Chapter 13: Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection
Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems
Goals of Protection

- In one protection model, computer consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so
Principles of Protection

- Guiding principle – principle of least privilege
  - Programs, users and systems should be given just enough privileges to perform their tasks
  - Limits damage if entity has a bug, gets abused
  - Can be static (during life of system, during life of process)
  - Or dynamic (changed by process as needed) – domain switching, privilege escalation
  - “Need to know” a similar concept regarding access to data

- Must consider “grain” aspect
  - Rough-grained privilege management easier, simpler, but least privilege now done in large chunks
    - For example, traditional Unix processes either have abilities of the associated user, or of root
  - Fine-grained management more complex, more overhead, but more protective
    - File ACL lists, RBAC

- Domain can be user, process, procedure
Domain Structure

- Access-right = <object-name, rights-set>
  where rights-set is a subset of all valid operations that can be performed on the object

- Domain = set of access-rights
Domain Implementation (UNIX)

- Domain = user-id

- Domain switch accomplished via file system
  - Each file has associated with it a domain bit (setuid bit)
  - When file is executed and setuid = on, then user-id is set to owner of the file being executed
  - When execution completes user-id is reset

- Domain switch accomplished via passwords
  - `su` command temporarily switches to another user's domain when other domain's password provided

- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)
Let $D_i$ and $D_j$ be any two domain rings.

If $j < i \Rightarrow D_i \subseteq D_j$
Multics Benefits and Limits

- Ring / hierarchical structure provided more than the basic kernel / user or root / normal user design

- Fairly complex -> more overhead

- But does not allow strict need-to-know
  - Object accessible in $D_j$ but not in $D_i$, then $j$ must be $< i$
  - But then every segment accessible in $D_i$ also accessible in $D_j$
Access Matrix

- View protection as a matrix (*access matrix*)
- Rows represent domains
- Columns represent objects
- \( Access(i, j) \) is the set of operations that a process executing in Domain\(_i\) can invoke on Object\(_j\)
### Access Matrix

<table>
<thead>
<tr>
<th>domain</th>
<th>object</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td></td>
<td>read</td>
<td></td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>$D_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print</td>
</tr>
<tr>
<td>$D_3$</td>
<td></td>
<td></td>
<td>read</td>
<td>execute</td>
<td></td>
</tr>
<tr>
<td>$D_4$</td>
<td></td>
<td>read</td>
<td>write</td>
<td>read</td>
<td>write</td>
</tr>
</tbody>
</table>
Use of Access Matrix

- If a process in Domain $D_i$ tries to do “op” on object $O_j$, then “op” must be in the access matrix.

- User who creates object can define access column for that object.

- Can be expanded to dynamic protection:
  - Operations to add, delete access rights
  - Special access rights:
    - owner of $O_i$
    - copy op from $O_i$ to $O_j$ (denoted by “*”)
    - control – $D_i$ can modify $D_j$ access rights
    - transfer – switch from domain $D_i$ to $D_j$

- Copy and Owner applicable to an object
- Control applicable to domain object
Access matrix design separates mechanism from policy

- **Mechanism**
  - Operating system provides access-matrix + rules
  - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced

- **Policy**
  - User dictates policy
  - Who can access what object and in what mode

But doesn’t solve the general confinement problem
### Access Matrix of Figure A with Domains as Objects

<table>
<thead>
<tr>
<th>domain</th>
<th>object</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>Laser printer</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>read</td>
<td>read</td>
<td></td>
<td></td>
<td></td>
<td>switch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_2$</td>
<td></td>
<td></td>
<td>print</td>
<td></td>
<td></td>
<td>switch</td>
<td>switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_3$</td>
<td></td>
<td>read</td>
<td></td>
<td>execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_4$</td>
<td>read write</td>
<td>read</td>
<td>read</td>
<td>write</td>
<td></td>
<td>switch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Access Matrix with Copy Rights

#### (a)

<table>
<thead>
<tr>
<th>Domain</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>execute</td>
<td></td>
<td>write*</td>
</tr>
<tr>
<td>$D_2$</td>
<td>execute</td>
<td>read*</td>
<td>execute</td>
</tr>
<tr>
<td>$D_3$</td>
<td>execute</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (b)

<table>
<thead>
<tr>
<th>Domain</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>execute</td>
<td></td>
<td>write*</td>
</tr>
<tr>
<td>$D_2$</td>
<td>execute</td>
<td>read*</td>
<td>execute</td>
</tr>
<tr>
<td>$D_3$</td>
<td>execute</td>
<td></td>
<td>read</td>
</tr>
</tbody>
</table>
## Access Matrix With Owner Rights

(a) |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>object</strong></td>
</tr>
<tr>
<td><strong>domain</strong></td>
</tr>
<tr>
<td><strong>$D_1$</strong></td>
</tr>
<tr>
<td><strong>$D_2$</strong></td>
</tr>
<tr>
<td><strong>$D_3$</strong></td>
</tr>
</tbody>
</table>

(b) |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>object</strong></td>
</tr>
<tr>
<td><strong>domain</strong></td>
</tr>
<tr>
<td><strong>$D_1$</strong></td>
</tr>
<tr>
<td><strong>$D_2$</strong></td>
</tr>
<tr>
<td><strong>$D_3$</strong></td>
</tr>
</tbody>
</table>
### Modified Access Matrix of Figure B

<table>
<thead>
<tr>
<th>domain</th>
<th>object</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>laser printer</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>read</td>
<td></td>
<td>read</td>
<td></td>
<td></td>
<td></td>
<td>switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_2$</td>
<td></td>
<td></td>
<td></td>
<td>print</td>
<td></td>
<td></td>
<td>switch</td>
<td></td>
<td>switch control</td>
</tr>
<tr>
<td>$D_3$</td>
<td></td>
<td>read</td>
<td></td>
<td>execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_4$</td>
<td>write</td>
<td></td>
<td>write</td>
<td></td>
<td></td>
<td></td>
<td>switch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation of Access Matrix

- Generally, a sparse matrix
- Option 1 – Global table
  - Store ordered triples \(<\text{domain}, \text{object}, \text{rights-set}\>\) in table
  - A requested operation \(M\) on object \(O_j\) within domain \(D_i\) -> search table for \(<D_i, O_j, R_k>\)
    - with \(M \in R_k\)
  - But table could be large -> won’t fit in main memory
  - Difficult to group objects (consider an object that all domains can read)
- Option 2 – Access lists for objects
  - Each column implemented as an access list for one object
  - Resulting per-object list consists of ordered pairs \(<\text{domain}, \text{rights-set}\>\) defining all domains with non-empty set of access rights for the object
  - Easily extended to contain default set -> If \(M \in \text{default set}\), also allow access
Each column = Access-control list for one object
Defines who can perform what operation

Domain 1 = Read, Write
Domain 2 = Read
Domain 3 = Read

Each Row = Capability List (like a key)
For each domain, what operations allowed on what objects

Object F1 – Read
Object F4 – Read, Write, Execute
Object F5 – Read, Write, Delete, Copy
Implementation of Access Matrix (Cont.)

- Option 3 – Capability list for domains
  - Instead of object-based, list is domain based
  - **Capability list** for domain is list of objects together with operations allows on them
  - Object represented by its name or address, called a **capability**
  - Execute operation M on object Oj, process requests operation and specifies capability as parameter
    - Possession of capability means access is allowed
  - Capability list associated with domain but never directly accessible by domain
    - Rather, protected object, maintained by OS and accessed indirectly
    - Like a “secure pointer”
    - Idea can be extended up to applications

- Option 4 – Lock-key
  - Compromise between access lists and capability lists
  - Each object has list of unique bit patterns, called **locks**
  - Each domain as list of unique bit patterns called **keys**
  - Process in a domain can only access object if domain has key that matches one of the locks
Comparison of Implementations

- Many trade-offs to consider
  - Global table is simple, but can be large
  - Access lists correspond to needs of users
    - Determining set of access rights for domain non-localized so difficult
    - Every access to an object must be checked
      - Many objects and access rights -> slow
  - Capability lists useful for localizing information for a given process
    - But revocation capabilities can be inefficient
  - Lock-key effective and flexible, keys can be passed freely from domain to domain, easy revocation

- Most systems use combination of access lists and capabilities
  - First access to an object -> access list searched
    - If allowed, capability created and attached to process
      - Additional accesses need not be checked
    - After last access, capability destroyed
    - Consider file system with ACLs per file
Access Control

- Protection can be applied to non-file resources

- Solaris 10 provides role-based access control (RBAC) to implement least privilege
  - *Privilege* is right to execute system call or use an option within a system call
  - Can be assigned to processes
  - Users assigned *roles* granting access to privileges and programs
    - Enable role via password to gain its privileges
  - Similar to access matrix
Role-based Access Control in Solaris 10

- User 1
  - Role 1
    - Privileges 1
    - Privileges 2
  - Executes with role 1 privileges
  - Process
Revocation of Access Rights

- Various options to remove the access right of a domain to an object
  - Immediate vs. delayed
  - Selective vs. general
  - Partial vs. total
  - Temporary vs. permanent

- **Access List** – Delete access rights from access list
  - Simple – search access list and remove entry
  - Immediate, general or selective, total or partial, permanent or temporary

- **Capability List** – Scheme required to locate capability in the system before capability can be revoked
  - Reacquisition – periodic delete, with require and denial if revoked
  - Back-pointers – set of pointers from each object to all capabilities of that object (Multics)
  - Indirection – capability points to global table entry which points to object – delete entry from global table, not selective (CAL)
  - Keys – unique bits associated with capability, generated when capability created
    - Master key associated with object, key matches master key for access
    - Revocation – create new master key
    - Policy decision of who can create and modify keys – object owner or others?
Capability-Based Systems

- **Hydra**
  - Fixed set of access rights known to and interpreted by the system
    - i.e. read, write, or execute each memory segment
    - User can declare other *auxiliary rights* and register those with protection system
    - Accessing process must hold capability and know name of operation
    - *Rights amplification* allowed by trustworthy procedures for a specific type
  - Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights
  - Operations on objects defined procedurally – procedures are objects accessed indirectly by capabilities
  - Solves the *problem of mutually suspicious subsystems*
  - Includes library of prewritten security routines

- **Cambridge CAP System**
  - Simpler but powerful
  - **Data capability** - provides standard read, write, execute of individual storage segments associated with object – implemented in microcode
  - **Software capability** - interpretation left to the subsystem, through its protected procedures
    - Only has access to its own subsystem
    - Programmers must learn principles and techniques of protection
Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.

- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.

- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.
Protection is handled by the Java Virtual Machine (JVM)

A class is assigned a protection domain when it is loaded by the JVM

The protection domain indicates what operations the class can (and cannot) perform

If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library
## Stack Inspection

<table>
<thead>
<tr>
<th>protection domain:</th>
<th>untrusted applet</th>
<th>URL loader</th>
<th>networking</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket permission:</td>
<td>none</td>
<td>*.lucent.com:80, connect</td>
<td>any</td>
</tr>
<tr>
<td>class:</td>
<td>gui:</td>
<td>get(URL u):</td>
<td>open(Addr a):</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
|                    |     get(url);   |     doPrivileged {
|                    |     open(addr); |         open('proxy.lucent.com:80');
|                    |     ...         |     }       |
|                    | <request u from proxy> |            |     checkPermission (a, connect);
|                    |                 |             |     connect (a);
|                    |                 |             |     ...    |
End of Chapter 13