Data Security and Privacy

Topic 4: Analysis of DAC’s Weaknesses
Why Computers are Vulnerable?

• Programs are buggy

• Humans make mistakes

• Access control is not good enough
  – Discretionary Access Control (DAC) used in Unix and Windows assume that programs are not buggy
Access Control Check

• Given an access request, return an access control decision based on the policy
  – allow / deny

A Request → Access Control Check → Allow / Deny

The Policy
Discretionary Access Control

• No precise definition. Basically, DAC allows access rights to be propagated at subject’s discretion
  – often has the notion of owner of an object
  – used in UNIX, Windows, etc.

• According to TCSEC (Trusted Computer System Evaluation Criteria)
  – "A means of restricting access to objects based on the identity and need-to-know of users and/or groups to which they belong. Controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (directly or indirectly) to any other subject."

• Often compared to Mandatory Access Control
Analysis why DAC is not Good enough

- DAC causes the Confused Deputy problem
  - Solution: use capability-based systems

- DAC does not preserves confidentiality when facing Trojan horses
  - Solution: use Mandatory Access Control (BLP)

- DAC implementation fails to keep track of for which principals a subject (process) is acting on behalf of
  - Solution: fixing the DAC implementation to better keep track of principals
The Confused Deputy Problem

The Confused Deputy by Norm Hardy
Analysis of The Confused Deputy Problem

• The compiler runs with authority from two sources
  – the invoker (i.e., the programmer)
  – the system admin (who installed the compiler and controls billing and other info)
• It is the deputy of two masters
• There is no way to tell which master the deputy is serving when performing a write
• Solution: Use capability
ACCESS MATRIX MODEL

Objects (and Subjects)

U

V

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>r w own</td>
<td>r</td>
</tr>
<tr>
<td>V</td>
<td>r w own</td>
<td></td>
</tr>
</tbody>
</table>
IMPLEMENTATION OF AN ACCESS MATRIX

• Access Control Lists
  – Encode columns
• Capabilities
  – Encode rows
• Access control triples
  – Encode cells
ACCESS CONTROL LISTS (ACLs)

Each column of the access matrix is stored with the object corresponding to that column.

F

U:r
U:w
U:own

G

U:r
V:r
V:w
V:own
CAPABILITY LISTS

U  | F/r, F/w, F/own, G/r
V  | G/r, G/w, G/own

each row of the access matrix is stored with the subject corresponding to that row
ACCESS CONTROL TRIPLES

<table>
<thead>
<tr>
<th>Subject</th>
<th>Access</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>r</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>w</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>own</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>r</td>
<td>G</td>
</tr>
<tr>
<td>V</td>
<td>r</td>
<td>G</td>
</tr>
<tr>
<td>V</td>
<td>w</td>
<td>G</td>
</tr>
<tr>
<td>V</td>
<td>own</td>
<td>G</td>
</tr>
</tbody>
</table>

commonly used in relational DBMS
Different Notions of Capabilities

- Capabilities as a row representation of Access Matrices
- Capabilities used in POSIX/Linux as a way to divide the root power into multiple pieces that can be given out separately
- Capabilities as a way of implementing the whole access control systems
  - Subjects have capabilities, which can be passed around
  - When access resources, subjects select capabilities to access
    - An example is open file descriptors
  - We will examine this last notion in more depth
More on Capability Based Access Control

• Simulated by: a UNIX system where only owner of a file can open the file, and file sharing is done by passing opened file descriptors around

• Subjects have capabilities, which
  – Give them accesses to resources
    • E.g., like keys
    – Are transferable and unforgeable tokens of authority
      • Can be passed from one process to another
        – Similar to opened file descriptors

• Why capabilities may solve the confused deputy problems?
  – When access a resource, must select a capability, which also selects a master
How the Capability Approach Solves the Confused Deputy Problem

- Invoker must pass in a capability for $OUTPUT, which is stored in slot 3.
- Writing to output uses the capability in slot 3.
- Invoker cannot pass a capability it doesn’t have.
Capability vs. ACL

• Consider two security mechanisms for bank accounts.
• One is identity-based. Each account has multiple authorized owners. You go into the bank and show your ID, then you can access all accounts you are authorized.
  – Once you show ID, you can access all accounts.
  – You have to tell the bank which account to take money from.

• The other is token-based. When opening an account, you get a passport to that account and a PIN, whoever has the passport and the PIN can access
Capabilities vs. ACL: Ambient Authority

- Ambient authority means that a user’s authority is automatically exercised, without the need of being selected.
  - Causes the confused deputy problem
  - Violates the least privilege principle

- No Ambient Authority in capability systems
Capability vs. ACL: Naming

• ACL systems need a namespace for objects
• In capability systems, a capability can serve both to designate a resource and to provide authority.
• ACLs also need a namespace for subjects or principals
  – as they need to refer to subjects or principals
• Implications
  – the set of subjects cannot be too many or too dynamic
  – most ACL systems grant rights to user accounts principals, and do not support fine-grained subject rights management
Conjectures on Why Most Real-world OS Use ACL, rather than Capabilities

• Capability is more suitable for process level sharing, but not user-level sharing
  – user-level sharing is what is really needed

• Processes are more tightly coupled in capability-based systems because the need to pass capabilities around
  – programming may be more difficult
INHERENT WEAKNESS OF DAC

• Unrestricted DAC allows information flows from an object which can be read to any other object which can be written by a subject
  – Suppose A is allowed to read some information and B is not, A can reads and tells B

• Suppose our users are trusted not to do this deliberately. It is still possible for Trojan Horses to copy information from one object to another.
TROJAN HORSE EXAMPLE

Principal B cannot read file F

File F
ACL
A:r
A:w

File G
B:r
A:w
Principal B can read contents of file F copied to file G
Buggy Software Can Become Trojan Horse

- When a buggy software is exploited, it executes the code/intention of the attacker, while using the privileges of the user who started it.

- This means that computers with only DAC cannot be trusted to process information classified at different levels
  - Mandatory Access Control is developed to address this problem
  - We will cover this in the next topic
DAC’s Weaknesses Caused by The Gap

- A request: a subject wants to perform an action
  - E.g., processes in OS
- The policy: each principal has a set of privileges
  - E.g., user accounts in OS

- Challenging to fill the gap between the subjects and the principals
  - relate the subject to the principals
<table>
<thead>
<tr>
<th>Action</th>
<th>Process</th>
<th>Effective UID</th>
<th>Real Principals</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A Logs In</td>
<td>shell</td>
<td>User A</td>
<td>User A</td>
</tr>
<tr>
<td>Load Binary “Goodie” Controlled by user B</td>
<td>Goodie</td>
<td>User A</td>
<td>? ?</td>
</tr>
</tbody>
</table>

• When the Goodie process issues a request, what principal(s) is/are responsible for the request?
• Under what assumption, it is correct to say that User A is responsible for the request?

Assumption: Programs are benign, i.e., they only do what they are told to do.
UNIX DAC Revisited (2)

<table>
<thead>
<tr>
<th>Action</th>
<th>Process</th>
<th>Effective UID</th>
<th>Real Principals</th>
</tr>
</thead>
<tbody>
<tr>
<td>shell</td>
<td>AcroBat</td>
<td>User A</td>
<td>User A</td>
</tr>
<tr>
<td>Load AcroBat Reader Binary</td>
<td>AcroBat</td>
<td>User A</td>
<td>User A</td>
</tr>
<tr>
<td>Read File Downloaded from Network</td>
<td>AcroBat</td>
<td>User A</td>
<td>? ?</td>
</tr>
</tbody>
</table>

• When the AcroBat process (after reading the file) issues a request, which principal(s) is/are responsible for the request?
• Under what assumption, it is correct to say that User A is responsible for the request?

Assumption: Programs are correct, i.e., they handle inputs correctly.
Why DAC is vulnerable?

• Implicit assumptions
  – Software are benign, i.e., behave as intended
  – Software are correct, i.e., bug-free

• The reality
  – Malware are popular
  – Software are vulnerable

• The problem is not caused by the discretionary nature of policy specification!
  – i.e., owners can set policies for files
Why DAC is Vulnerable? (cont’)

• A deeper reason in the enforcement mechanism
  – A single invoker is not enough to capture the origins of a process

• When the program is a Trojan
  – The program-provider should be responsible for the requests

• When the program is vulnerable
  – It may be exploited by input-providers
  – The requests may be issued by injected code from input-providers

• Solution: include input-providers as the principals
Coming Attractions …

- The Bell LaPadula Model