(Review) 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
  - Are inflexible in the objects and operations being protected
    - System-defined functions: e.g., file operations
    - Require hardware support for efficiency
      - E.g., virtual-memory page protection
      - Not generally applicable

- Current-day OSes need additional flexibility
  - To protect arbitrary (user-defined) objects and operations
    - E.g., a server might want to restrict access to certain services
  - Need these checks to be performed efficiently

- Solutions involve language features and their trusted implementation

Outline

- Announcements
  - Lab 6 due back on May 6th
  - Final exam on May 9th, 2:00 – 3:50pm, 101 WWH
    - Review on May 6th

- Language-based Protection
  - rationale
  - language-based protection mechanisms
    - examples: software-fault isolation (SFI), type safety, proof-carrying code
  - secure-services using language-based protection
    - capabilities, stack introspection, namespace management
  - pros and cons

- Security

[Silberschatz/Galvin/Gagne: Section 18.7, Chapter 19]

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 at load time that the byte code obeys the restrictions of the Java type system
  - Assumptions
    - Object handle represents the desired set of privileges
      - Note that Java allows a reference to be cast up and down (run-time checks)
    - No way to forge an object reference
      - Byte code verifier only ensures this for Java code running within the JVM
    - Implication: Compiler, Verifier, Run-time system, Operating System, Node hardware are all part of the trusted computing base
  - Generalization: Proof-carrying code

Solutions for “Memory” Protection (cont’d)

An alternative scheme: Software fault isolation

- Instrument the program binary with code that performs checks at run-time
  - Before each load/store instruction, check address
  - Before each call/jmp instruction, check address
- 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?
Solutions for Secure System Services (cont’d)

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

Solutions for Secure System Services (cont’d)

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

Java Security

- Java basics
  - Strongly typed language
  - No arbitrary pointers: references act as capabilities
  - Can only manipulate object in restricted ways
    • E.g., only methods can manipulate private data
- JDK 1.0 (c. 1995)
  - distinguished between trusted and untrusted code (applets)
  - Applets ran in a “sandbox”
    • Lots of restrictions, enforced using methods of the SecurityManager class
    • Only installed for the applet classloader (namespace management)
- JDK 1.1 (c. 1997)
  - Some signed applets can be granted total trust
- Problems: Too little or too much trust

Java Security (cont’d)

- Java 2 permits distinctions at a finer granularity
  - Based on URL from which the class was loaded, any digital signatures …
  - Class placed in a protection domain
    • Privileges of a protection domain specified in a policy file
    • Maintained as run-time state
  - Stack introspection algorithm when method M requires permission P
    • Functionality implemented in SecurityManager and AccessController classes
    • Unwind each frame and check if P is allowed for each protection domain
    • Intersection of permissions
    • doPrivileged tags caller’s frame: algorithm terminates on encountering tag
  - Standard permissions
    • Filesystem, Sockets, System properties
    • Can be extended ….
  - Guarded Objects permit access control for individual object instances
    • Above controls access for all instances of a class
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

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
  - A 1990 study showed that 25% of the passwords of ~14,000 accounts could be obtained by simple guesses
  • Permuting the user/account name
  • Dictionary words
  • Myths and legends, characters from literature …
- 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 Schemes: Password “Salting”

- Concatenate each password with a random number before encrypting
  - Random number typically stored in the clear in the password file

```
crypt(3)  
<table>
<thead>
<tr>
<th>salt</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bits</td>
<td>56 bits</td>
</tr>
</tbody>
</table>
```

Based on DES

What Does the Salt Do?

- Prevents duplicate passwords from being visible in the password file
  - Since different users are likely to have different “salts”

- Increases the effective size of passwords
  - “A 12-bit salt makes it 4096 times harder to crack a password”
  - How true is this statement?
    - If the attacker pre-encrypts a dictionary of guessed passwords
    - If the attacker manages to obtain the password file where the salt is stored in plaintext (as is usually the case)

- Prevents use of hardware implementations of the encryption routine
  - Given current technology, software implementations are limited in how many guesses they can try

More Elaborate Schemes: One-time Passwords

- Definition: A secure hash function \( f \)
  - \( f \) takes a variable length string and produces a fixed-length string
  - Hard to reverse: given output, difficult to determine input
  - Collision-free: hard to find two inputs that hash to same output

- Given a secure hash function \( f \), and a shared secret \( s \), the passwords correspond to
  \[ f(s), f(s^2), f(s^3), \ldots, f(s^N) \]
  - authentication server initially stores \( p_0 \)
  - client sends \( p_{i+1} \), server verifies that \( p_i = f(p_{i+1}) \)
    - Properties of secure hash functions prevent computing the inverse
    - note that the secret \( s \) is never transmitted

- Variants of this scheme (some less robust)
  - Code books: a sequence of passwords, each used exactly once