CSCI 667: Concepts of Computer Security

Lecture 12

Prof. Adwait Nadkarni

Derived from slides by William Enck, Patrick McDaniel and Trent Jaeger

OS Security

OS Security

- A secure OS should provide (at least) the following mechanisms
  - Memory protection
  - File protection
  - General object protection
  - Access authentication
- How do we go about designing a trusted OS?
- "Trust" in this context means something different from "Secure"

Access Control Lists

- ACL: a list of the principals that are authorized to have access to some object.
- Eg.,

	O_2
S_1	Y
S_2	Y
S_3	Y

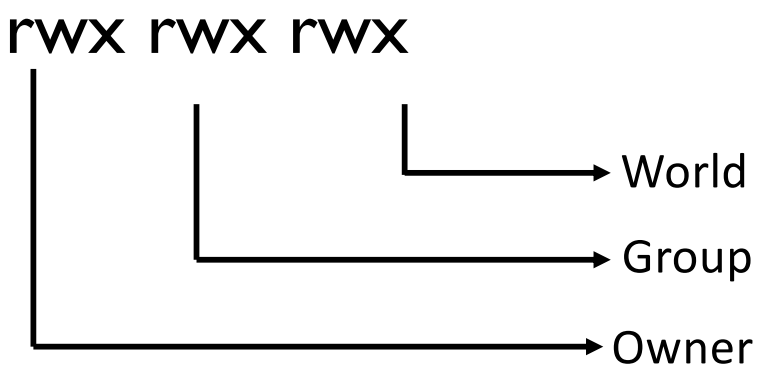
- Or more correctly:

  O_1: S_1
  O_2: S_1, S_2, S_3
  O_3: S_3

- We are going to see a lot of examples of these throughout the semester.

The UNIX FS access policy

- Really, this is a bit string ACL encoding an access matrix
- E.g., `-rwxrw---x  1 adwait  staff  0 Oct  9 16:42 file.txt`



- And a policy is encoded as "r", "w", "x" if enabled, and "-" if not, e.g,

rwxrw---x

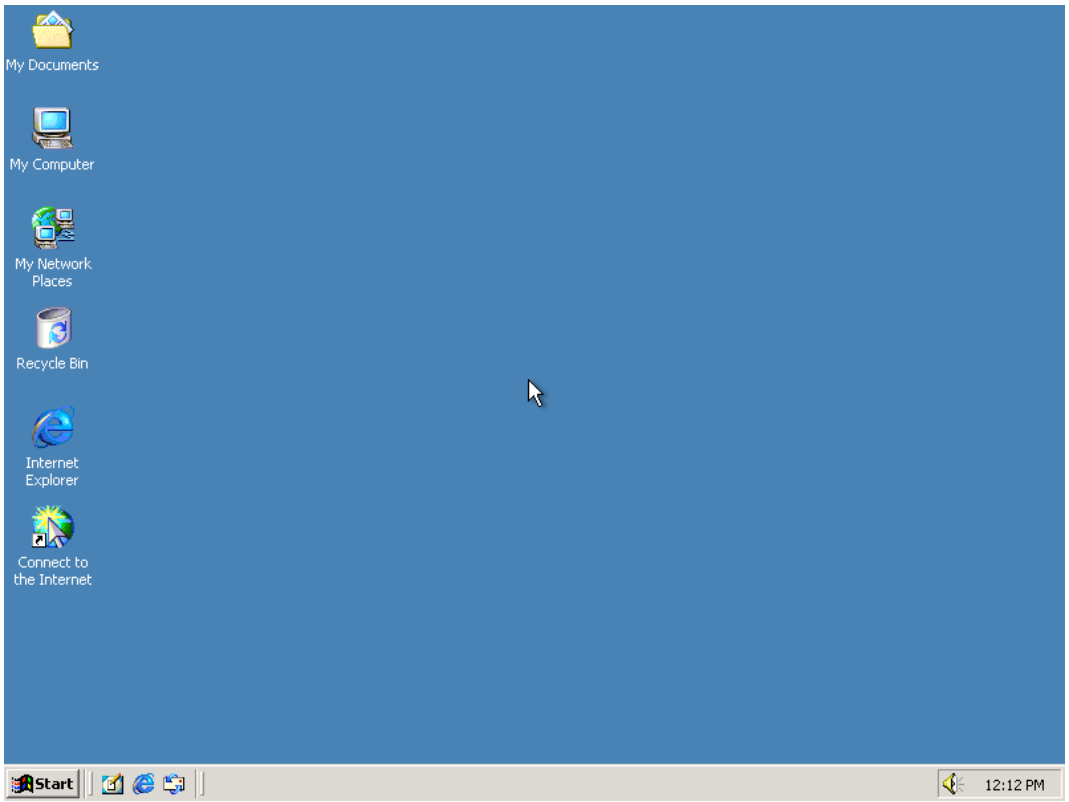
- Says user can read, write and execute, group can read and write, and world can execute only.

Caveats: UNIX Filesystem

- Access is often not really this easy: you need to have certain rights to parent directories to access a file (execute, for example)
  - The reasons for this are quite esoteric
- **The preceding policy may appear to be contradictory**
  - *A member of the group does not have execute rights, but members of the world do, so …*
  - A user appears to be both allowed and prohibited from executing access
  - Not really: these policies are ***monotonic*** … the absence of a right does not mean they should not get access at all, just that that particular identity (e.g., group member, world) is not explicitly not be given that right.

Windows grows up ...

- Windows 2000 marked the beginning of real OS security for the windows systems ...



Tokens

- Like the UID/GID in a UNIX process
  - User
  - Group
  - Aliases
  - Privileges (predefined sets of rights)
- May be specific to a domain
- Composed into global SID
- Subsequent processes inherit access tokens
  - Different processes may have different rights

Access Control Entries

- DACL in the security descriptor of an object
  - e.g., like "rwx"
    - List of *access control entries* (ACEs)

ACE structure (proposed by Swift et al)

1. Type (grant or deny)
2. Flags
3. Object Type: global UID for type (limit ACEs checked)
4. InheritedObjectType: complex inheritance
5. Access rights: access mask
6. Principal SID: principal the ACE applies to

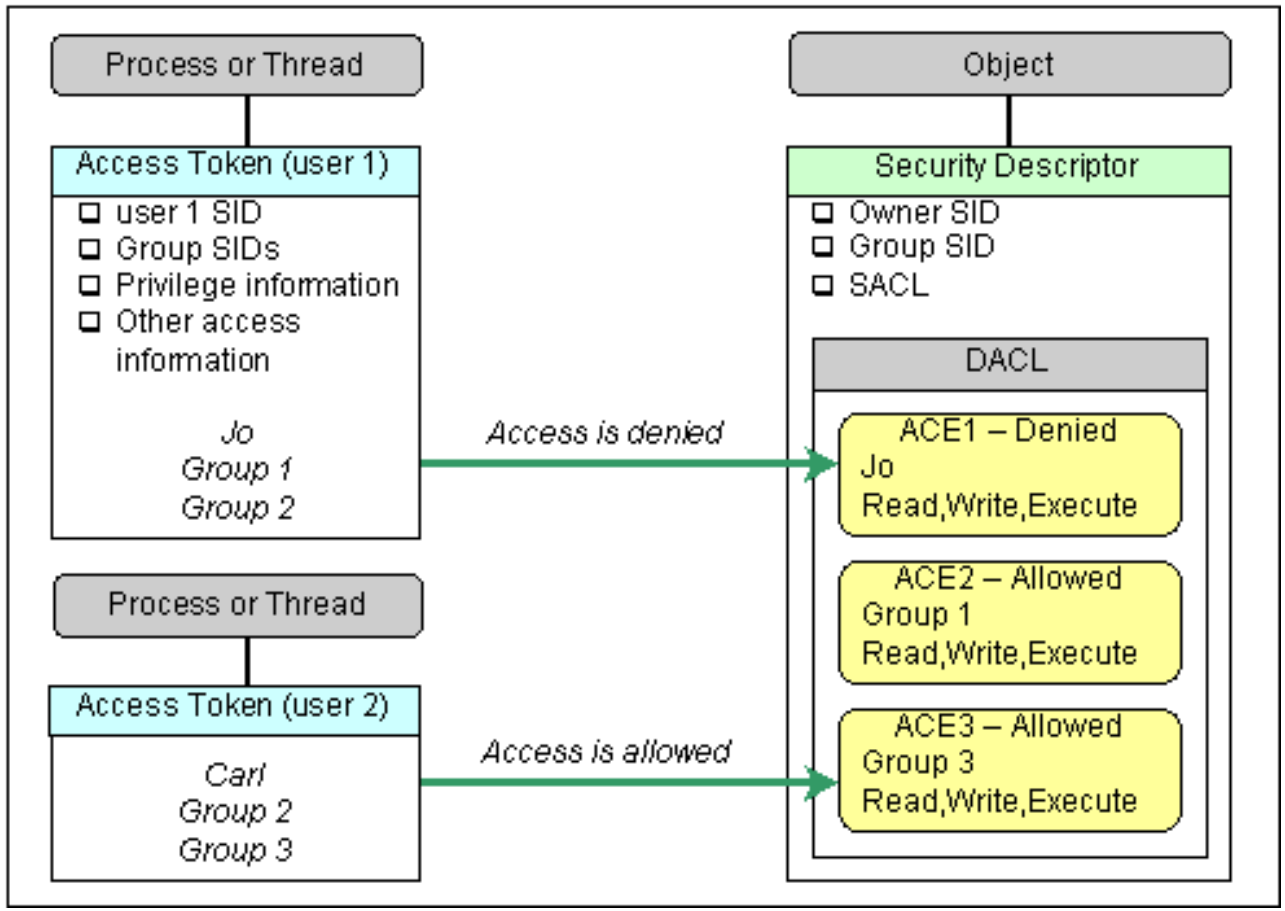
ACE Authorization

- The ACEs for a particular request are totally ordered.
- Start form the top and check each:
- Checking algorithm
  - Authorizing for SIDs in token on *set of rights*

1. if ACE matches SID (user, group, alias, etc)
   a. ACE denies access for specified right -- *deny*
   b. ACE grants access for some rights -- *need full coverage*
2. If reach the bottom and not all granted, request denied

Access Checking with ACEs

- Example



Window Vista Integrity

- Integrity protection for writing
- Defines a series of protection level of increasing protection
  - installer (highest)
  - system
  - high (admin)
  - medium (user)
  - low (Internet)
  - untrusted (lowest)

- Semantics: If subject's (process's) integrity level *dominates* the object's integrity level, then the write is allowed

Vista Integrity

S1(installer)

O1(admin)

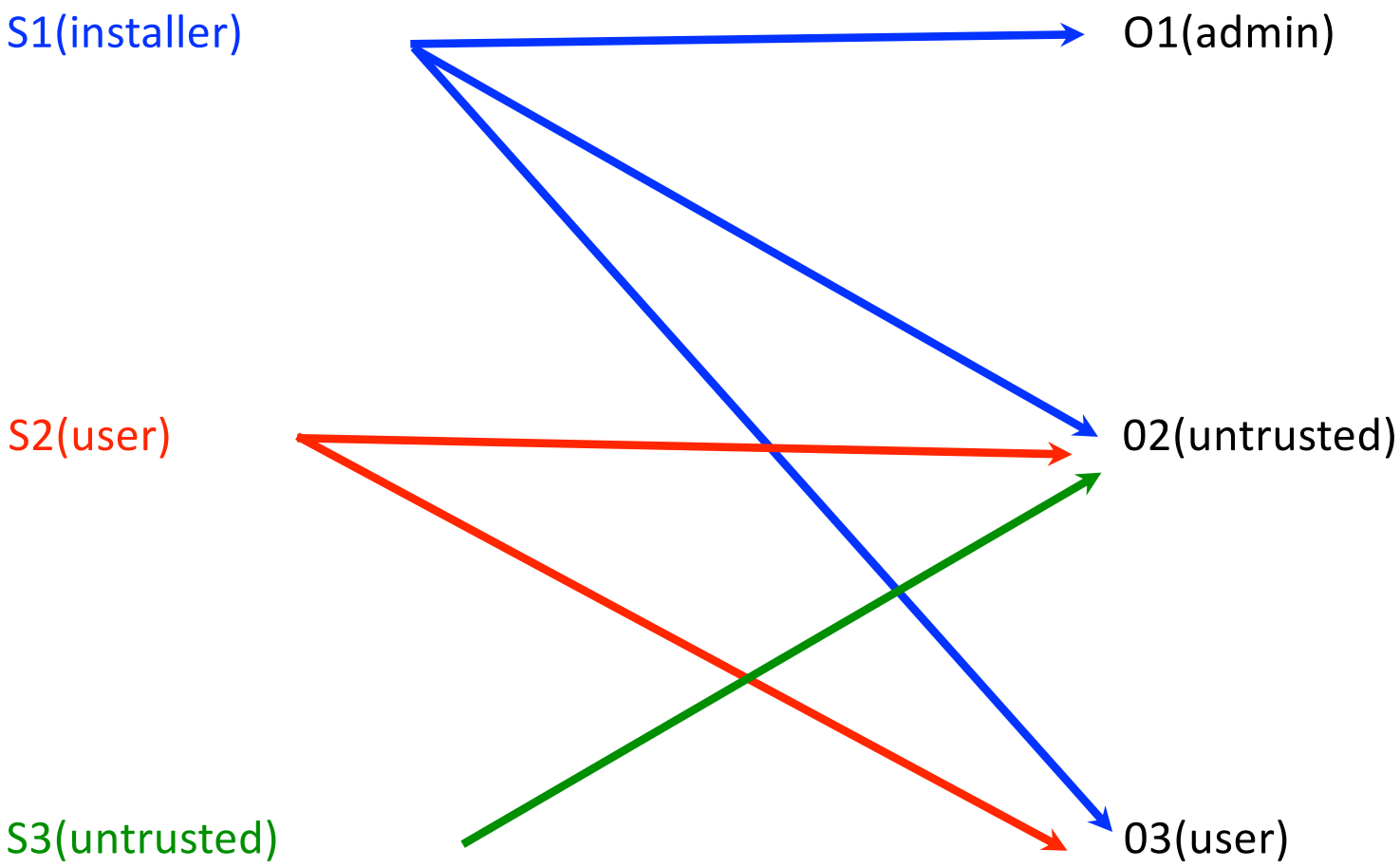
S2(user)

02(untrusted)

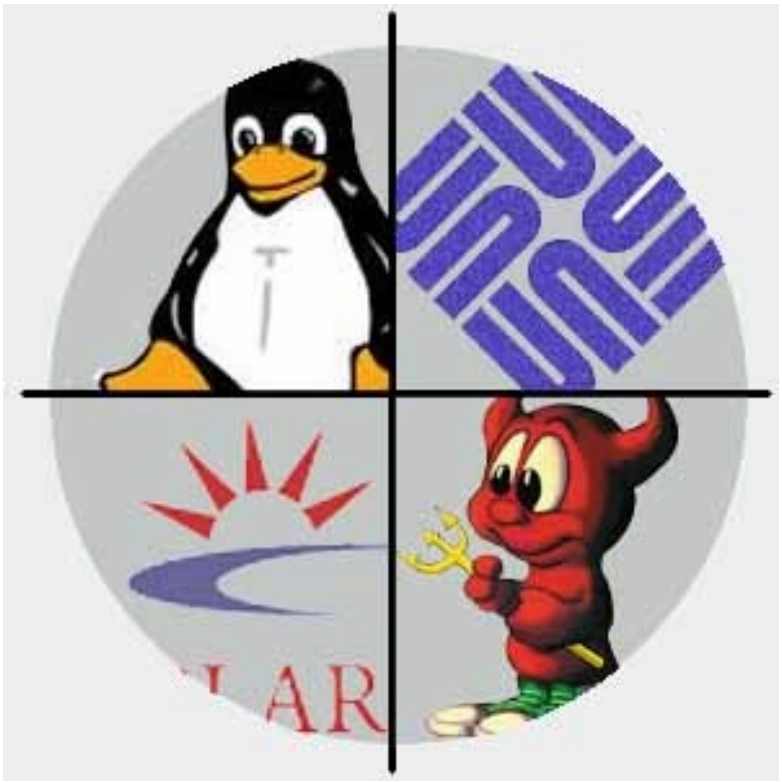
S3(untrusted)

03(user)

Vista Integrity



And now back to UNIX ...



UID Transition: Setuid

```
-rwsrw---x  1 adwait  staff  0 Oct  9 16:42 file.txt
```

- A special bit in the mode bits
- Execute file
  - Resulting process has the effective (and fs) UID/GID of file owner
- Enables a user to *escalate privilege*
  - For executing a trusted service
- Downside: User defines execution environment
  - e.g., Environment variables, input arguments, open descriptors, etc.
- Service must protect itself or user can gain root access
- *All UNIX services involves root processes -- many via setuid*

/tmp Vulnerability

- creat(pathname, mode)
- S-bit to the world triple: only owner can modify that file.
- O_EXCL flag
  - if file already exists this is an error
- Potential attack
  - Attacker creates file in shared space (/tmp)
  - Give it a filename used by a higher authority service
  - Make sure that service has permission to the file
  - If *creat* is used without O_EXCL, then can share the file with the higher authority process

Other OS Vulnerabilities

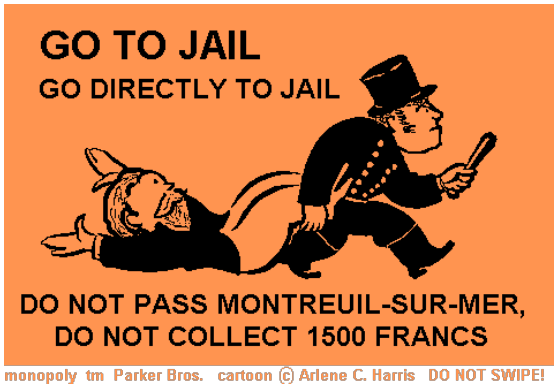
- Objects w/o sufficient control
  - Windows registry, network
  - Executables are everywhere
    - Web content, Email, Documents (Word)
- Libraries
  - Load order permits malware defined libraries
  - Library search path can load malicious libraries
- Labeling is wrong
  - Mount a new file system; device
- Malware can modify your permissions
  - Inherent to discretionary model

Sandboxing

- An execution environment for programs that contains a limited set of rights
  - A subset of your permissions (meet secrecy and integrity goals)
  - Cannot be changed by the running program (mandatory)

UNIX Chroot

- Create a *domain* in which a process is *confined*
  - Process can only read/write within file system subtree
  - Applies to all descendant processes
  - Can carry file descriptors in 'chroot jail'




```
SYNOPSIS
     chroot [-u -user] [-g -group] [-G -group,group,...] newroot [command]
```

Chroot Vulnerability

- Unfortunately, chroot can trick its own system
  - define a passwd file at <newroot>/etc/passwd
  - run su
    - su thinks that this is the real passwd file
  - gives root access
    - Use mknod to create device file to access physical memory
- Setup requires great care
  - Never run chroot process as root
  - Must not be able to get root privileges
  - No control by chrooted process (user) of contents in jail
  - Be careful about descriptors, open sockets, IPC that may be available

Capabilities

Process-specific Permissions

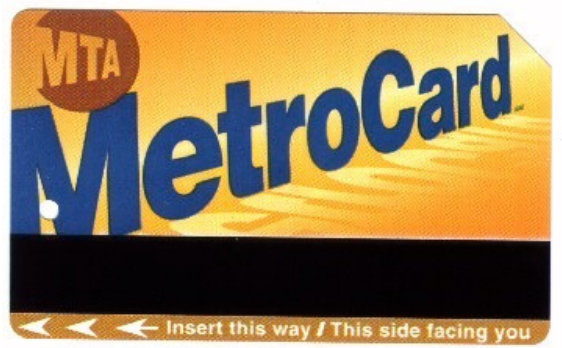
- Design the permissions of a process specific to its use
- How do we *change the permissions* of a process in an ACL system?

Confused Deputy Problem

- Imagine a multi-client server
  - Clients have a different set of objects that they can access
- In an ACL system, the server always has access to all the objects
  - What happens if a client tricks the server into accessing into another client's objects?
  - Shouldn't the server only have access to that client's objects for its *requests*?

Capabilities

- A capability is the tuple (object, rights)
- A capability system implements access control by checking if the process has an appropriate capability
  - Simple, right?
  - This is a little like a ticket in the Kerberos system

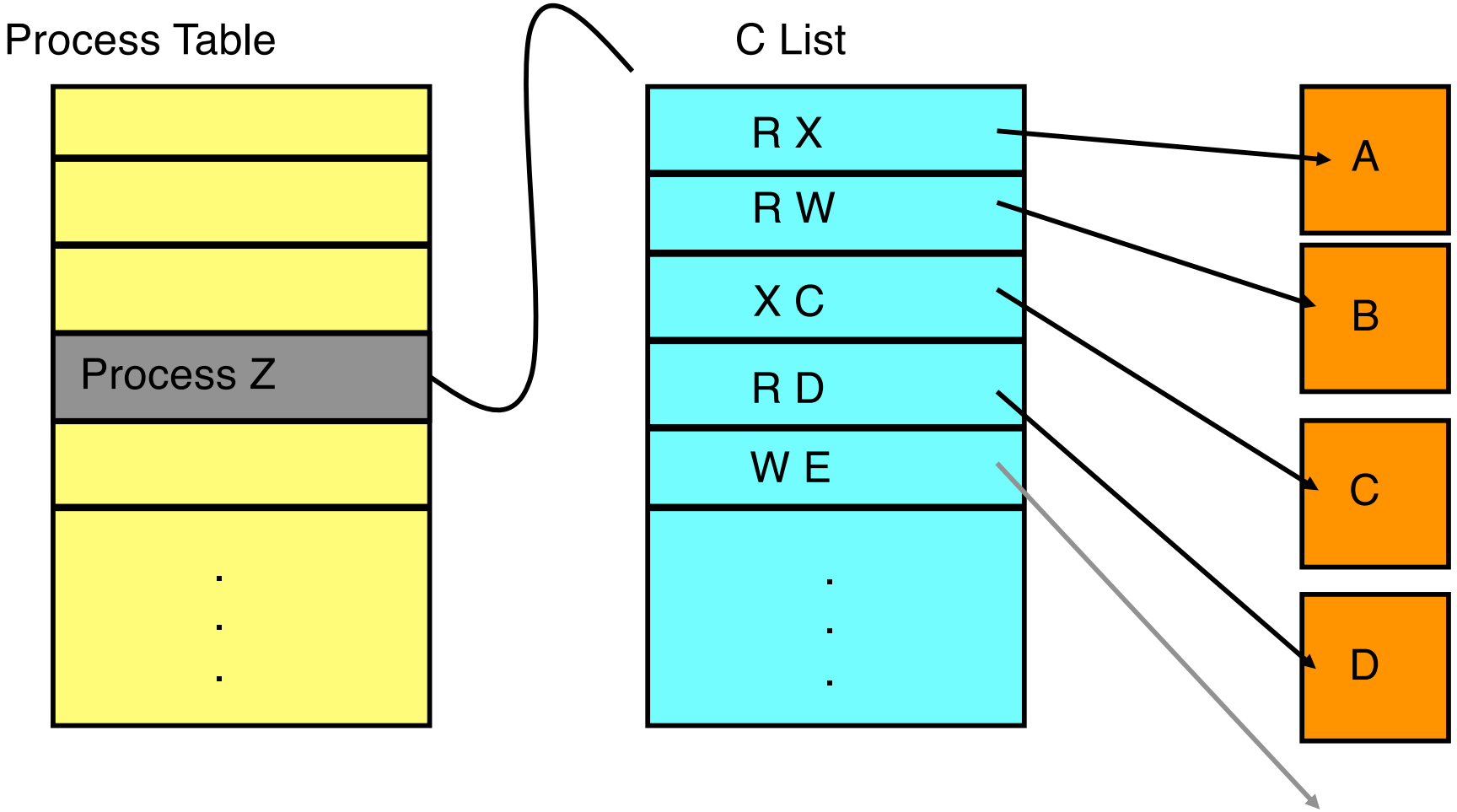


- Q: Does this eliminate the need for authentication?

Capabilities

- A: Well, yes and no …
- Capabilities remove the overhead of managing per object rights, but add the overhead of managing capabilities
- Moreover, to get any real security, they have to be unforgeable
  - Hardware tags (to protect capabilities)
  - Protected address space/registers
  - Language based techniques
    - Enforce access restrictions on caps.
  - Cryptography
    - Make them unforgeable

Real OS Capabilities



- The OS kernel manages capabilities in the process table, out of reach of the process
- Capabilities added by user requests (that comply with policy)

User space capability?

- Well, what are the requirements?
  - Authenticity/integrity - do not want malicious process to forge capabilities
- Start with the data itself: [object, rights]
  - Object is typically encoded with identifier, or by some other tag (capabilities are sometimes known as tags)
  - Rights are often fixed (read, modify, write, execute, etc.)
- Now, do what you would with any other data (assume the kernel has a secret key k)

$$E(k, [O_i, r_1, r_2, \ldots r_n])$$

- What's wrong with this construction (from the website of one of the experts in the area)?

The right construction

- Encryption does not provide authenticity/integrity, it provides confidentiality

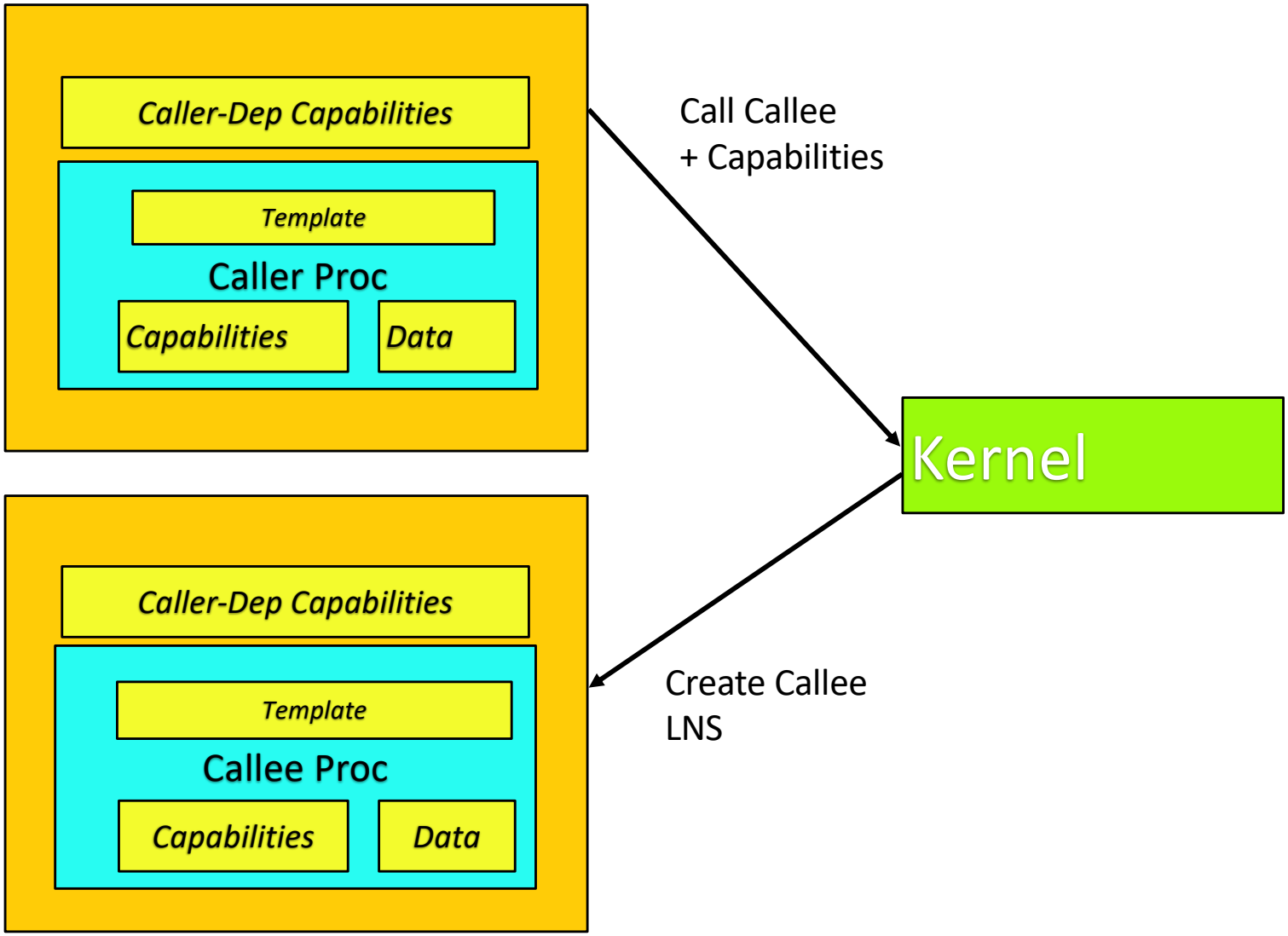
$$[O_i, r_1, r_2, \ldots r_n], \text{HMAC}(k, [O_i, r_1, r_2, \ldots r_n])$$

- So how would you attack the preceding construction?

Procedure-Level Protection Domains

- HYDRA
  - Each procedure defines a new protection domain
- Procedure
  - Code
  - Data
  - Capabilities to other objects
    - **Caller-independent**
    - **Caller-dependent templates**
- Local Name Space
  - Capabilities are bound here
  - Record of a procedure invocation (procedure instance)
- Process
  - Stack of LNSs

How HYDRA works



- Q: Which object defines the *protection domain*?

Implications of Fine-Grained Protection

- Programmer
  - Must define *templates* for procedure
  - Connect the procedure rights together
- Performance Impact

- Q: Do we need to manage rights at this level?

Multics

MAC Systems

- Major Effort: *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

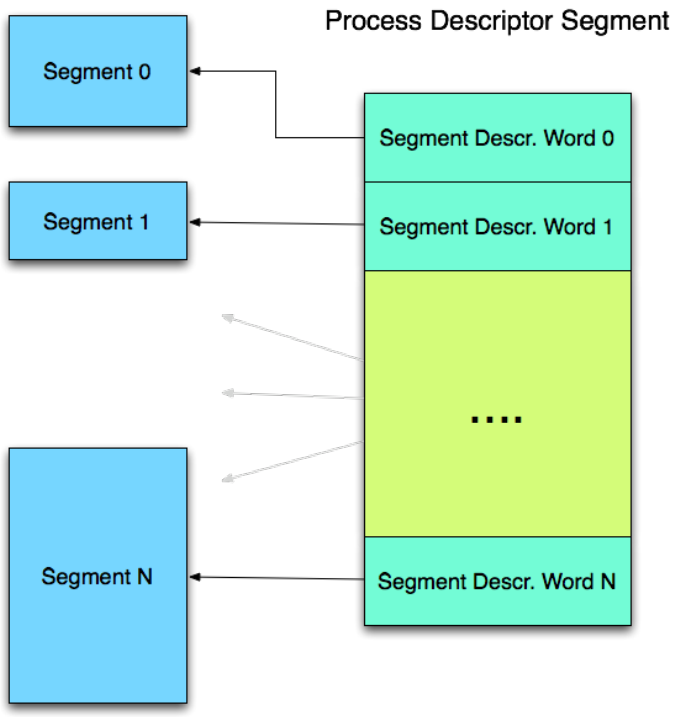
- Secrecy
  - Multilevel security
- Integrity
  - Rings of protection
- Reference Monitoring
  - Mediate segment access, ring crossing
- Resulting system is considered a high point in secure system design

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!

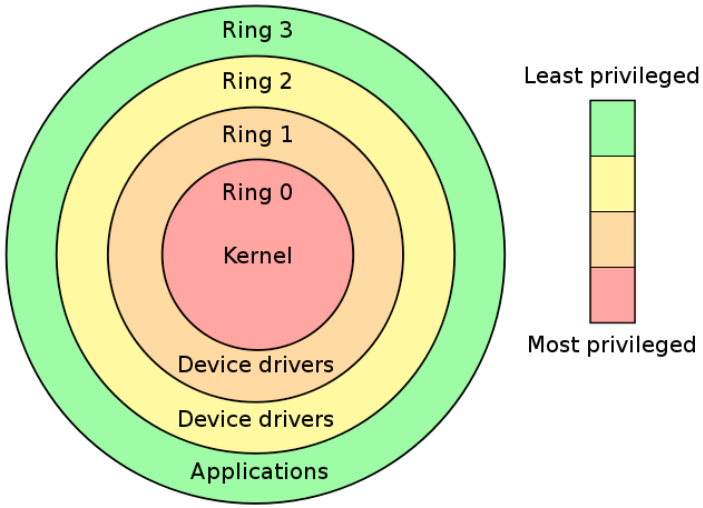
Segment Management

- 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

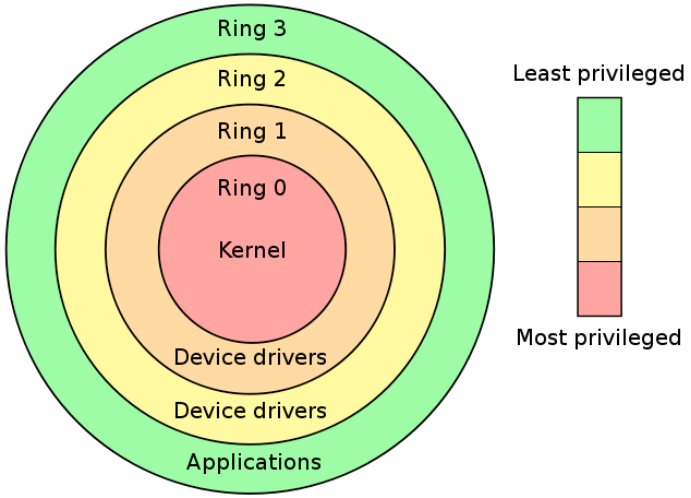
- Successively less-privileged "domains"
- Modern CPUs support 4 rings
  - Use 2 mainly: Kernel and user
- Intel x86 rings
  - Ring 0 has kernel
  - Ring 3 has application code



- 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...

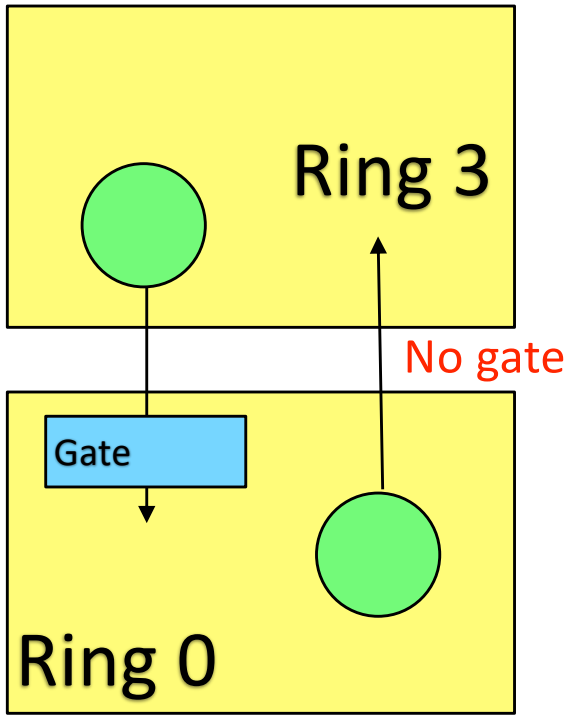


Intel Protection Ring Rules

- 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?

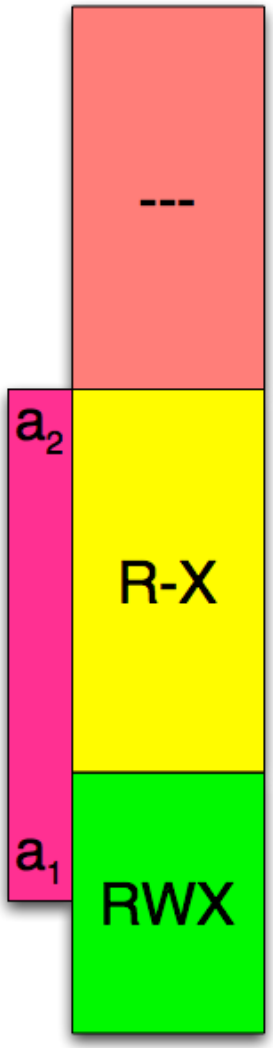
Protection Ring Rules

- 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 (system calls)



Multics Interpretation

- 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 Interpretation (cont'd)

- 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