Principles of Systems Security

15-712 Fall 2007

Why?

- Why are we reading a paper from before most of the people in this classroom were born?
- Classic Saltzer paper - thinks lucidly and precisely about issues
  - Coherent, well-integrated definitions of problems and techniques
    - Without a solid definition, hard to reason about systems
    - Particularly true in security & availability
  - Like RPC paper - lays out foundational challenges in systems security that remain pertinent today
  - Some of the phrases have changed, but core ideas roughly unchanged...

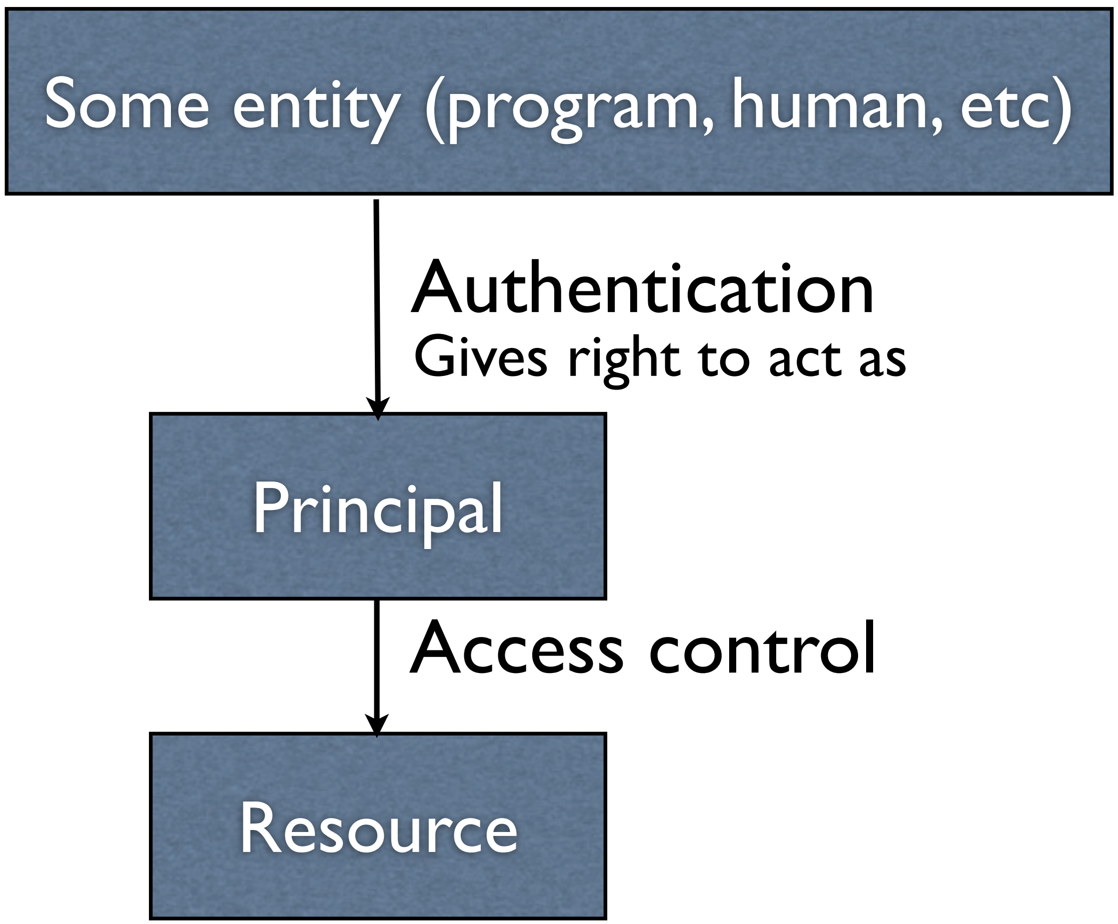
Goals

- “Security” is over-broad term. 3 typical components (using today’s terminology):
  - Confidentiality: Only authorized entities may view data
  - Integrity: Only authorized entities may modify data
  - Availability: Authorized entities can access {data, services} when they need to
- “Privacy” - ability to decide whether, and to whom, personal information is released - is a social goal. 3-tuple above *can*, but need not, provide privacy.

Who are the players?

- Principal: the unit of accountability; an entity to whom authorizations are granted.
  - 1-to-many and many-to-1 relationship with humans, programs, etc.
    - "The finance group" may read this document
    - "Dave's web browser" may write to /tmp/safari-dga/
- Resource: that which is protected; data, services, objects, etc., to which access may be granted

Linking them together



Security Policies

- Set of rules describing what principals can access what resources under what conditions
- Tons of variance here
  - Static vs Dynamic policies
    - e.g., "Chinese Wall" policy: can access A or B, but not both
      - e.g., can audit client or invest in client, but not both (insider trading...)
      - As soon as you touch data object A, you can't access B
  - Discretionary vs Mandatory ("DAC" vs "MAC")
    - DAC: User can decide who accesses data (unix, AFS, etc.)
    - MAC: Security admin (e.g.) decides
      - e.g., military "top secret" -- just because you get to know about the secret alien spaceships doesn't mean you can tell anyone else

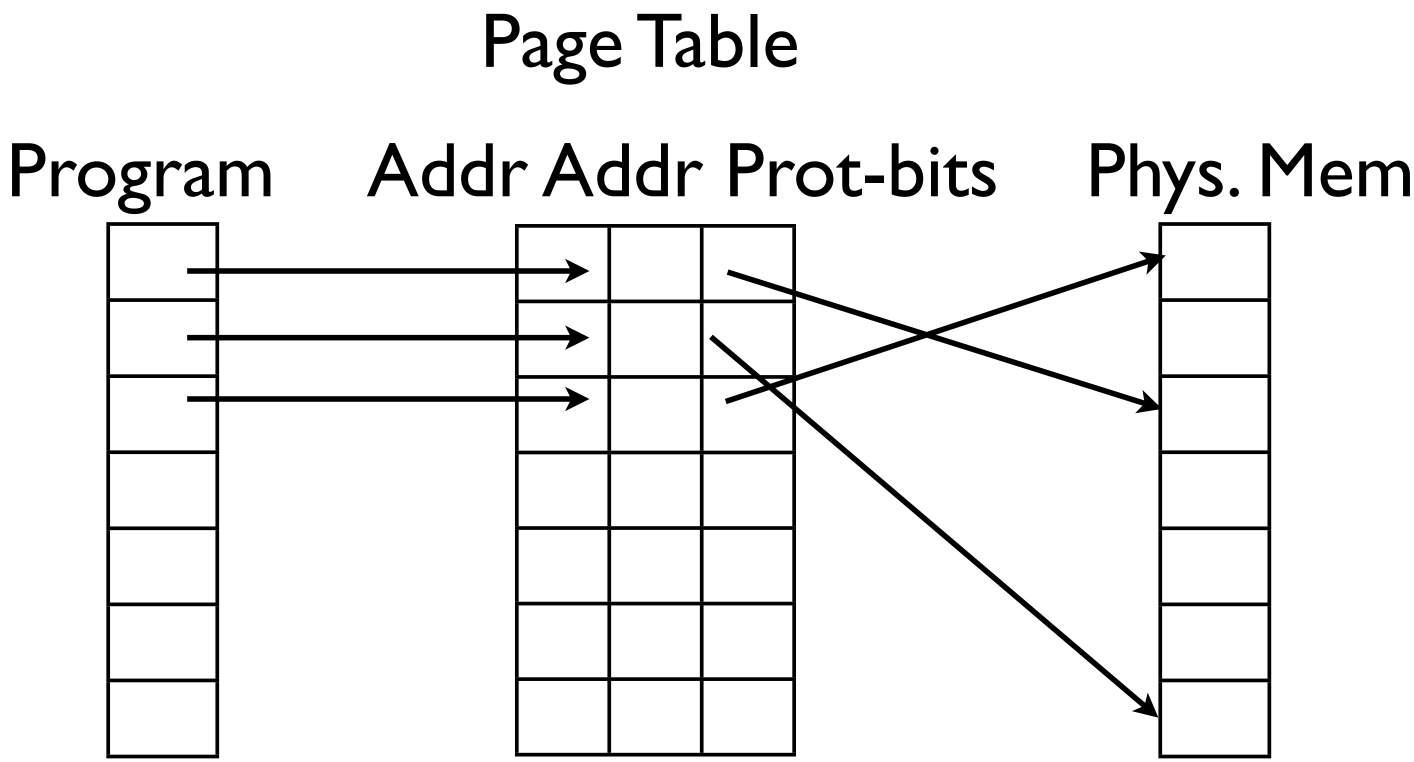
The tension

- Flexibility: What security policies can you provide?
  - Granularity of sharing?
    - All-or-nothing (easy) vs. per-object or sub-object
  - User-supplied policies or just system policies?
- Efficiency: How much work must be done on each access to resource? (e.g., must you search an ACL on each memory access??)
- Simplicity: Complex systems may be more vulnerable to implementation flaws
  - Impl must be correct; *policy specification* must also be correct. e.g., SE Linux policies very complex == few people use them in daily practice. Finer-grained == more policy bits to set...

Capabilities vs. ACLs

- Long-running debate in security (over-emphasized?)
- Reality:
  - Low-level mechanisms (e.g., memory access) *must be fast. Really fast. Hardware fast.*
    - Impractical to check ACL on every access
      - Capability or cached permission vector
      - Consider modern memory protection

MMUs, paging, and TLBs



On mem access: Check TLB. If no hit, page fault, load entry into TLB (much slower), else allow access

Updating page table

- Who can modify page table?
  - Only programs with “privilege bit”
    - Different architectures implement differently
      - supervisor bit in some register
      - protection rings (lower rings can write >= rings)
        - MULTICS had 8 rings; x86 has 4
        - e.g, kernel @ 0, device drivers @ 1, apps @ 3
        - Not a lot of OSes actually use all of them, alas -- most use 0 and 3.
          - Note that “root” != ring 0. Root is an OS concept; ring 0/supervisor means “can do _anything_”

Caching Permissions

- TLB/page table is example of permission caching
  - Necessary for fast low-level mechanisms
  - Increases complexity: requires mechanisms for consistency (e.g., TLB flushes, etc.)

UNIX Example

- Filesystem:
  - access-list permissions with 3 entries: owner, group, world
  - Discretionary access: owner can modify g/w perms
  - Access control on *open* - afterwords, get file descriptor
    - FD looks a lot like a capability!
    - Can be passed between programs...
    - Makes subsequent access fast, but in UNIX, means that writes can continue after chmod/chown (no revocation!)
      - This is good and bad. Consider:

Capability passing in UNIX

- Protected file "/secret.txt"
  - Want to implement complex access control
- chown some-principal /secret.txt
- Run "guard" program as some-principal that opens / secret.txt
- To request access, user's program sends unix-domain socket message to guard: "let me in!"
- Guard can reply and pass back FD to read secret.txt
  - user's program now has access even though it doesn't have access to the file
  - BUT: guard has no way to revoke permission.
  - Revocation is consistently challenging issue...
- (This technique isn't theoretical - used in practice...)

Capabilities

- Unforgeable pointer to protected resource
  - “Unforgeable” -- tagged, stored in special segments;
  - Generalizes:
    - dist. systems: cryptographically protected/generated
    - sometimes just picked from huge (128+-bit) namespace
    - compiler-enforced: java (modula3, etc.) references
- Allows for arbitrary, controlled sharing
-

Using Capabilities

- Program must have been explicitly given capability
- May use it as desired
  - May pass it to other programs (propagation)
    - Consider: How does this interact with a MAC system? What if I'm not *allowed* to give you read access to grades database even if I want to? (Copy bits? Limited depth?)
      - Solving this starts to sound like ACLs or more general mechanisms...
  - How do we revoke??
  - These are hard ?s:
    - Store somewhere special so can audit/find?
    - Require indirection step through "broker" of some sort?
      - Like TLB - then just invalidate broker, force people to get new capability (common technique)

Modern Examples

- Security-Enhanced Linux
  - Fine-grained, flexible access control in Linux
    - MAC + DAC + dynamic policies...
- AsbestOS
  - Decentralized Information flow control
    - Data "taints" processes that it touches
    - taint is automatically propagated

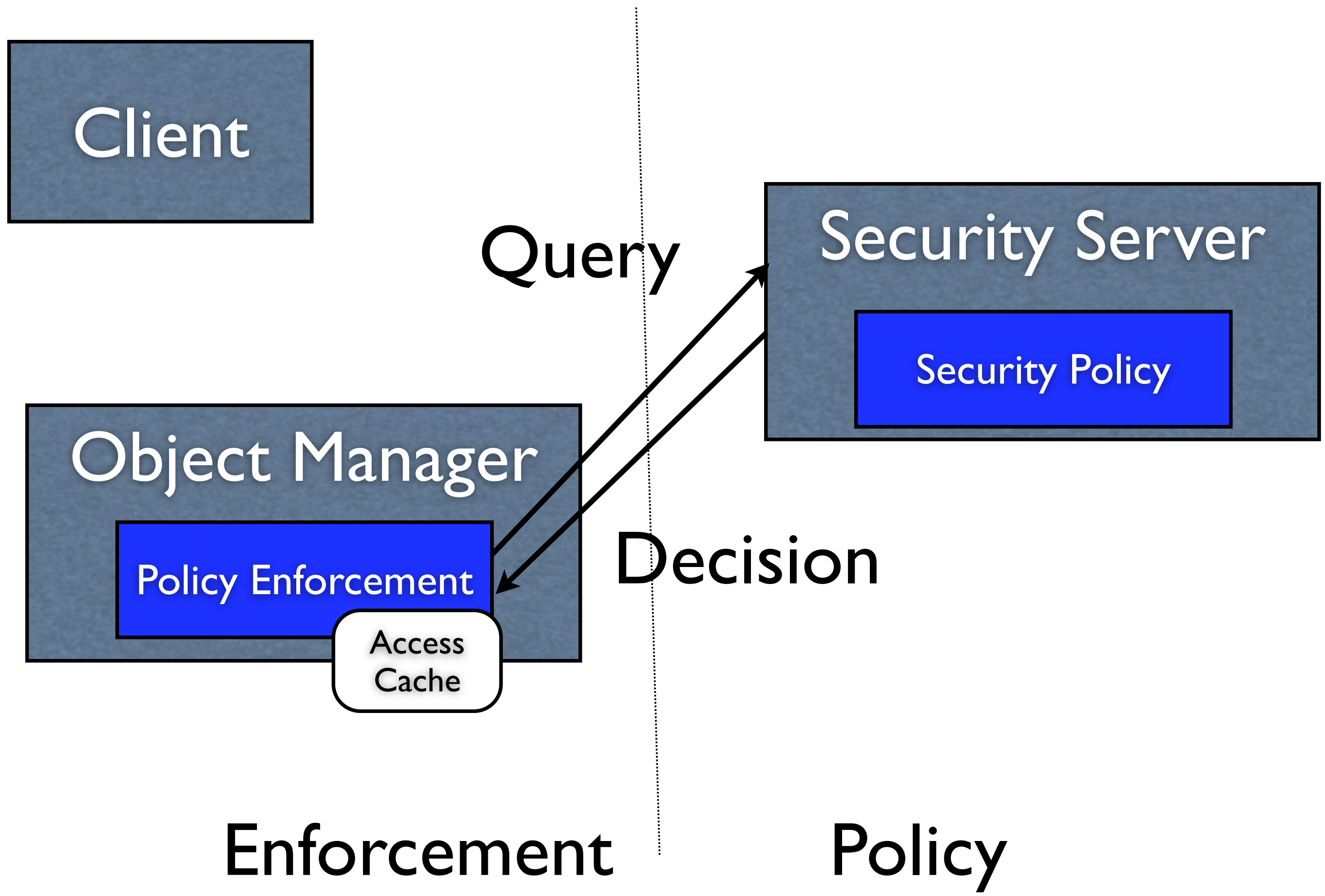
SELinux

- Core idea: Consult security policy for every security decision
  - Cache results in components that grant access
    - e.g., filesystem
    - Provide a cache invalidation mechanism
      - Much like the TLB example from earlier
- Security policy can be arbitrary program
  - But most of SELinux focus is on access-list *like* policies
    - Major example: Type enforcement (abstracts users/principals into types, limits interaction between types).

Authentication

- Login program (etc)
  - Trusted to allow starting user (e.g., "login-user") to switch to another user
  - SElinux core deals only in labels; auth programs map usernames to internal labels

Architecture



Revocation

- Roughly two-phase commit on policy changes
  - Sec server -> object managers: “A change is coming! Flush your caches!”
  - Object managers update internal state
  - Object managers -> sec server: “Okay, done”
  - When all OMs notify, sec server will let them continue
- Requirement: OMs must revoke in timely fashion; relatively small # of OMs (filesystem, network, virtual memory manager, etc.)

Why bother?

- Examples:
  - Ensure that the user ‘root’ cannot modify boot params
  - Flexible permissions: don’t need root to bind port 25 (sendmail) or write to mail files - just give sendmail append-only access to /var/spool/mail/* (containment)
- Challenge:
  - Writing correct policy for each application
    - What files does sendmail legitimately need to access? (foreach p in programs...)
  - The raw SELinux policy files are *painful.*
  - But tools emerging to create automatically. (phew)

DIFC

- Distributed Information Flow Control
- If goal is to protect information
  - Why not take an information-centric approach?
- example:
  - If program “A” sees dga’s credit card
    program “A” no longer allowed to send data to anyone other than dga
- One such system: Flume, SOSP 2007

Tags & Labels

- Alice & Bob have files on a server, but want to keep some of those files private
  - What if Bob d/l's malicious text editor that posts his files to slashdot or copies to /tmp a+r?
  - Bob tags secret data with tag *b*
  - If process *p* reads data tagged with *b*, then b \in Sp
  - p with b \in Sp can write only to procs/files q with b \in Sq
  - p with b \in Sp cannot xmit over untrusted channel

How to get work done?

- Some processes trusted to *declassify* (remove tags from the labels on a process or file)
- Implementation: Apply DIFC to file descriptors
  - e.g., IPC: two procs p and q have a socket
  - Flume checks labels on messages* (I'm lying - there's another abstraction in here, but ignore) and can drop as appropriate
  - File I/O: doesn't bother revoking (UNIX FD semantics)

Example

- /bin/sh editor
  - sh can write to terminal, must have an "export" tag
  - Bob trusts sh to export only to terminal, so launches shell with b- \in Osh (shell can remove the "b-" tag from its labels
  - sh launches editor with secrecy Sed = {b} ("editor may read Bob's files, but can't remove tag b")
  - editor opens secret file, gets tagged with {b}
- Implemented with Linux Security Module system call interposition
  - most syscalls forwarded to reference monitor which can monitor/allow/deny

Back to paper

- Example techniques - pretty current...

1. labeling files with lists of authorized users,
2. verifying the identity of a prospective user by demanding a password,
3. shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation,
4. enciphering information sent over telephone lines,
5. locking the room containing the computer,
6. controlling who is allowed to make changes to the computer system (both its hardware and software),
7. using redundant circuits or programmed cross-checks that maintain security in the face of hardware or software failures,
8. certifying that the hardware and software are actually implemented as intended.

Functional Protection

- None
- Total isolation
- System controlled sharing
  - Direct use of ACLs & default controls
- User-programmed controls
  - Extension of controls with programmed checks
- Tracking of dissemination (audit trail)
  - Ie., restriction of “classified info” after exposed to first qualified principal

Design principles

- Economy of mechanism: simple & small
- Fail-safe defaults: presumed no
- Complete mediation: wholistic assertions
- Open design: confidence thru inspection
- Separation of privilege: user & code tests
- Least privilege: grant only what is needed
- Least common mechanism: beware of widely used code
- Psychological acceptability: UI needs to work for users
- Attacker’s work factor: attacking has to cost a lot
- Compromise recording: need to track possible break-ins

Eval

- This is a survey paper
  - no eval
- Really more of a "report from the front"
  - Few in computer science were involved in really defining the hardware support for isolation (failure isolation first, then Byzantine attack)
  - Much of this became "virtual memory" and "user/kernel modes and traps" long before they became security