Outline

- Announcements
  - Lab 6 demos scheduled for April 26th and 27th
  - Review session on May 1st
- Protection
  - Access matrix: Additional operations
  - Rights revocation
- Language-based protection
  - language-based protection mechanisms
  - secure-services using language-based protection
- Security
  - the security problem
  - authentication

[Silberschatz/Galvin: Chapters 19, 20]

Access Matrix Operations: Domain Migration

- Uniform treatment of domains and objects
  - domains are also treated as objects
  - The corresponding entry \((i, j)\) specifies if a switch is permitted from domain \(i\) to domain \(j\)

- Example

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>F2</th>
<th>printer</th>
<th>D1</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td>s</td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td>pr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>r</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>rw</td>
<td>rw</td>
<td></td>
<td></td>
<td>s</td>
</tr>
</tbody>
</table>

Additional Access Matrix Operations (Rights)

- Copy
  - read* means read right can be copied to another domain

\[
\begin{array}{cccccc}
F1 & F2 & \text{printer} & D1 & D2 \\
\hline
D1 & r &     &       & s \\
D2 &     & pr &       &   \\
D3 & r^* & r  &       &   \\
D4 & rw & rw &       & s \\
\end{array}
\]

- variants
  - copy-and-delete (cut)
  - copy one level

- Owner: create and remove rights associated with a particular object
- Control: permit modifications of a domain’s rights by a process in another domain
### Revocation of Access Rights

Several dimensions

- **Immediate revocation**
  - Can be delayed sometimes
- **Selective revocation (members of the domain)**
  - Not always possible and might be an "all-or-nothing" choice
- **Partial revocation (rights of the object)**
- **Temporary or permanent**
  - This requires the possibility revoking the rights for some time and then reinstating them

### Revocation

- **With Access Lists**
  - Easy to implement any of the above variants
  - Just scan the list and modify
  - Revocation is immediate

- **With Capability Lists**
  - A much more difficult revocation problem
  - Indexed by domains
    - Therefore, an object can be scattered across multiple domains
  - An update to the access rights of an object must therefore involve an exhaustive search

### Some Solutions

- **Periodic reacquisition** of capabilities
  - Updates are periodic and so revocations are not visible immediately

- **Back pointers**: A list of pointers connect an object to all the domain capabilities
  - This solution is quite costly

- Cheaper solution: The capabilities point to the object indirectly via a global "object table" which can be quickly updated to change things
  - Empty table entries imply illegal accesses

### Language-based Protection
Language-based Protection: Rationale

- Protection solutions discussed so far require **involvement of OS kernel**
  - To validate access rights or capabilities at run time
  - Tend to be high overhead
  - Also, inflexibility in the objects and operations being protected
    - System-defined functions: e.g., file operations
  - Hardware solutions where checks must be performed efficiently
    - E.g., virtual-memory page protection
    - Not generally applicable

- Current-day operating systems need additional flexibility
  - Arbitrary (user-defined) objects and operations
    - E.g., a server might want to restrict access to certain services
  - Need for protection checks to be performed efficiently

- Solutions involve **language features** and their **trusted implementation**

Language-based Protection: Solution

- **Shared resources/services defined as objects**
  - Abstract data types
  - Protection enforced on access to an object’s methods

- **Goal of protection**: Ensure that only allowed methods are invoked
  - Definition of **allowed** depends on context
    - More generally, the software module that is doing the call

- **Object-oriented languages** already provide such support …
  - … for enforcing abstraction boundaries
    - E.g., **private** variables and methods are not accessible outside their class
    - Enforced statically by the compiler (e.g., Java)
    - Sometimes, also requires run-time support (e.g., C++)

- **General problem** is known as **type safety**
  - Languages where type-based access is enforceable: **safe languages**

Language-based Protection: Two Issues

- “**Memory**” protection
  - A program will not access memory or execute code for which it is not authorized
  - Builds on top of language type safety

- Secure system services
  - A program will not access unauthorized system services (files, graphics, …)
  - Type safety is not sufficient
    - Type safety performs **local checks** at the call interface
    - Not convenient to enforce **global properties** such as:
      - Program A can access all files, while program B can only access /tmp
      - Particularly when programs access resources using intermediate modules

Solutions for “**Memory**” Protection

- **Type safety**
  - Compiler/run-time system enforces that only allowed methods are being invoked
    - Based on “**type**” of the object handle
  - In Java, the **byte-code verifier** traces all control flows and can verify that the byte code obeys the restrictions of the Java type system

- An alternative scheme: **Software fault isolation**
  - Instrument the program binary with code that performs checks at run-time
  - Example checks:
    - From certain functions, only addresses in a specific range are accessed
    - Only certain functions are called
  - Requires a fair bit of work to limit overhead
Solutions for Secure System Services

Scheme 1: Software capabilities

- A way of ensuring that pointers to objects cannot be forged
  - Ensured in hardware capability systems using tagged memory
- A precondition for type-safety based protection
- Only way of obtaining a software capability is to be explicitly given it
  - Either as part of initialization
  - Or as a result of calling another capability
- In Java, a capability is just an object reference
  - Type safety prevents an object reference from being forged
  - WHY?
- Problem: How do you distinguish between two software modules holding the same reference?

Scheme 2: Stack Introspection

- Access to services/methods restricted based on program call chain
  - Run-time examination of the stack
  - Typically, from newest to oldest
- Typical interface
  - EnablePrivilege: creates an enabled privilege based on user
  - DisablePrivilege: discards the enabled privilege
  - checkPrivilege: searches the stack to see if an enabled privilege exists
- In our example
  - Program A's call to the intermediate software module could create an enabled privilege
  - This privilege is later checked by the file system access module

Scheme 3: Name space management

- Idea: Enforce protection by controlling how names in a program are resolved into runtime classes
  - In our example, ensure that in Program B’s context, the filesystem class actually resolves to a different object that only permits access to /tmp files
- Implementation of name space management in Java
  - Modify the Java ClassLoader
    - So that it searches for class implementations in a program-specific fashion

Language-based Protection: Pros and Cons

- Pros
  - Precision and flexibility: Only some software modules can have privileges
  - Rights amplification: As required, rights can be increased/decreased
  - E.g., an object’s method can have access to its private members
  - Security: Static checking can enforce more general security policies
  - E.g., a lock that is being acquired will always be released
  - Efficiency
- Cons
  - Need for a trusted compiler and run-time system
  - Particularly, in the presence of multiple sources
  - Single language restriction
  - Garbage collection: What happens to deallocated pointers?
  - Revocation
  - Performance of regular code
Security

Protection vs. Security

- Protection is concerned with attack from within
- Security is concerned with attack from outside
  - The motivations can be malicious or accidental
  - Typical concerns are unauthorized
    - access to ...
    - modification of ...
    - destruction of ...
  - ... information

Domains of Security

- Typically
  - physical
    - includes the protection of the system from physical attack
  - human
    - access must be limited based on specific authorizations

- Degree vs. Cost
  - investment in security measures depends on sensitivity of data
    - e.g. software theft, exam peeking, money theft

Authentication

- Protection mechanisms
  - depend on ability to identify executing programs and processes
  - typically associated with the user ID

- How does the system identify (authenticate) legitimate users?
  - done popularly via passwords
  - also via more elaborate schemes
    - physical characteristics: e.g. handwriting (more generally, biometrics)
    - location: e.g. a particular terminal
    - possession of a key or a card

- Levels
  - single-level: one-time authentication (at first application)
  - multiple-levels: authentication required for sub-systems
Problems with Passwords

- Passwords are subject to being
  - guessed
  - systematic repetitive testing (all words in the system dictionary)
  - observed
  - over someone’s shoulder
  - over the network: several programs transmit passwords in plain-text format
  - stolen
  - e.g. fake “login” program
- Passwords must be stored somewhere
  - this file or program must also be secure

Password Encryption

- Technical Help: One-way functions
  - there exist functions $f(x)$ such that
    - computing their inverse is impossibly time-consuming
  - example
    - $f(x) = \text{permutebits}(x) \mod p$
    - $p$ is a prime number
- Unix uses an encryption function $f$ (f may be known)
  - for each user $U$ with password $P$
    - the OS stores $(U, f(P))$ in the password file
  - an attacker may
    - discover the code for $f$
    - break into the password file
  - … but still not be able to compute any of the $P$s
    - by reversing the encryption process

Management Precautions

- However, encryption does not prevent somebody from guessing the password and checking if the guess is correct
  - By encrypting the guessed password
- To reduce chance of password discovery
  - system management may hide the password entries
  - restrict space of passwords
    - require long passwords involving at least one numeric character
    - disallow use of dictionary words
  - require passwords to be changed frequently
- More elaborate scheme: Adding “salt” to a password
  - prepend each password with a random number before encrypting
    - May or may not be stored in the password file
    - difficult to guess the random number
    - works well for cross-network attacks (e.g., rsh)

Next Lecture

- Security (contd.)
  - one-time passwords
  - encryption schemes
  - program and system threats
  - example: Windows NT

Reading

- Silberschatz/Galvin: Chapter 20