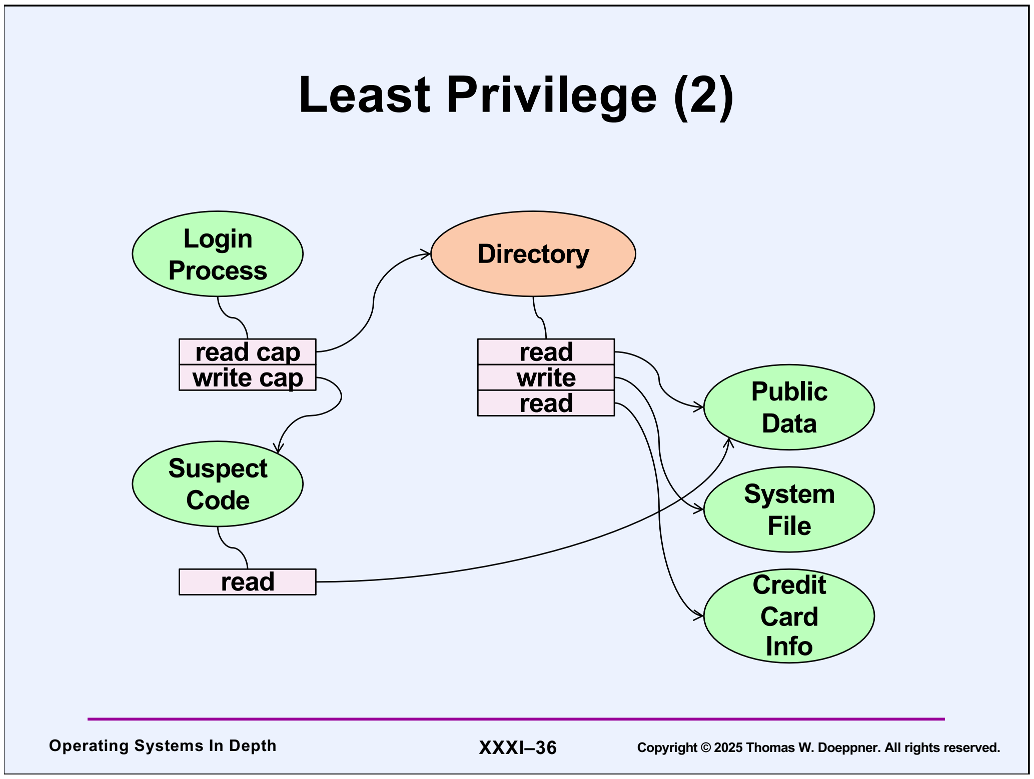


Here we have a “directory object” that provides capabilities to other objects. Object A may use its read-capability capability to fetch capabilities from the directory object.



Least Privilege (1)

Here we want to run a program that we've recently downloaded from the web. We create a process in which to run it, giving our login process a write-capability capability to it.



Least Privilege (2)

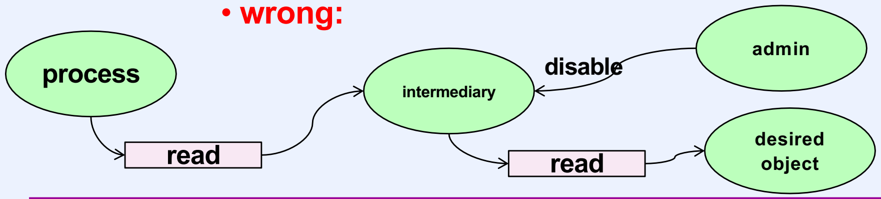
We give the new process a read capability for some public data, but no capability for anything else (and particularly no capability for getting other capabilities from the directory).

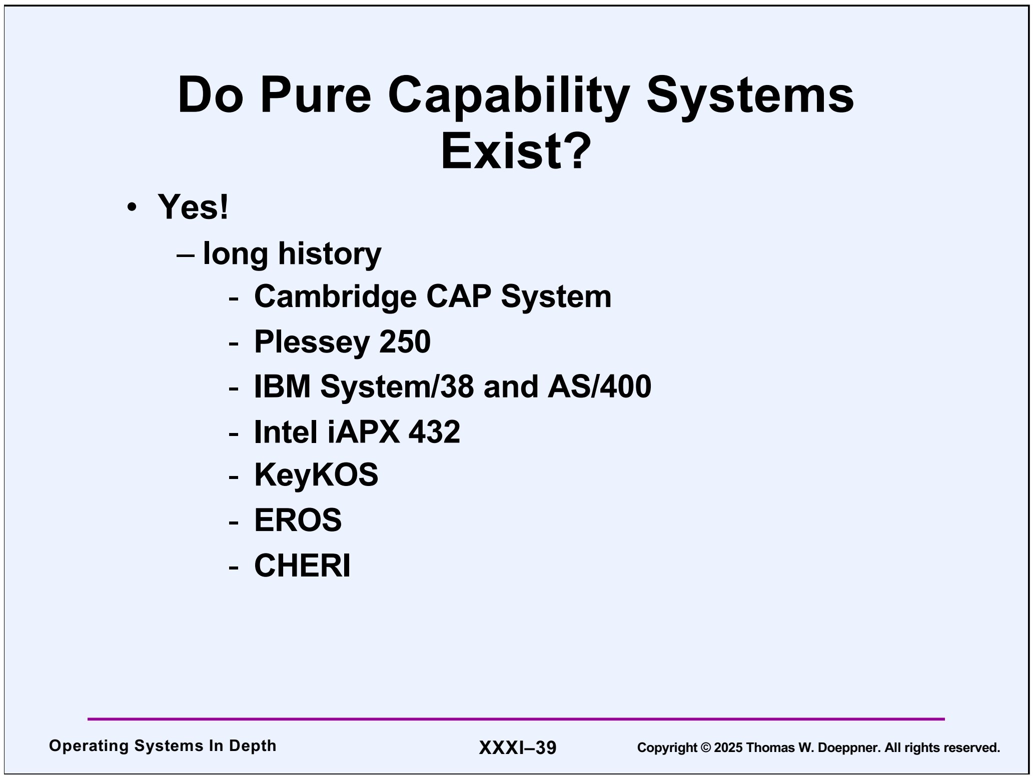
Issues

- Files aren't referenced by names. How do your processes get capabilities in the first place?
  – your "account" is your login process
    - created with all capabilities it needs
    - persistent: survives log-offs and crashes

Issues

- Can MAC be implemented on a pure capability system?
  - proven impossible twice
    - capabilities can be transferred to anyone
      - wrong: doesn't account for write-capability and read-capability capabilities
    - capabilities can't be retracted once granted
      - wrong:

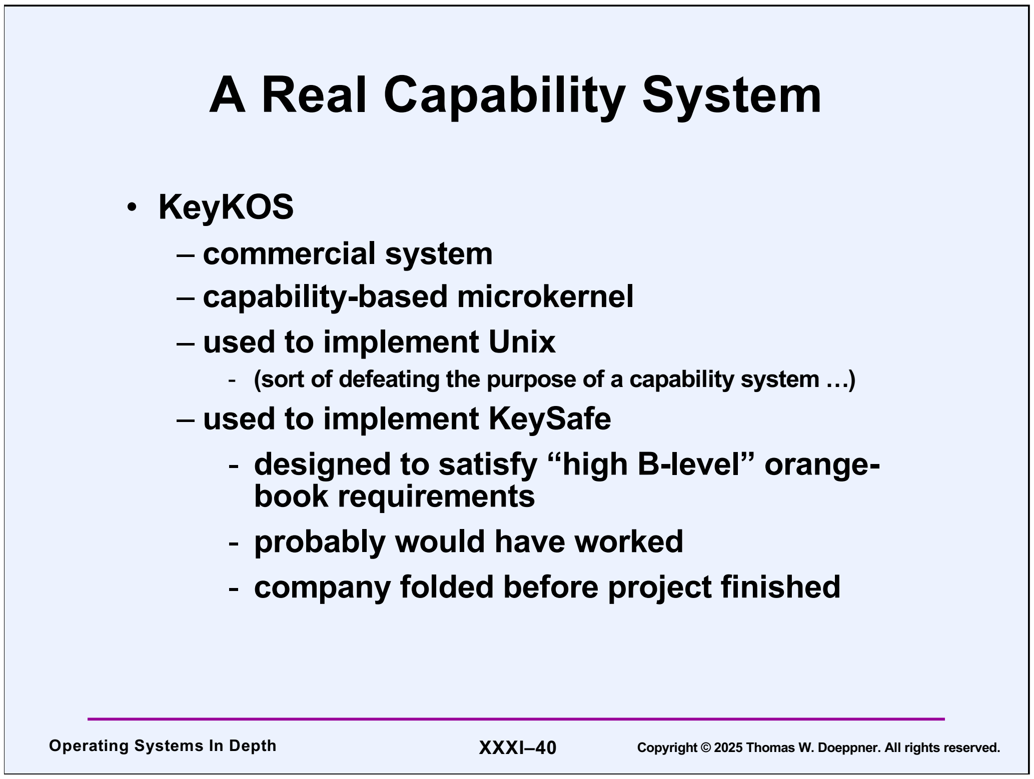




Do Pure Capability Systems Exist?

- Yes!
  - long history
    - Cambridge CAP System
    - Plessey 250
    - IBM System/38 and AS/400
    - Intel iAPX 432
    - KeyKOS
    - EROS
    - CHERI

Note that this long history doesn't have any recent examples.

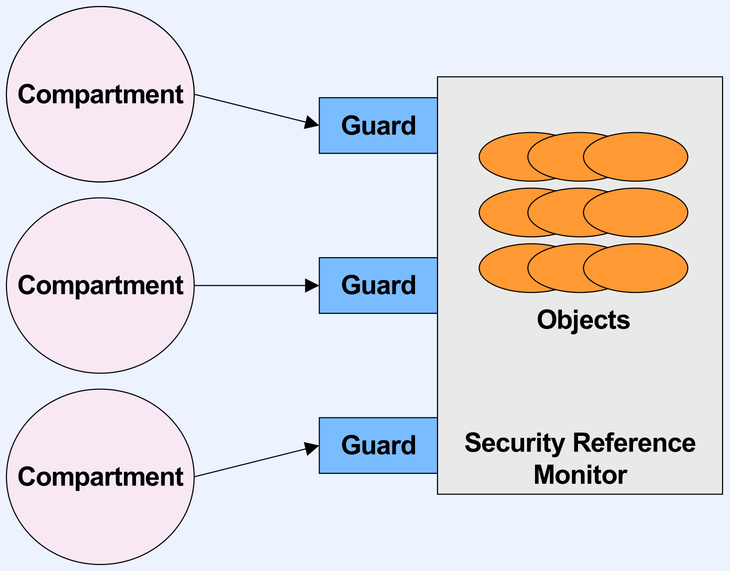


A Real Capability System

- KeyKOS
  - commercial system
  - capability-based microkernel
  - used to implement Unix
    - (sort of defeating the purpose of a capability system …)
  - used to implement KeySafe
    - designed to satisfy "high B-level" orange-book requirements
    - probably would have worked
    - company folded before project finished

The KeyKOS implementation of Unix was the system that was defeated in slide XX-26.

KeySafe


Compartment
Compartment
Compartment
Guard
Guard
Guard
Objects
Security Reference Monitor

Speculative Execution Attacks

An Important Assumption

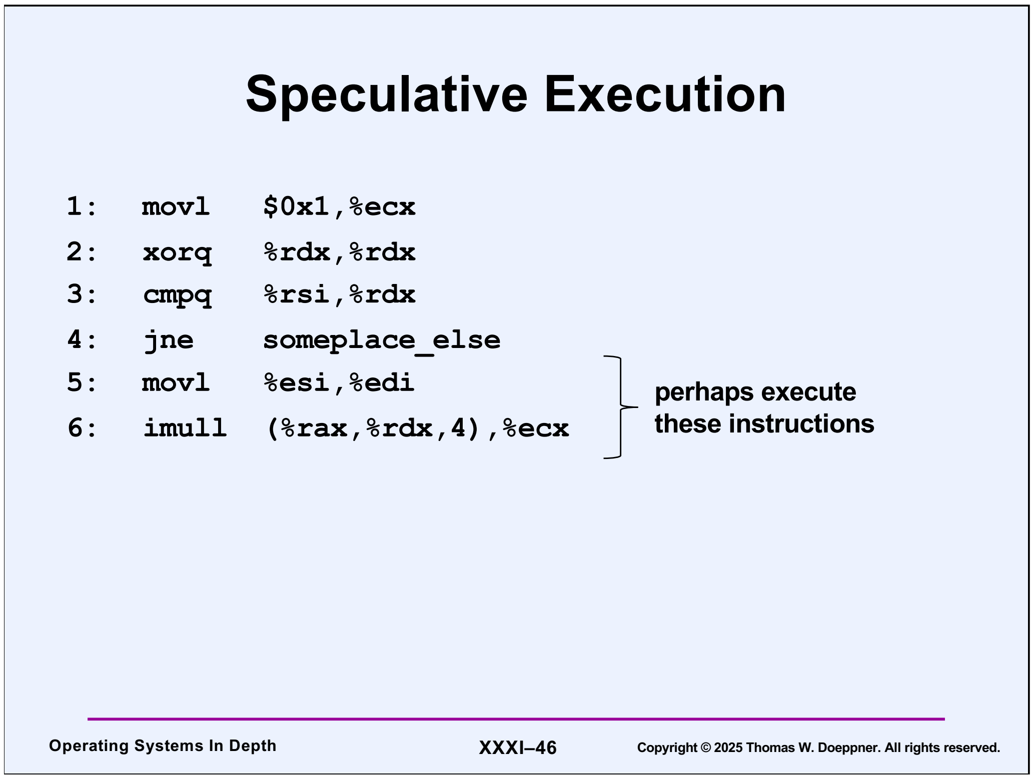
- Pages cannot be read if read permission is off
  - on Intel x86-64
    - rings 0-2 can read all mapped pages
    - ring 3 can read only those pages for which read permission is explicitly given (in the page-table entries)

What If the Assumption is Wrong?

- Code in ring 3 can read everything that's currently mapped
- In Linux, all physical memory is mapped into kernel's address space
  - kernel address space mapped into every process
  - every process can read all physical memory

Making the Assumption True

- Processor checks page protection on each access of memory
  - a fault occurs and access is not allowed if page is marked not readable

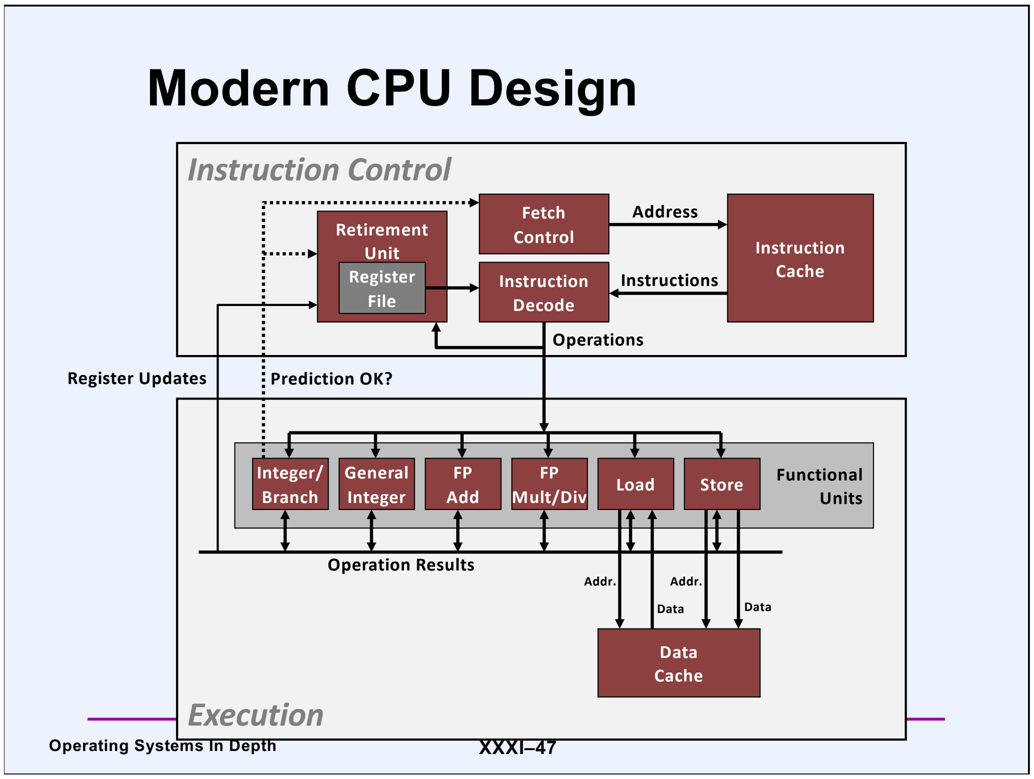


Speculative Execution

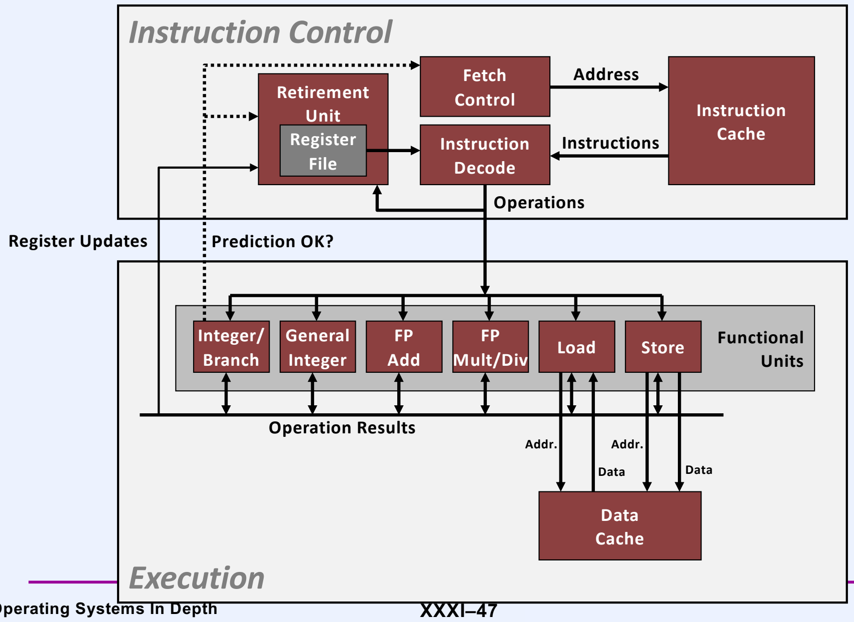
```
1:   movl    $0x1,%ecx
2:   xorq    %rdx,%rdx
3:   cmpq    %rsi,%rdx
4:   jne     someplace_else
5:   movl    %esi,%edi
6:   imull   (%rax,%rdx,4),%ecx
```

perhaps execute these instructions

This slide is adapted from CS 33. The last two instructions might well be executed, but whether they actually have an effect depends on whether the conditional jump is taken.



Modern CPU Design



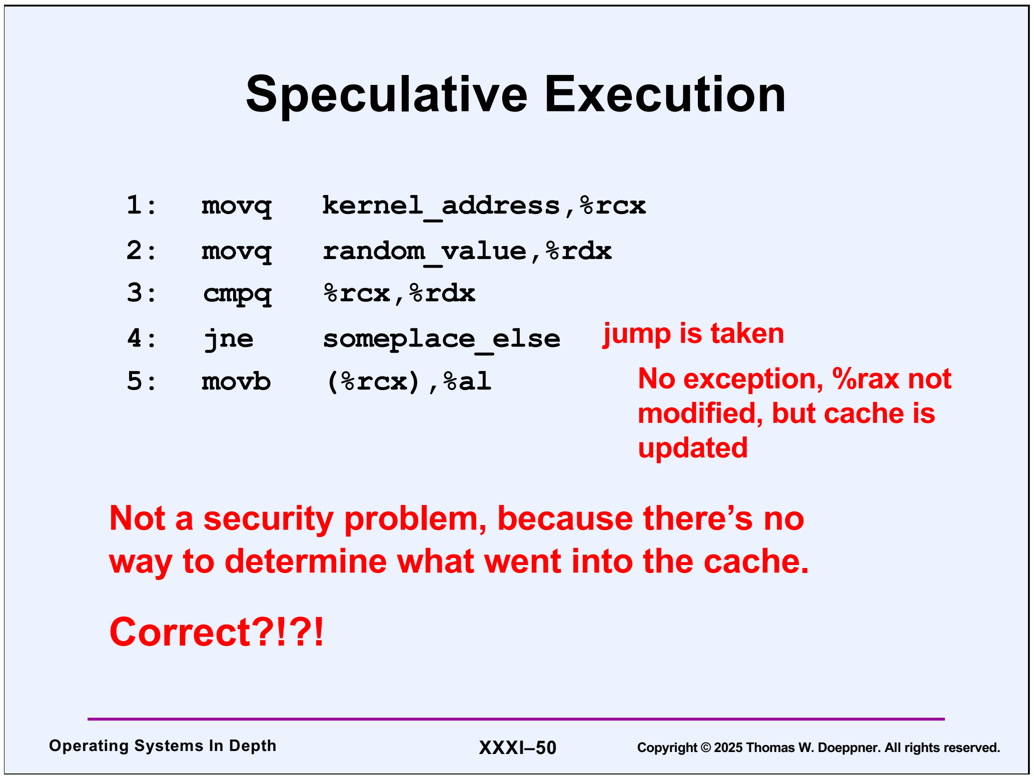
Supplied by CMU.

Speculative Execution

- If speculatively executed instruction turns out to be used (branch not taken), its results are *retired*
  - e.g., if a load into a register, the register is updated
  - if speculatively executed instruction results in exception, exception is taken
- If not used (branch taken), register is not updated
  - exceptions are ignored

Micro Operations

- Machine instructions are actually composed of micro operations, which can be executed concurrently by the various functional units
- To fetch from memory into register, (at least) two micro operations are used:
  - load value from memory
    - value goes into cache
  - check if access is allowed
    - if not, register not updated
    - exception occurs
      - unless executed speculatively and instruction not used



Speculative Execution

```
1:   movq    kernel_address,%rcx
2:   movq    random_value,%rdx
3:   cmpq    %rcx,%rdx
4:   jne     someplace_else
5:   movb    (%rcx),%al
```

jump is taken

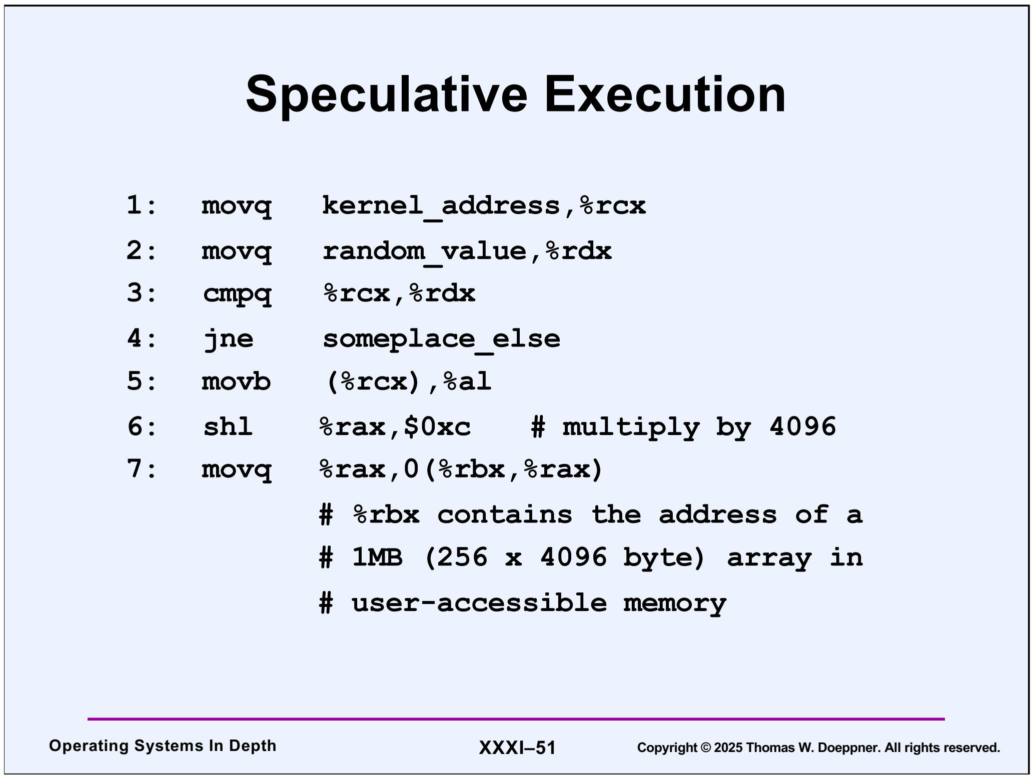
No exception, %rax not modified, but cache is updated

Not a security problem, because there's no way to determine what went into the cache.

Correct?!?!

In this example, the instruction at line 5 is speculatively executed. It loads into %al the contents of a byte location within the kernel. This should result in an exception, but assuming the jump is taken, %al is not updated and the exception is ignored. However, the cache is updated.

Why is the kernel address being compared with a random value? It's so the hardware can't predict which way the comparison will go.

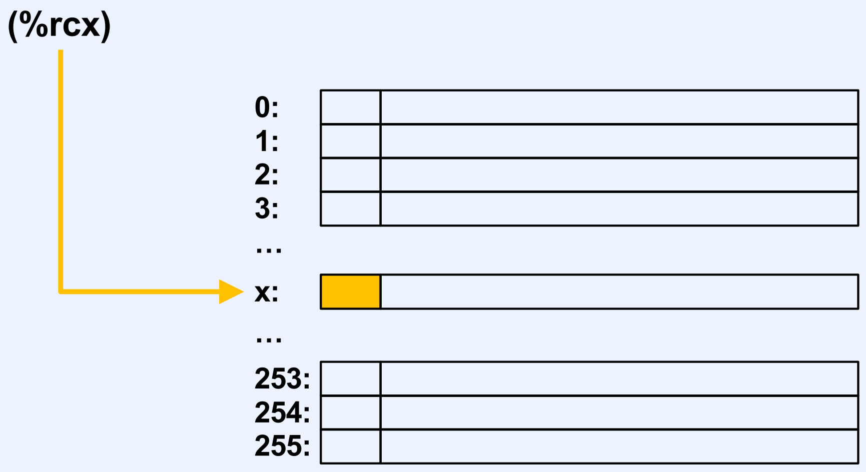


Speculative Execution

```
1:   movq   kernel_address,%rcx
2:   movq   random_value,%rdx
3:   cmpq   %rcx,%rdx
4:   jne    someplace_else
5:   movb   (%rcx),%al
6:   shl    %rax,$0xc    # multiply by 4096
7:   movq   %rax,0(%rbx,%rax)
            # %rbx contains the address of a
            # 1MB (256 x 4096 byte) array in
            # user-accessible memory
```

In this version of the example, we use the value speculatively read from the kernel to (speculatively) index into an array (recall that register %al is the low byte of register %rax), where a value is (speculatively) written (it doesn't really matter what's written there). The value is first multiplied by 4096, so that, effectively, the value is indexing into an array of 4096-byte items. Thus if the processor does prefetching, it won't prefetch beyond the array element being accessed (we write 8 bytes into the element, even though its size is 4096 bytes).

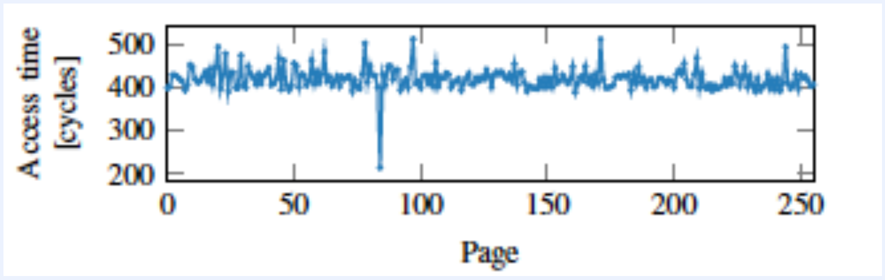
The Array



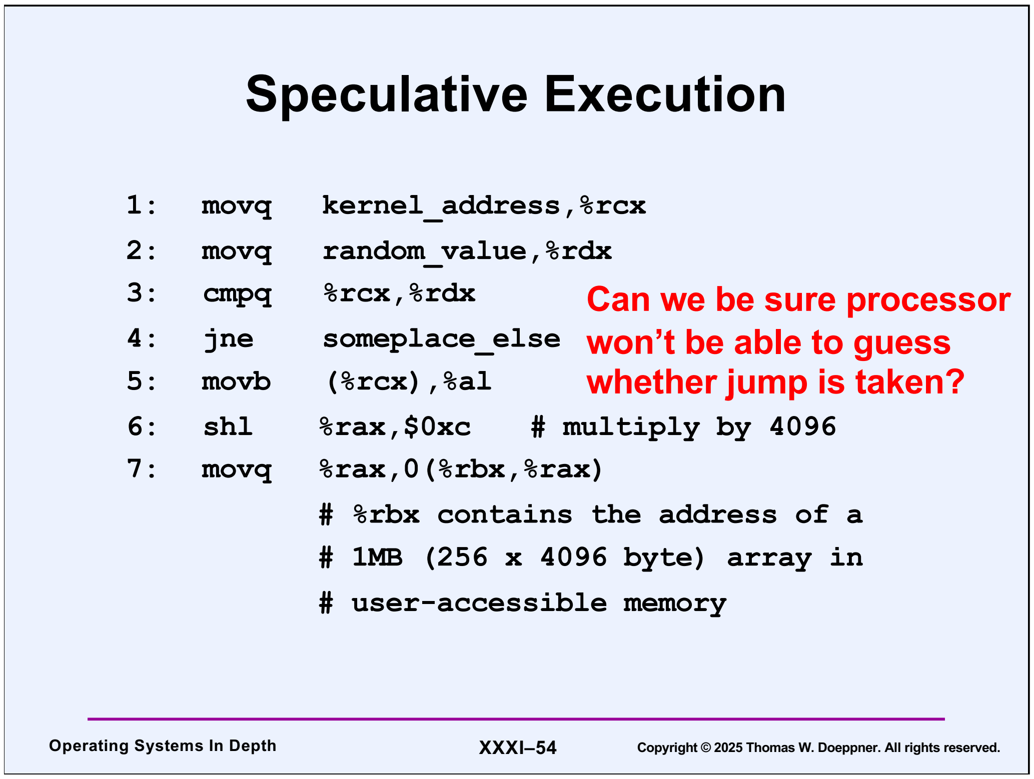
We indexed into the array using the value read from kernel memory and speculatively stored a value there. Since these speculatively executed instructions ultimately did not take effect, the actual array was never modified. But portions of it, accessed by the speculatively executed instructions, are in the processor's cache.

Assume that none of the array was in the processor's cache until this value was written. If we now attempt to read the first byte of each element of the array, all accesses will be uncached and slow, except for the access to that element we've just written to. Thus, we can figure out which element was written to based on the time it takes to access it.

Accessing the Array



This graph is from https://meltdownattack.com/meltdown.pdf, produced by the discoverers of the attack.

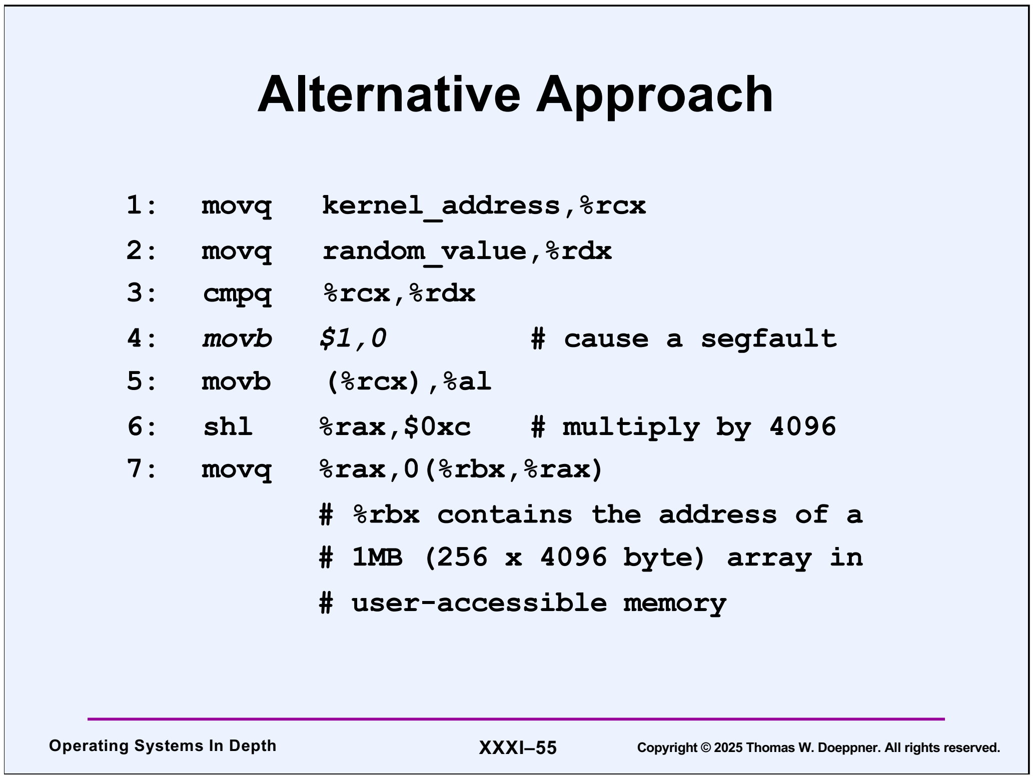


Speculative Execution

```
1:   movq   kernel_address,%rcx
2:   movq   random_value,%rdx
3:   cmpq   %rcx,%rdx
4:   jne    someplace_else
5:   movb   (%rcx),%al
6:   shl    %rax,$0xc    # multiply by 4096
7:   movq   %rax,0(%rbx,%rax)
            # %rbx contains the address of a
            # 1MB (256 x 4096 byte) array in
            # user-accessible memory
```

Can we be sure processor won't be able to guess whether jump is taken?

If we run this code sufficiently often, the processor's speculative-execution logic may figure out that the jump is going to be taken and choose not to speculatively execute the following instructions. Another approach to make this work is to use Intel's implementation of transactional memory, TSX, to ensure that the code in lines 5 through 7 is not retired.



Alternative Approach

```
1:   movq   kernel_address,%rcx
2:   movq   random_value,%rdx
3:   cmpq   %rcx,%rdx
4:   movb   $1,0          # cause a segfault
5:   movb   (%rcx),%al
6:   shl    %rax,$0xc     # multiply by 4096
7:   movq   %rax,0(%rbx,%rax)
            # %rbx contains the address of a
            # 1MB (256 x 4096 byte) array in
            # user-accessible memory
```

Here we replace the fourth line with code that causes a segfault. (Lines 2 and 3 could be removed.) The instructions in lines 5 through 7 will still be speculatively executed, but because of the segfault they will not be retired. Thus the effect is the same as with the previous approach. However, what do we do about the segfault? We can set up a signal handler for it – perhaps the handler accesses the array to determine which cache line was written to.

Kernel Address Mapping

- On Linux and OSX
  - kernel is mapped into every user process
  - all physical memory is mapped into kernel

  - thus all physical memory can be accessed via speculative execution

Meltdown

- Can read all of kernel memory on a Linux machine at the rate of 503 KB/sec
  - it's not dependent on software bugs: works on a bug-free kernel
  - achieved by using Intel TSX so that the speculatively executed instructions are not retired
  - slows down to 122 KB/sec if segfaults are used

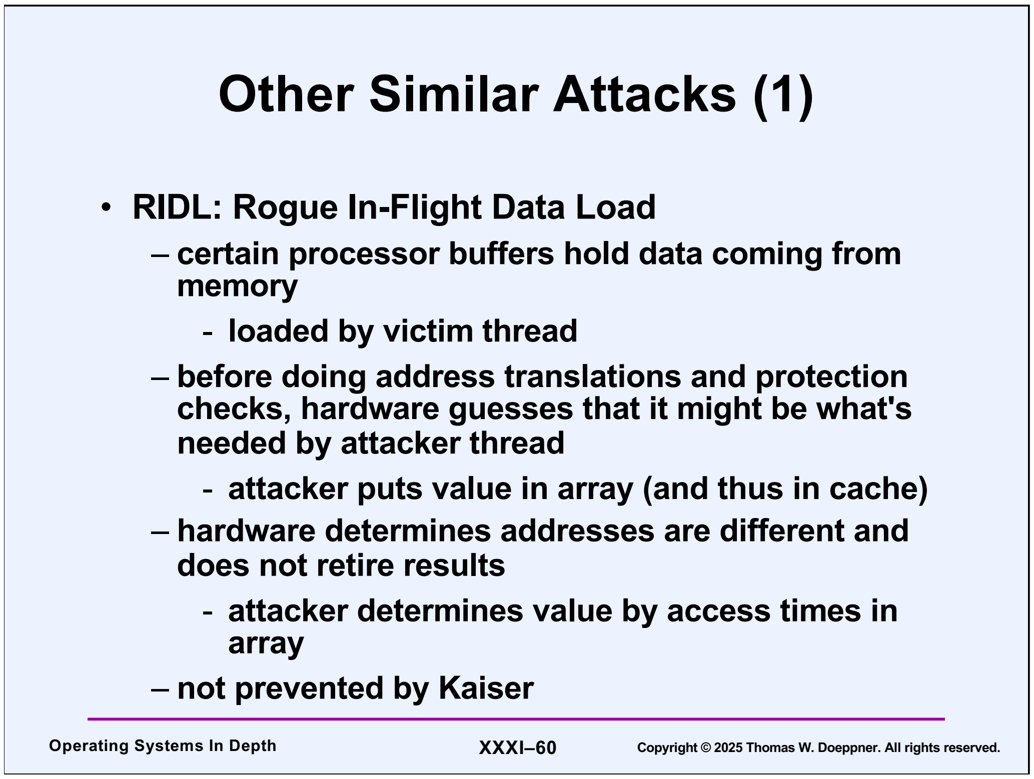
Countermeasures

- KASLR
  - kernel address-space layout randomization
  - kernel starts at a randomly chosen address
    - 40 bits of randomization
    - try all possibilities …
- KAISER
  - “kernel address-space isolation to have side channels efficiently removed”
  - kernel address space is not mapped when in user mode
  - prevents meltdown!

Quiz 4

As stated, KAISER requires that there are no mappings of kernel virtual addresses when the processor is running in ring 3 (user mode).

a) This is completely innocuous and has no adverse effects
b) This may “break” some otherwise correct user code
c) This may have noticeable performance effects
d) Both b and c are true

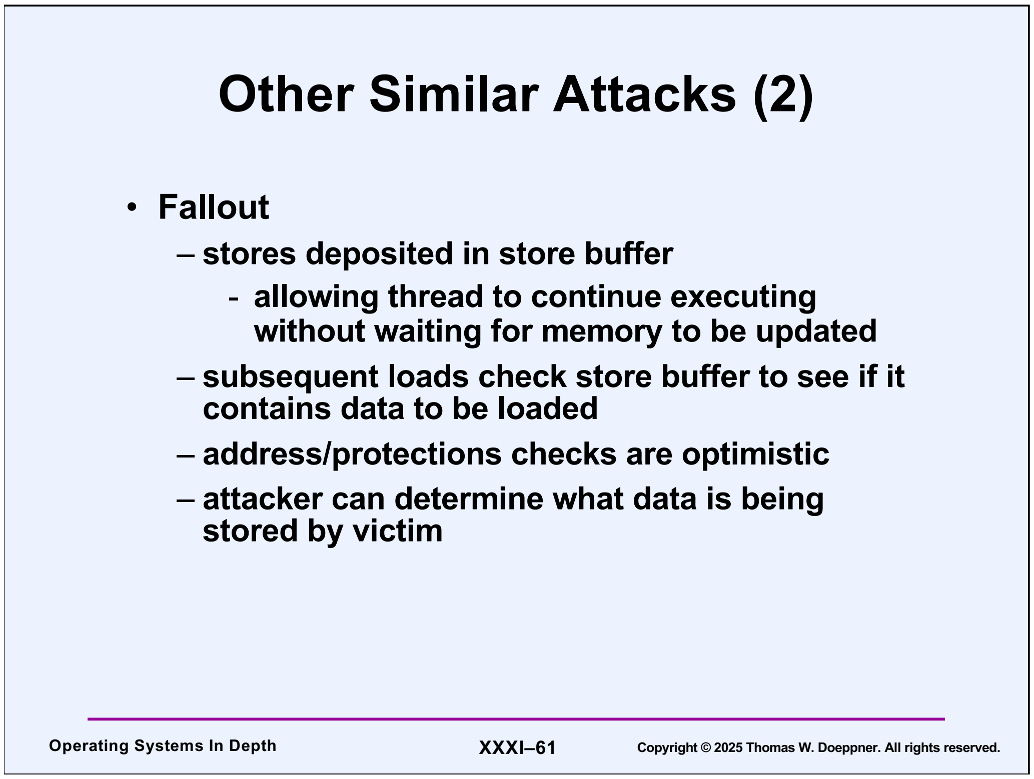


Other Similar Attacks (1)

- RIDL: Rogue In-Flight Data Load
  - certain processor buffers hold data coming from memory
    - loaded by victim thread
  - before doing address translations and protection checks, hardware guesses that it might be what's needed by attacker thread
    - attacker puts value in array (and thus in cache)
  - hardware determines addresses are different and does not retire results
    - attacker determines value by access times in array
  - not prevented by Kaiser

This attack is described in https://mdsattacks.com/files/ridl.pdf. It's dependent on the Intel x86-64 processor design, though similar attacks might be possible in other architectures.

By running the "passwd" program in the victim process (many times), attacker in concurrently executing process could learn contents of /etc/shadow.



Other Similar Attacks (2)

- Fallout
  - stores deposited in store buffer
    - allowing thread to continue executing without waiting for memory to be updated
  - subsequent loads check store buffer to see if it contains data to be loaded
  - address/protections checks are optimistic
  - attacker can determine what data is being stored by victim

Details can be found in https://mdsattacks.com/files/fallout.pdf.

Summary

- Attacks rely on extreme optimization done in hardware
  - depend on detailed knowledge of hardware architecture
  - much of this was learned via reverse engineering
- Attacks allow one to extract information across virtual machine and container boundaries