# System Protection

Michael Brockway

November 16, 2016

# Contents

- What is Protection? - Principles
- Domains of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection

References

- based on Operating System Concepts (8th Ed), Silberschatz et al, chapter 14. You are recommended to read this chapter of the module textbook.

# What is Protection

Model: computer as a collection of objects, hardware, software

Each object has a unique name and can be accessed through a well-defined set of operations

The *Protection problem* is to ensure that each object is accessed correctly and only by those processes that are allowed to do so.

# Principles of Protection

*Principle of least privelege*

- Programs, users, systems be given just enough privileges (and no more) to perform their tasks
- limits damage if the entity has a bug or is abused/misused accidentally/on purpose
- can be *static* during life of system or during life of process
- or *dynamic* (changed by process as needed): eg domain switching, privilege escalation
- A "Need to know" concept regarding access to data

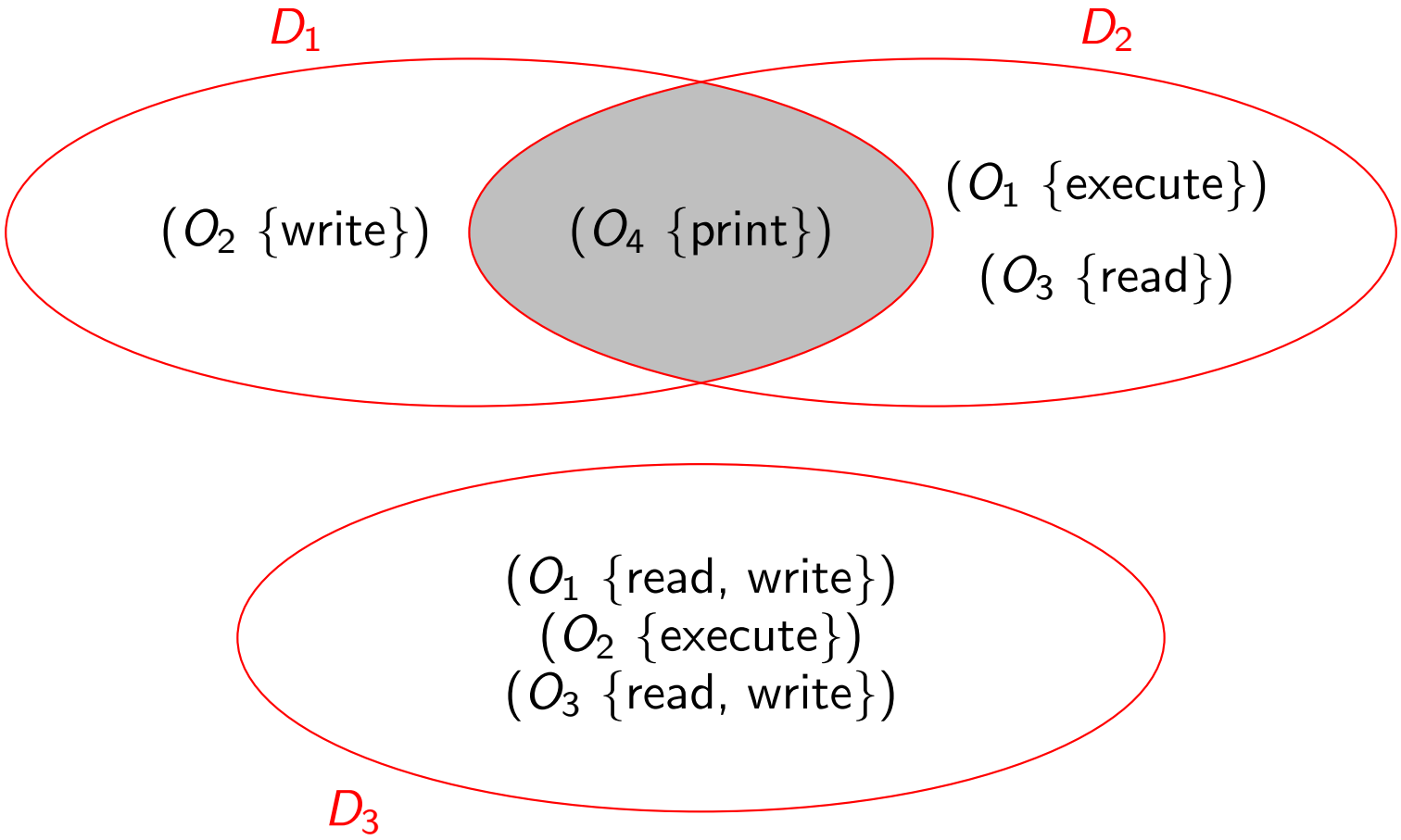
Can be

- Coarse-grained: priveleged given in large chunks: eg "admin" versus "ordinary user" rights
- ... or fine-grained: every file, program, has a set of permissions for every entity that may access it: more work for the system but more protective.

# Domains

- A *domain* of protection is a set of *access rights* where
- an *access right* is a pair consisting of an *object* and a *rights set*
  - A set of permissions governing access to the object
- User domains, process domains, ....
- Each user or process associated with the domain has access to each object (file, program, system resource) determined by the rights set.

# Domains - example



# Domains in 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 = 1, then user-id is set to owner of the file being executed
  - When execution completes user-id is reset
- Domain switch is protected with passwords
  - `su` command temporarily switches to another user's domain when the other domain's password is provided.
- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain, if original domain has the privilege or if password given.

# Access Matrix

Protection by domains can be presented as a matrix:

- one row per domain
- one column per object
- $access[i, j]$ = set of operations a process executing in domain $D_i$ can invoke on object $O_j$

Example

| | $O_1$ (file) | $O_2$ (file) | $O_3$(file) | $O_4$ (printer) |
|---|---|---|---|---|
| $D_1$ | | write | | print |
| $D_2$ | execute | | read | print |
| $D_3$ | read, write | execute | read, write | |
| $D_4$ | | read | execute | |

# Access Matrix, ctd

- If a process in domain $D_i$ tries to do $op$ on object $O_j$, then need $op \in access[i, j]$
- User who creates object can define access column for that object
- Access matrix approach 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

# Use of Access Matrix

Access matrix approach can be expanded to *dynamic* protection

- operations to add, delete access rights
- Special access rights:
  - owner of $O_i$
    - If $D_k$ owns $O_i$ (owner $\in$ $access[k, j]$) then a $D_k$ process may grant or revoke permissions anywhere in column $j$.
  - copy *op* permission from row $D_i$ to $D_j$
    - denoted by "*" below
    - limited to column(s) where copy permission given
  - control: $D_i$ can modify $D_j$ access rights
  - switch: from domain $D_i$ to $D_j$
- Copy, Owner apply to any object
- Control applies to a domain object

# Access Matrix - switch, control

Matrix as above, but with domains as objects

| | $O_1$ (file) | $O_2$ (file) | $O_3$ (file) | $O_4$ (prtr) | $D_1$ (dom) | $D_2$ (dom) | $D_3$ (dom) | $D_4$ (dom) |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | | write | | print | | switch | | |
| $D_2$ | exec | | read | print | | | switch | switch control |
| $D_3$ | read write | exec exec | read write | | | | | |
| $D_4$ | | read | exec | | switch | | | |

A process executing in a domain $D$ has permission to switch to domain $D'$ provided 'switch' permission appears in row $D$, column $D'$.

- $D_1 \rightarrow D_2$;
- $D_2 \rightarrow D_3$ or $D_4$;
- $D_4 \rightarrow D_1$

A process in $D_2$ may alter access rights throughout $D_4$.

# Access Matrix - copy

| | $O_1$ (file) | $O_2$ (file) | $O_3$ (file) |
|---|---|---|---|
| $D_1$ | exec | | write* |
| $D_2$ | exec | read* | exec |
| $D_3$ | exec | | |

A $D_1$ process may give write permission on file $O_3$ to a process in another domain. A $D_2$ process may give read permission on file $O_2$ to a process in another domain.

Thus, for instance, ...

| | $O_1$ (file) | $O_2$ (file) | $O_3$ (file) |
|---|---|---|---|
| $D_1$ | exec | | write* |
| $D_2$ | exec | read* | exec |
| $D_3$ | exec | read | |

A copy permission may (or may not) *propagate* – eg read* rather than read in $(D_3, O_2)$

# Access Matrix - owner

| | $O_1$ (file) | $O_2$ (file) | $O_3$ (file) |
|---|---|---|---|
| $D_1$ | exec<br>owner | | write |
| $D_2$ | | read*<br>owner | read*<br>owner<br>write |
| $D_3$ | exec | | |

If owner $\in$ *access*$[D, O]$ then a process running in $D$ may grant or revoke any permission in column $O$. Thus, for instance,

| | $O_1$ (file) | $O_2$ (file) | $O_3$ (file) |
|---|---|---|---|
| $D_1$ | exec<br>owner | | write |
| $D_2$ | | read*<br>owner<br>write* | read*<br>owner<br>write |
| $D_3$ | | write | write |

# Access Matrix - Implementation

- Generally, a *sparse* matrix
- One option - a global table
  - Store triples (domain, object, rights-set) in a single table
  - A requested operation M on object $O$ within domain $D \rightarrow$ search table for $(D, O, R)$ with $M \in R$.
  - The table probably too large to keep in memory
  - Managing groups of object with same permissions unwieldy - eg consider an object that all domains can read.
- Alternativey, keep *access lists* for objects
  - Each column implemented as an access list for one object
  - Each resulting per-object list is a set of pairs (domain, rights-set) defining all domains with non-empty set of access rights for the object
  - Easily extended to contain a 'default set': if $M \in$ default set, also allow access
  - Each column is an access control list for one object; each row a *capability list* for one domain.

# Access Matrix - Implementation ctd

- Another option: A capability list for domains
  - domain based rather than object-based
  - A *capability list* for a domain is list of pair (object $O$, $\{$operations allowed $O\}$)
  - To execute operation M on object $O$, a process requests the operation and specifies capability as a parameter. Possession of capability $\Rightarrow$ access is allowed
  - Capability list associated with domain but never directly accessible by domain
    - Rather, it is protected object, maintained by OS and accessed indirectly
    - Like a "secure pointer"
    - The approach can be extended to applications
- Fourth option: Lock-key
  - Compromise between access lists and capability lists
  - Each object has list of unique bit patterns, *locks*
  - Each domain as list of unique bit patterns, *keys*
  - Process in domain can access object $\Leftrightarrow$ domain has key that matches one of the locks

# Comparison of Implementations

Trade-offs ...

- 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: 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: on First access to an object, the access list searched

- If allowed, capability created and attached to process; additional accesses need not be checked
- After last access, capability destroyed
- Example: File system with Access Lists per file

# Revocation of Access

Options

- 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: A scheme is 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
- Indirection: capability points to global table entry which points to object: delete entry from global table, not selective
- Keys: unique bits associated with capability, generated when capability created

# Language-based Protection

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

- The language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.
- The implementation interprets protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.
- Protection can be specified by a *declaration*: eg in Java, `protected`, `private`, `synchronized`; thread-safe collection (and other) classes

Ref: Silberschatz et al §14.9

# Protection in Java

The Java Virtual Machine (JVM) has built-in protection mechanisms

- A class is assigned a protection domain when it is loaded by the JVM, depending (configuably) on source URL and any accompanying digital signatures.
- The protection domain indicates what operations the class can (and cannot) perform.
- Eg a class file from a trusted server may be allowed access to user's home directory, while a class from an untrusted server may be granted no file system access at all.

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. The Java philosophy is that the library must be sufficiently priveleged for the this. The library method *takes responsibility* - presumably, also performing whatever checks are necessary to ensure it is safe.

# Protection in Java - Stack Inspection

- Every thread in the JVM has a subroutine call stack.
- When a caller may not be trusted a method executes an access request within a `doPrivileged` block;
- `doPrivileged()` is a static method of class `AccessController`, passed a class with a `run()` to invoke;
- On entry to `doPrivileged {... }` the method's stack frame is annotated to indicate grant of access and the block executed;
- When a protected resource is subsequently requested by this method or one that it calls, a call to `checkPermissions()` invokes a subroutine call stack inspection ...
- Stack frames are inspected from most recent $\rightarrow$ older;
- If a frame with `doPrivileged` annotation found, `checkPermissions()` returns silently; if (otherwise) access is *dis*allowed, an texttttAccessControlException is thrown.

# Protection in Java - Stack Inspection - example

| protection domain: | untrusted applet | URL loader | networking |
|---|---|---|---|
| socket permission: | none | *.lucent.com:80, connect | any |
| class: | gui:<br>...<br>get(url);<br>open(addr);<br><br>... | get(URL u):<br>...<br>doPrivileged {<br>open<br>('proxy.lucent.com:80');<br>}<br>⟨request u from proxy⟩<br>... | open(Addr a):<br>...<br>checkPrmsn<br>(a, connect);<br><br>connect(a);<br>... |

`gui()` is a method of a class in the *untrusted applet* domain

- `get(url)` is a method of a class in the *URL loader* domain, which is permitted to `open(...)` sessions to sites at `lucent.com` for retrieving URLs. This will succeed because `checkPermissions()` in the network library verifies that that `get()` performs `open(...)` in a `doPrivileged` block.

# Protection in Java - Stack Inspection - example

- However, the untrusted applet's `open(...)` will throw an exception, because `checkPermissions()` will find no `doPrivileged` annotation in the stack frame of the `gui()` method.