Operating System Security, Continued
CS 239
Security for Networks and System Software
May 22, 2002

Introduction

- Designing trusted operating systems
- Encapsulated environments

Designing Trusted Operating Systems

- Security professionals tend to speak of *trust,* rather than security, in this context
- A more practical definition of what OS users want
- The user's trust that the OS will provide certain security features properly

Security Policies and Trusted Operating Systems

- A *policy* is a statement of the security we expect the system to enforce
- We trust a system to the degree we believe it properly implements its policy

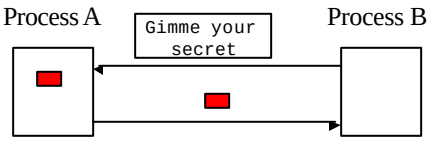
Discretionary and Mandatory Access Control

- Discretionary access control means that the users can choose to enforce it
  - Or not
- Mandatory access control means the system forces access control on the users
  - Whether they like it or not

More on Mandatory Access Control

- Allows higher authorities to control what users do with data they can access
- Can prevent a user from telling a secret to someone who "shouldn't" know it

Returning to Our Example



What if the system authorities don't want A to tell the secret to B?
Can we prevent this?

Why Would We Want To Prevent It?

- What if the secret is proprietary information?
- What if the secret is essentially access to valuable software?
- What if we're concerned that B will be able to fool A?
  - Perhaps via social engineering?
- What if A and B are processes, not people?

Common Security Policies

- Designed to state what we do and don't want to allow
  - Like the previous example
- Military security policies
- Commercial security policies

Military Security Policies

- Based on several *ranks* of security
  - Unclassified
  - Restricted
  - Confidential
  - Secret
  - Top secret
- And compartmentalized by the need to know

Clearances in Military Security

- A *clearance* describes what information a subject can know
- All information has some security label
- A subject can access information only if he has the proper clearance
- A combination of the rank and the compartment allowed

Determining Security Access in Military Models

- Based on a dominance relationship
- A subject dominates an object iff:
  - the subject has a more restricted rank than the object and
  - the subject has access to the all the compartments of the object

Commercial Security Policies

- Typically less rigid and hierarchical than military policies
- But with similar concerns
- Generally more flexibility in setting up levels and compartments
- And in assigning access privileges

Clark-Wilson Security Policy

- Particularly concerned with data integrity
- System designer specifies *well-formed transactions*
- System must guarantee that all permitted operations conform to such transactions

Separation of Duty Security Policy

- To guarantee that important commercial activities are not performed improperly by employees
- Requires active participation by multiple parties to achieve a goal
  - Even if one or more parties is permitted to perform every step

Chinese Wall Security Policy

- Meant to provide strict separation between parts of a company
  - For intellectual property reasons
  - Or to prevent conflicts of interest
- Defines classes of conflicts among different groups in the company
- Subjects cannot access information from more than one class member

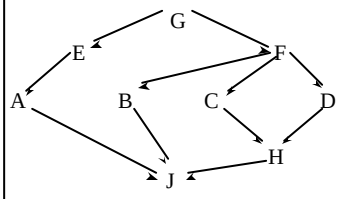
Models of Security

- Lattice model
- Bell-La Padua model
- Many other models exist
  - Some are practical
  - Some are useful for proving theoretical limits of security

Lattice Model of Security

- A generalization of military model
- Elements of the lattice are the security labels of the subjects and objects
- A partial ordering is defined on the lattice elements
- Access is permitted from one element to another if first is "greater" than the second

Example of the Lattice Model



- E can access A, but A can't access E
- A and B cannot access each other
- Everyone can access J
- G can access everyone

Bell-La Padua Confidentiality Model

- Describes allowable paths of information flow in a secure system
- Another formalization of military security model
- Designed for systems that handle data at multiple levels of sensitivity

Important Security Properties for Bell-LaPadua

- Simple security property
- *-Property
- Tranquility property

Simple Security Property

- Subject *s* may have read access to object *o* only if *C(o) <= C(s)*
- Means that I can read any object if I have a higher enough security class
- So the general can listen to what the private says

*-Property

- Subject *s* who has read access to object *o* may have write access to object *p* only if *C(o) <= C(p)*
- Means that I can only write to objects at my security class or higher
- Means the general can't say anything to the private
- Prevents *write-down*

Tranquility Property

- Classification of a subject or object can change
  - But not while the subject is accessing anything
  - Or while the object is being accessed
- Thereby assuring complete mediation

Thinking About This Security Model

- Let's say I want it in my operating system
- How do I get it?
- What are the implications of having it?

Desired Security Features of a Normal OS

- Authentication of users
- Memory protection
- File and I/O access control
- General object access control
- Enforcement of sharing
- Fairness guarantees
- Secure IPC and synchronization
- Security of OS protection mechanisms

Extra Features for a Trusted OS

- Mandatory and discretionary access control
- Object reuse protection
- Complete mediation
- Audit capabilities
- Intruder detection capabilities

How To Achieve OS Security

- Kernelized design
- Separation and isolation mechanisms
- Virtualization
- Layered design

Advantages of Kernelization

- Smaller amount of trusted code
- Easier to check every access
- Separation from other complex pieces of the system
- Easier to maintain and modify security features

Reference Monitors

- An important security concept for OS design
- A *reference monitor* is a subsystem that controls access to objects
  - It provides (potentially) complete mediation
- Very important to get this part right

Assurance of Trusted Operating Systems

- How do I know that I should trust someone's operating system?
- What methods can I use to achieve the level of trust I require?

Assurance Methods

- Testing
- Formal verification
- Validation

Secure Operating System Standards

- If I want to buy a secure operating system, how do I compare options?
- Use established standards for OS security
- Several standards exist

Some Security Standards

- U.S. Orange Book
- European ITSEC
- U.S. Combined Federal Criteria
- Common Criteria for Information Technology Security Evaluation

The U.S. Orange Book

- The earliest evaluation standard for trusted operating systems
- Defined by the Department of Defense in the late 1970s
- Now largely a historical artifact

Purpose of the Orange Book

- To set standards by which OS security could be evaluated
- Fairly strong definitions of what features and capabilities an OS had to have to achieve certain levels
- Allowing "head-to-head" evaluation of security of systems
  - And specification of requirements

Orange Book Security Divisions

- A, B, C, and D
  - In decreasing order of degree of security
- Important subdivisions within some of the divisions
- Requires formal certification from the government (NCSC)
  - Except for the D level

Some Important Orange Book Divisions and Subdivisions

- C2 - Controlled Access Protection
- B1 - Labeled Security Protection
- B2 - Structured Protection

The C2 Security Class

- Discretionary access
  - At fairly low granularity
- Requires auditing of accesses
- And password authentication and protection of reused objects
- Windows NT has been certified to this class

The B1 Security Class

- Includes mandatory access control
  - Using Bell-La Padua model
  - Each subject and object is assigned a security level
- Requires both hierarchical and non-hierarchical access controls

The B3 Security Class

- Requires careful security design
  - With some level of verification
- And extensive testing
- Doesn't require formal verification
  - But does require "a convincing argument"
- Trusted Mach is in this class

~~Logging and Auditing~~

- An important part of a complete security solution
- Practical security depends on knowing what is happening in your system
- Logging and auditing is required for that purpose

Logging

- No security system will stop all attacks
  - Logging what has happened is vital to dealing with the holes
- Logging also tells you when someone is trying to break in
  - Perhaps giving you a chance to close possible holes

Access Logs

- One example of what might be logged for security purposes
- Listing of which users accessed which objects
  - And when and for how long
- Especially important to log failures

Other Typical Logging Actions

- Logging failed login attempts
  - Can help detect intrusions or password crackers
- Logging changes in program permissions
  - Often done by intruders

Problems With Logging

- Dealing with large volumes of data
- Separating the wheat from the chaff
  - Unless the log is very short, auditing it can be laborious
- System overheads and costs

Log Security

- If you use logs to detect intruders, smart intruders will try to attack logs
  - Concealing their traces by erasing or modifying the log entries
- Append-only access control helps a lot here
- Or logging to hard copy
- Or logging to a remote machine

~~Verifying System Security~~

- Security mechanisms are great
  - If you have proper policies to use them
- Security policies are great
  - If you follow them
- For practical systems, proper policies and consistent use are a major security problem

Auditing

- A formal (or semi-formal) process of verifying system security
- "You may not do what I expect, but you will do what I inspect."
- A requirement if you really want your systems to run securely

Auditing Requirements

- Knowledge
  - Of the installation and general security issues
- Independence
- Trustworthiness
- Ideally, big organizations should have their own auditors

When Should You Audit?

- Periodically
- Shortly after making major system changes
  - Especially those with security implications
- When problems arise
  - Internally or externally

Auditing and Logs

- Logs are a major audit tool
- Some examination can be done automatically
- But part of the purpose is to detect things that automatic methods miss
  - So some logs should be audited by hand

A Typical Set of Audit Criteria

- For a Unix system
- Some sample criteria:
  - All accounts have passwords
  - Limited use of setuid root
  - Display last login date on login
  - Limited write access to system files
  - No "." in PATH variables

What Does an Audit Cover?

- Conformance to policy
- Review of control structures
- Examination of audit trail (logs)
- User awareness of security
- Physical controls
- Software licensing and intellectual property issues

Encapsulated Environments

- If you can't trust an executable, how can you run it?
- Put it in a box where it can't do much harm
- Today's systems offer only limited abilities to do that

Options for Encapsulation Today

- Create a new user ID for the application
  - Be real careful about the privileges given to that user
- Run it under the Java virtual machine
  - In the most restrictive mode

Improved Encapsulation Solutions

- Alter the OS
- Use existing OS mechanisms to build new protection domains
- Address space protection
- Language-based solutions

OS-Based Access Control Improvements

- Change the OS to add finer granularity access controls
- And/or more flexibility in setting up security domains
- Use the new OS tools to solve the problem
  - Begging the question of, how?

Pros and Cons of OS-Based Solutions

+ Potentially good performance
+ With good design, arbitrary flexibility
– You must alter the OS
– High security penalties if you blow it
– Only likely to be effective if lots of folks play the game

Example - DTE

- Use OS alteration to allow checking of separate access control database
- Each process' security permissions specified in database
- When process tries to do something, check database to see if it's permitted

Leveraging Existing Operating System Features

- Make clever use of existing OS features to improve access control
- Usually by trapping particular system calls in clever ways
- When trapped, apply access control to them in new ways

Pros and Cons of Leveraging OS Features

+ Often pretty cheap and easy to build
+ Can work at the user level
+ Can use existing, proven access control as a fallback
– Security retrofits have a dismal history
– May have performance problems
– May offer limited leverage

Example - Janus

- Designed to limit access for Web helper programs
- Uses the Unix `/proc` file system to trap system calls from these processes
- When trapped, check to see if they are allowable
- High overhead whenever you do this
  – So better not do it often

Address Space Protection

- The approaches already discussed have a fundamental limitation -
  – They only protect things outside the process' address space
- Most access control assumes a process should have unlimited access to its own address space

Intra-Address Space Protection

- Why shouldn't a process completely control its address space?
- Because of composable applications
- For performance reasons, different components may need to share an address space
- Yet they may have their own security requirements

Building Programs Out of Components

- Increasingly, programs are being built out of pre-written components
  – Due to COM, CORBA, etc.
- So to build a program, slap together half a dozen pre-existing pieces
  – And add a little of your own code
- But can you trust the pieces?

An Example

- You are building a large application
- Rather than develop your own btree package, you want to buy a commercial one
- It will be heavily used, so you want to link it into your process
- How can you be sure it won't misbehave?

Access Control Implications of Finer Granularity

- Within a single address space, we need multiple access control domains for file references, IPC, etc.
- **But we also need access control for memory references!**
- Can no longer rely on hardware virtual memory protection

Approaches to Protecting Memory

- Segment matching
- Address sandboxing

The Basic Problem To Be Solved

- Two mutually distrusting code segments share a single address space
- They export operations to each other
- How can we guarantee that they touch each other only through those interfaces?
- Given that they can issue each other's addresses

Other Constraints

- Must not be limited to a single language
  - Any executable must work
- Must be enforced at run time
- Must be relatively cheap
  - Or you might as well move the code to a different address space

Segment Matching

- Examine executable about to be loaded for "unsafe instructions"
- What is unsafe?
  - Any jump or store to address that can't be statically verified
  - E.g., jump through register, store through register

Handling Unsafe Instructions

- Define virtual memory segments that a piece of code can legitimately address
- For each unsafe instruction, insert new instructions in the executable to check it at run time
- Could be done at compile time or load time

Checking Unsafe Instructions

- Fundamentally, examine the non-static address the code proposes to use
- If it's within the code's boundaries, let it happen
- If not, prevent it
- And report the violation

Costs of Segment Matching

- Must reserve several registers for this purpose
  - Four, in Berkeley implementation
- Additional instructions performed
  - Four, in Berkeley implementation for a typical RISC processor

Address Sandboxing

- Reduces the cost of providing this level of safety
- But loses ability to pinpoint attempts to bypass the security
- Essentially, instead of checking, just apply a mask to unsafe addresses
  - Mask ensures that address is within permitted segments

When Is Software-Enforced Fault Isolation Valuable?

- It's expensive, because it adds instructions to code
  - Perhaps in common cases
- But not nearly as expensive as IPC
- So it wins if the code performs a lot of IPC
- Also requires fast RPC across protection domains

Virtual Machine and Language Approaches

- These approaches don't allow rapid downloading and execution of programs
  - Which is highly valuable
- What if you couldn't write a program that behaved badly?
- What if the machine enforced that?

The Virtual Machine Approach

- Define a virtual machine that does not allow "insecure" operations
- Write all untrusted programs in a language that works on that virtual machine
- Run imported programs through an interpreter for that language

How Do You Do This "Right"?

- Carefully design a virtual machine that cannot perform insecure operations
  - If properly implemented
- Require all imported programs to be written in its language
- Interpret those programs at run time
  - Or compile at download time

Why Isn't This Easy?

- How do you design a virtual machine that does useful things?
  - But nothing insecure
- How do you implement the virtual machine and compiler/interpreter?
- Can this perform well enough?

Example - Java

- Java tackles these problems
- The Java virtual machine is meant to provide a secure execution environment
  - Also portable
- The Java language ensures that all program operations are in the context of that VM

Language and Virtual Machine Definitions

- All security depends on the virtual machine not allowing insecure things
- And on the language only working on the real machine through the virtual machine
- So they must be carefully defined to not allow any insecure operations

Secure Implementation of the Virtual Machine

- Given that the definition of the virtual machine is secure, we must be sure that the implementation matches the definition
- Essentially, this is the same problem as verifying that an OS is secure
  - Perhaps on a smaller scale, though

Java Interpreters

- Only allow Java "source" code to be executed
  - "Source" code is actually Java bytecode
    - A portable "assembly language"
- Then run it through a trusted interpreter
  - Which verifies that only approved Java VM operations are invoked

Access Control and the Java Virtual Machine

- At best, this approach limits access to the Java virtual machine
- So you must define that VM so a Java program cannot do anything "bad"
- What is allowed is a key issue
  - All the security is based on the virtual machine operations being acceptable

Functionality vs. Security: the Java Version

- The same old issue arises
- More security or more functionality
- Java originally chose strong security
  - Modulo the usual bugs
- But people couldn't do what they needed to do
- So Java's security model was weakened
- And now security-conscious people turn off Java in their browsers