Protection Mechanisms

PROTECTION:
Set of *mechanisms* to ensure *security* of system
Protection Mechanisms

**Protection:**
Set of *mechanisms* to ensure *security* of system

**Two basic security issues:**
- **Internal security (access permission):**
  Establishing legality of an agent’s access to some object.
- **External security (authentication):**
  Establishing agent’s identity:
  ➔ Login authentication
  ➔ Authentication of remote agents (requires crypto)
Protection Mechanisms

Protection:
Set of *mechanisms* to ensure *security* of system

Two Basic Security Issues:
- **Internal security (access permission):** Establishing legality of an agent’s access to some object.
- **External security (authentication):** Establishing agent’s identity:
  - Login authentication
  - Authentication of remote agents (requires crypto)

Here we concentrate on the issue of access rights.
## Access matrix model

<table>
<thead>
<tr>
<th>Agents</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$O_3$</th>
<th>$O_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>terminate</td>
<td>wait, signal,</td>
<td>read</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>wait, signal, terminate</td>
<td></td>
<td></td>
<td>read, execute write</td>
</tr>
<tr>
<td>$S_3$</td>
<td>wait, signal, receive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_4$</td>
<td>control</td>
<td></td>
<td>execute</td>
<td>write</td>
</tr>
</tbody>
</table>

Used by protection system to determine whether access is allowed
Access matrix properties

- Rows define agents’ *protection domains*
- Columns define objects’ *accessibility*
- Dynamic data structure: frequent
  - permanent changes (e.g. chmod)
  - temporary changes (e.g. setuid)
- Very *sparse* with many repeated entries
- Usually not stored explicitly.
Issues for Protection System Design

- Propagation of rights:
  ➔ Can agent grant access to another?
- Restriction of rights:
  ➔ Can agent propagate restricted rights?
- Revocation of rights:
  ➔ Can access, once granted, be revoked?
- Amplification of rights:
  ➔ Can unprivileged agent perform restricted operations?
- Determination of object accessibility:
  ➔ Which agents have access?
  ➔ Is object accessible at all (garbage collection)?
- Determination of agent’s protection domain:
  ➔ Which objects are accessible?
Access matrix implementation: ACLs

Represent column-wise: *access control list* (ACL):

- ACL associated with object.
  - Propagation: meta-right (e.g., *owner* can *chmod*)
  - Restriction: meta-right
  - Revocation: meta-right
  - Amplification: protected-invocation right (e.g., *setuid*)
  - Accessibility: explicit in ACL
  - Protection domain: hard (if not impossible)
Access matrix implementation: ACLs

Represent column-wise: *access control list* (ACL):

- ACL associated with object.
  - Propagation: meta-right (e.g., *owner* can *chmod*)
  - Restriction: meta-right
  - Revocation: meta-right
  - Amplification: protected-invocation right (e.g., *setuid*)
  - Accessibility: explicit in ACL
  - Protection domain: hard (if not impossible)

- Usually condensed via *domain classes* (UNIX groups)
- Full ACLs used by Multics, Apollo Domain, Andrew FS, NT.
Access matrix implementation: ACLs

Represent column-wise: access control list (ACL):

- ACL associated with object.
  - Propagation: meta-right (e.g., owner can chmod)
  - Restriction: meta-right
  - Revocation: meta-right
  - Amplification: protected-invocation right (e.g., setuid)
  - Accessibility: explicit in ACL
  - Protection domain: hard (if not impossible)

- Usually condensed via domain classes (UNIX groups)

- Full ACLs used by Multics, Apollo Domain, Andrew FS, NT.

- Can have negative rights, to:
  - reduce “window of vulnerability”,
  - simplify exclusion from groups.

- Sometimes implicit (process hierarchy).
Access matrix implementation: Capabilities

Represent row-wise: *capabilities*

- *Capability list* associated with agent.
- Each capability confers a certain right to its holder.
  - ➔ Propagation: copy capabilities between agents (how?)
  - ➔ Restriction: lesser rights require new ("derived") capabilities
  - ➔ Revocation: requires invalidation of capabilities from *all agents*
  - ➔ Amplification: special invocation capability.
  - ➔ Accessibility: requires inspection of all capability lists (how?)
  - ➔ Protection domain: explicit in capability list.
Access matrix implementation: Capabilities

Represent row-wise: *capabilities*

- *Capability list* associated with agent.
- Each capability confers a certain right to its holder.
  - Propagation: copy capabilities between agents (how?)
  - Restriction: lesser rights require new (“derived”) capabilities
  - Revocation: requires invalidation of capabilities from *all agents*
  - Amplification: special invocation capability.
  - Accessibility: requires inspection of all capability lists (how?)
  - Protection domain: explicit in capability list.
- Can have *negative rights*, to:
  - reduce “window of vulnerability”,
  - simplify management of groups of capabilities.
- Successful commercial system: IBM System/38 *et fils* [Sol97]
- Popular among research distributed OS.
Capabilities

- Main advantage of capabilities is the fine-grain access control:
  ➔ Easy to provide specific access to selected agents.
Capabilities

- Main advantage of capabilities is the fine-grain access control:
  ➔ Easy to provide specific access to selected agents.
- Capability presents *prima facie* evidence of the right to access:
  ➔ capability ⇒ *object identifier* (naming),
  ➔ capability ⇒ (set of) *access rights*,
Capabilities

- Main advantage of capabilities is the fine-grain access control:
  - Easy to provide specific access to selected agents.
- Capability presents *prima facie* evidence of the right to access:
  - capability ⇒ *object identifier* (naming),
  - capability ⇒ (set of) *access rights*,
  ⇒ Any representation must contain object ID and access rights.
  ⇒ Any representation must protect capability from forgery.
Capabilities

- Main advantage of capabilities is the fine-grain access control:
  ➔ Easy to provide specific access to selected agents.
- Capability presents *prima facie* evidence of the *right to access*:
  ➔ capability ⇒ *object identifier* (naming),
  ➔ capability ⇒ (set of) *access rights*,
  ⇒ Any representation must contain object ID and access rights.
  ⇒ Any representation must protect capability from forgery.
- How implemented and protected?
Capabilities

- Main advantage of capabilities is the fine-grain access control:
  ➔ Easy to provide specific access to selected agents.
- Capability presents *prima facie* evidence of the *right to access*:
  ➔ capability ⇒ *object identifier* (naming),
  ➔ capability ⇒ (set of) *access rights*,
  ➔ Any representation must contain object ID and access rights.
  ➔ Any representation must protect capability from forgery.
- How implemented and protected?
  ➔ *tagged* (protected by hardware),
  ➔ *partitioned* (protected by software),
  ➔ *sparse* (protected by obscurity).
Tagged Capabilities

- *Tag bit(s) with every (group of) memory word(s):*
  - Tags identify capabilities.
  - Capabilities are used like “normal” pointers.
  - Hardware checks permissions on dereferencing capability.
  - User code can copy capabilities.
  - Modifications turn tags off.
  - Only privileged instructions (kernel) can turn tags on.
Tagged Capabilities

- Tag bit(s) with every (group of) memory word(s):
  - Tags identify capabilities.
  - Capabilities are used like “normal” pointers.
  - Hardware checks permissions on dereferencing capability.
  - User code can copy capabilities.
  - Modifications turn tags off.
  - Only privileged instructions (kernel) can turn tags on.
    - Propagation easy.
    - Restriction requires kernel to make new capability.
    - Revocation virtually impossible (memory scan!)
    - Amplification possible (see below).
    - Accessibility impossible to determine.
    - Protection domain difficult to establish.
Tagged Capabilities

-Tag bit(s) with every (group of) memory word(s):
  - Tags identify capabilities.
  - Capabilities are used like “normal” pointers.
  - Hardware checks permissions on dereferencing capability.
  - User code can copy capabilities.
  - Modifications turn tags off.
  - Only privileged instructions (kernel) can turn tags on.
    - Propagation easy.
    - Restriction requires kernel to make new capability.
    - Revocation virtually impossible (memory scan!)
    - Amplification possible (see below).
    - Accessibility impossible to determine.
    - Protection domain difficult to establish.

- IBM System/38 [Ber80], AS/400 [Sol97], many historical systems.
PROTECTED PROCEDURE CALL (AS/400)

- AS/400 has a segmented memory architecture.
- Capabilities confer rights over segments.
- Capabilities can confer invocation rights.
- Each user has a profile, which is essentially a capability list.
- Capabilities can be of profile adoption type:
  - On invocation, segment owner’s profile is added to caller’s protection domain.
  - Normal pointers can be dereferenced if the profile contains appropriate capabilities.
  - On return, profile adoption is cancelled.
  - User can denote subset of their profile to be used in adoption (profile propagation).
TAGGED CAPABILITIES OUTSIDE RAM

- Disk has no tags.
- AS/400 page size is 4kB.
- Physical disk blocks are 520B, logical blocks 512B.
- Extra 64B per page store tag bits (among others).
  - On page-out page must be scanned and all tags collected.
  - On page-in all tags must be reconsituted.
  - Significant processing overhead with all I/O.
TAGGED CAPABILITIES SUMMARY

- Secure through hardware protection.
- Convenient for applications (appear as “normal” pointers).
- Checked by hardware $\Rightarrow$ fast validation.
- Hardware solution is not for everyone.
- Capability hardware is complex (and slow?)
- Separate mechanisms required for I/O and distribution.
Partitioned Capabilities

- System maintains capability list (clist) with each process (in PCB).
  - User code uses indirect references to capabilities (clist index).
  - System validates access via clist when mapping any page.
Partitioned Capabilities

- System maintains capability list (clist) with each process (in PCB).
  - User code uses indirect references to capabilities (clist index).
  - System validates access via clist when mapping any page.
    - Validation is implicit at page fault or explicit mapping time.
    - Propagation: system intervention to copy between clists.
    - Restriction: kernel to make new capability.
    - Revocation: kernel to remove cap from all (or specific) clists.
    - Accessibility can only be determined by scanning all clists.
    - Protection domain is explicitly represented in clist.
Partitioned Capabilities

- System maintains capability list (clist) with each process (in PCB).
  - User code uses indirect references to capabilities (clist index).
  - System validates access via clist when mapping any page.
    - Validation is implicit at page fault or explicit mapping time.
    - Propagation: system intervention to copy between clists.
    - Restriction: kernel to make new capability.
    - Revocation: kernel to remove cap from all (or specific) clists.
    - Accessibility can only be determined by scanning all clists.
    - Protection domain is explicitly represented in clist.

- Hydra [CJ75], Mach [RTY+88], KeyKOS [BFF+92], Grasshopper [DdBF+94], Eros [SSF99] and many others.
PROPAGATING PARTITIONED CAPABILITIES (MACH):

- Capabilities can be propagated via IPC.
  1. User must insert capabilities (clist indices) into special field in message.
  2. Kernel looks up clists and inserts representation of “real” capability (*marshaling*).
  3. Receiver’s kernel inserts capabilities into receiver’s clist.
  4. Kernel replaces capability in message by clist index.
PROPAGATING PARTITIONED CAPABILITIES (MACH):

- Capabilities can be propagated via IPC.
  1. User must insert capabilities (clist indices) into special field in message.
  2. Kernel looks up clists and inserts representation of “real” capability (*marshaling*).
  3. Receiver’s kernel inserts capabilities into receiver’s clist.
  4. Kernel replaces capability in message by clist index.

- Can be simplified if IPC is local.

- Amplification can be performed by schemes similar to AS/400.
PARTITIONED CAPABILITIES SUMMARY

- Secure through kernel protection.
- Validation at mapping time ⇒ apps use “normal” pointers.
- Fast validation (clist check is simple, validation cached by MMU).
- Propagation requires marshaling and kernel intervention.
- Reference counting possible to detect unaccessible objects.
Sparse Capabilities

Basic idea similar to encryption:

• Add bit-string to make valid capabilities a very small subset of the capability space.
• Can be encrypted object info or something like a password.
• Capabilities are pure user-level objects, which can be passed around like other data.
• Appropriate for user-level servers.
**Example: Signature capabilities**

“First Migration Scheme” [GL79], designed to allow migration of tagged capabilities in distributed systems.

- tamper proof via encryption with secret kernel key
- can freely be passed around
- need to decrypt on each validation
- users do not know which object capability refers to

- $f$: one-way function (secure digest), $E$: encryption function
“Second Migration Scheme” [GL79]

Object ID Access rights

\[ E(K, C) \] Encrypted capability

Object ID Access rights Signature

Object ID visible, yet still tamper proof.
AMOEBA’S CAPABILITIES

Appropriate for user-level servers [MT86].
Properties of Amoeba Capabilities

- Port identifies server.
  ➔ Kernel resolves server and caches server location.
- Port IDs are large (48-bit) sparse numbers.
  ➔ Knowledge implies send rights.
- Creator ("owner") has all rights.
- Server uses OID to look up rights, checks fields to validate.
PROPERTIES OF AMOEBA CAPABILITIES

- Port identifies server.
  ➔ Kernel resolves server and caches server location.
- Port IDs are large (48-bit) sparse numbers.
  ➔ Knowledge implies send rights.
- Creator (“owner”) has all rights.
- Server uses OID to look up rights, checks fields to validate.
  ➔ Validation done by user-level server when invoked.
  ➔ Propagation easy, as capabilities are “normal” data.
  ➔ Restriction requires server to make new capability.
  ➔ Revocation done by server removing entry from object table.
  **But** not very helpful if only one capability per access mode.
  ➔ Amplification possible according to server policies.
  ➔ Accessibility is impossible to determine.
  ➔ Protection domain is impossible to determine.
**AMOEBA RIGHTS RESTRICTION**

- Used by server to derive lesser capabilities on request.
- No need to store derived capability in object table.
IMPROVED VERSION (NOT IMPLEMENTED)

- Set of *commuting* one-way functions $f_i$, one for each access mode bit:
  \[ f_i(f_j(x))) = f_j(f_i(x)). \]
- To remove access mode $i$, obtain new check field as:
  \[ C' = f_i(C). \]
- Can be done by user without server intervention.
SERVER AUTHENTICATION: F-BOXES

• Hardware device “F-box” at each network connection
  ➔ When requesting messages for port $G$, F-box will only accept messages destined for port $P = f(G)$, where $f$ is a one-way function
  ➔ Server publishes $P$ as port ID
  ➔ Intruder who does not know $G$ cannot access messages
• Scheme depends on physical security of F-boxes (or their implementation in the OS).
• Never been implemented (to my knowledge).
Password capabilities

- Used in the Monash Password Capability System [APW86], Opal [CLFL94], Mungi [HEV+98].
PROPERTIES OF PASSWORD CAPABILITIES

- Passwords must be protected (eavesdropping, Trojan horses).
- Separate passwords for different rights (good idea to package rights with caps).
- No encryption ⇒ easy to validate.
  - Validation done by kernel on access or presentation and cached by MMU.
  - Propagation easy, as capabilities are “normal” data.
  - Restriction requires kernel to make new capability.
  - Revocation done by kernel removing entry from object table.
  - Amplification possible similar to AS/400.
  - Accessibility is impossible to determine.
  - Protection domain is known to kernel.
SPARSE CAPABILITIES SUMMARY

- Statistically secure (like encryption).
- Validation at mapping time \(\Rightarrow\) applications can use “normal” pointers.
- Validation may be slow, but kernel and MMU can cache.
- No kernel intervention required on most operations.
- Reference counting impossible to detect inaccessible objects.
References


