Access Control and Protection
Overview

- Access control: What and Why
- Abstract Models of Access Control
  - Discretionary access control
  - Mandatory access control
- Real systems: Unix Access Control Model
Access control

Access Control (AC) must decide which principals (people, processes, machines,..) have access to which resources in the system (files they can read/write, program they can execute, how to share data with others,..).

Recall: principals are humans, programs
Access control

Several levels

• Application level: AC limits users accesses
• Middleware: applications are written on top of a middleware (eg Database); AC controls applications verify rules (eg a Database that credits one user must debit another user)
• Operating system: middleware uses facilities of OS
• Hardware: the operating system
Access control

As we work up from hardware to applications things are more complicated and less reliable

Most frauds involve staff accidentally discovering features of the application code that they can exploit in an opportunistic way

Questions:

• Can we find logical models to study these systems?

• We can use these models to design new safe systems?
The Reference Monitor

- An abstract model of protection
  - Sometimes quite close to implementation
  - e.g. SELinux (Flask Architecture)
- In practice should offer:
  - Complete mediation
  - Be Tamper proof
  - Be small enough to understand (verify)
- Important Idea:
  - Computer systems are BIG and Complex, Security relevant part often SMALL, extract that part out that deals with security so that we can understand/verify it
Challenge of Controlled Sharing

Two conflicting principles

1. Principle of Least privilege
   • Make controls granular enough and apply them to enforce them to minimize harm.

2. Psychological acceptability
   • Get as close to the users mental model (be it the programmer, user or admin) as possible.

Reconciling these two is a fundamental challenge.
Operating Systems

• Most heavily studied area for access control
• First computers where single user/no network
• Timesharing systems introduced need for access control
  - studied heavily in the 70’s and 80’s.
• Still an open research area, why?
  - First 20 years: restrict what a user can do with data, focus on military problems, thought problem was malicious users
  - Last 10: Malicious applications the primary problem; complex software we use today: we still have lots to learn about how programs are built, how people use them
Subjects and Objects

- **Subjects**
  - can be processes, modules, roles

- **Objects**
  - can be files, processes, etc.

- Authentication often used to bootstrap subjects, but not necessary.
  - e.g. process assumes identity of one subject, then another

- Authenticated users have different access privileges
**Access Control Matrix**

- 2-dim matrix describes the state of the system
  - Also 3-dim matrix: users, data, programs
- **Access Rights**: Read, Write, eXecute, no, (r/w/x/-)
  - There are other possibilities: delegation, modify, ...
  - Also 3-dim matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>r</td>
<td>w</td>
<td>rw</td>
<td>x</td>
</tr>
<tr>
<td>bob</td>
<td>-</td>
<td>rx</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>charlie</td>
<td>-</td>
<td>rx</td>
<td>rx</td>
<td>r</td>
</tr>
<tr>
<td>dave</td>
<td>rwx</td>
<td>rwx</td>
<td>rwx</td>
<td>r</td>
</tr>
</tbody>
</table>
**Access Control Matrix**

- **Example**
  - Alice, system administr.: universal access but not on audit data
  - Bob: manager can execute program and oper.
  - System
  - Accounts program: writes data
  - Dave, auditor: controls

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Operat. system</th>
<th>Accounts program</th>
<th>Account. data</th>
<th>Audit data</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>rwx</td>
<td>rwx</td>
<td>rw</td>
<td>r</td>
</tr>
<tr>
<td>bob</td>
<td>rx</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Accounts prog.</td>
<td>rx</td>
<td>r</td>
<td>rw</td>
<td>w</td>
</tr>
<tr>
<td>dave</td>
<td>rx</td>
<td>rw</td>
<td>r</td>
<td>r</td>
</tr>
</tbody>
</table>
Grouping

In real systems there are many subjects and objects

- **Grouping Subjects**
  - Groups e.g. staff = {alice, dave}, students = {bob, charlie}

- **Grouping Objects**
  - Types e.g. system_file = {A, B}, user_file = {C, D}

- **Roles**: similar to groups but
  - A fixed set of access permissions one may assume for some time (example: officer of the watch in a ship: is a person that is in the group of officer and is responsible for watch: on each ship there is exactly one officer at any moment and the role changes over the day)
Access control: MAC and DAC

Two main approaches:

• **Mandatory Access Control (MAC):**
  - the operating system constrains the ability of a subject to access or perform some sort of operation on an object. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system kernel examines these security attributes and decides whether the access can take place.

• **Discretionary access control (DAC):**
  - allows users the ability to make policy decisions (to allow or deny access and/or assign security attributes)
Access control

- Many real systems combine in different ways MAC and DAC
- There are also other approaches
  - The Chinese Wall Model - it combines elements of DAC and MAC
  - RBAC Model - it is a DAC model; however, it is sometimes considered a policy-neutral model
  - The Biba Model - relevant for integrity
  - The Information-Flow model - generalizes the ideas underlying MAC
DAC

• DAC (Discretionary Access Control) policies control the access of subjects to objects on the basis of subjects' identity, objects' identity and permissions

• When an access request is submitted to the system, the access control mechanism verifies whether there is a permission authorizing the access

• Such mechanisms are discretionary in that they allow subjects to grant other subjects authorization to access their objects at their discretion
The HRU Model

The Harrison-Ruzzo-Ullman (HRU) model. We are given

- $S$ be a set of subjects (users)
- $O$ be a set of objects
- $R$ be a set of access rights ($r$- read, $w$- write, $o$- owner,...)
- an access matrix $M = (M_{so})_{s \in S, o \in O}$
- the entry $M_{so}$ is the specifies the rights subject $s$ has on object $o$

The model includes six primitive operations for manipulating the set of subjects, the set of objects, and the access matrix:

- enter $r$ into $M_{so}$ / delete $r$ from $M_{so}$
- create subject $s$ / delete subject $s$
- create object $o$ / delete object $o$
The HRU Model - Commands

Commands in the HRU model have the format

\[
\text{command } c(x_1,\ldots,x_k)
\]
\[
\text{if } r_1 \text{ in } M_{s_1,o_1} \text{ and }
\]
\[
\text{if } r_2 \text{ in } M_{s_2,o_2} \text{ and }
\]
\[
\vdots
\]
\[
\text{if } r_m \text{ in } M_{s_m,o_m}
\]
\[
\text{then } op_1,\ldots,op_n
\]
\[
\text{end}
\]

- The indices \( s_1,\ldots,s_m \) and \( o_1,\ldots,o_m \) are subjects and objects that appear in the parameter list \( c(x_1,\ldots,x_k) \)
- The condition part of the command checks whether particular access rights are present; the list of conditions can be empty
- If all conditions hold, then the sequence of basic operations is executed
The HRU Model: examples

command create_file (s,f)
    create f /* create file f
    enter o into Ms,f /* s is the owner of f
    enter r into Ms,f /* s can read f
    enter w into Ms,f /* s can write f
end

command grant_read (s,p,f)
    if o in Ms,f /* if s is the owner of file f
        then enter r into Mp,f /* allow p to read f
end
The HRU Model - Protection Systems

A protection system is defined as

- A finite set of rights
- A finite set of commands

A protection system is a state-transition system

- The effects of a command are recorded as a change to the access matrix (usually the modified access control matrix is denoted by $M'$)
- Hence the access matrix is changing; as in a finite state machine the state describes the current situation of the protection system
The HRU Model: Safety

- A state, i.e. an access matrix $M$, is said to leak the right $r$ if there exists a command $c$ that adds the right $r$ into an entry in the access matrix that previously did not contain $r$.
- Delegation occurs when a right is leaked; this is not necessarily bad: many systems allow subjects to give other subjects access rights.

What do we mean by saying that a state is "safe"?

**Definition**: “the user should be able to tell whether what he is about to do (give away a right, presumably) can lead to the further leakage of that right to truly unauthorized subjects” [HRU76]
HRU Model: safety

An example of “unsafe” protection system
Consider the following two commands:

```plaintext
command grant_execute (s,p,f)
    if o in Ms,f
    then enter x into Mp,f
end

command modify_own_right (s,f)
    if x in Ms,f
    then enter w into Ms,f
end
```
HRU Model: safety

An example of “unsafe” protection system

• Suppose Bob writes an application program; he wants this program to be run by other users but not modified by them

• The previous protection system is not safe with respect to this policy; consider the following sequence of commands:
  - Bob: grant_execute (Bob, Alice, P1)
  - Alice: modify_own_right (Alice, P1)

then entry $M_{alice,P1}$ of access matrix contains the w access right
The HRU Model - Safety

**Definition.** Given a protection system and a right $r$, the initial configuration $Q_0$ is *unsafe* for $r$ (or leaks $r$) if there is a configuration $Q$ and a command $c$ such that
- $Q$ is reachable from $Q_0$
- $c$ leaks $r$ from $Q$

$Q_0$ is *safe* for $r$ if $Q_0$ is not unsafe for $r$

**Equivalently** A matrix $M$ that is the state of a protection system, is *safe* with respect to the right $r$ if no sequence of commands can transform $M$ into a state that leaks $r$

**Theorem.** Given an access matrix $M$ and a right $r$, verifying the safety of $M$ with respect to $r$ is an undecidable problem

**NOTE:** This implies that we can analyse with simple protection systems; complex cases cannot be decided (equivalently DAC models in complex cases might be unsafe)
The Trojan Horse

DAC models are unable to protect against Trojan Horses embedded in application programs

Alice can read and write O1 (secret file); nobody else should read O1

At the end Bob can read O2 (and therefore O1)
DAC: conclusions

- **Advantages:**
  - Flexibility in terms of policy specification
  - Supported by all OS and DBMS
- **Drawbacks:**
  - Complex policies are difficult to analyse
  - No information flow control (Trojan Horses attacks)
MAC

• MAC specifies the access that subjects have to objects based on subjects and objects classification

• This type of security has also been referred to as multilevel security

• Many of the MLS/DBMSs have been designed based on the Bell and LaPadula (BLP) model
MAC model: Multilevel Security

Classical military model

• Vertical classification levels
  - Unclassified < Confidential < Secret < Top Secret
• Horizontal compartments
  - Nuclear, Terrorism, Biowar, etc.

Elements of the model:

• objects - passive entities containing information to be protected
• subjects: active entities requiring accesses to objects (users, processes)
• access modes: types of operations performed by subjects on objects
BLP Model - access classes

• An access class consists of two components
  a security level a category set

• The security level is an element from a totally ordered set - example
  {Top Secret (TS), Secret (S), Confidential (C), Unclassified (U)} where
  TS > S > C > U

• The category set is a set of elements, dependent from the application area in which data are to be used - example
  {Army, Navy, Air Force, Nuclear}
BLP Model - Access classes

Access class \( c_i = (L_i, SC_i) \) dominates access class \( c_k = (L_k, SC_k) \), denoted as \( c_i \geq c_k \), if:

- \( L_i \geq L_k \), the security level of \( c_i \) is greater or equal to the security level of \( c_k \)
- the category set of \( c_i \) includes the category set of \( c_k \)

Example

\[ c_1 = (TS, \{\text{Nuclear, Army}\}) \]
\[ c_2 = (TS, \{\text{Nuclear}\}) \]
\[ c_3 = (C, \{\text{Army}\}) \]

- \( c_1 \geq c_2 \)
- \( c_1 > c_3 \) (TS > C and \( \{\text{Army}\} \) is included \( \{\text{Nuclear, Army}\} \))
- \( c_2 < > c_3 \)
Bell-Lapadula

- Access classes

```
<confidential>
<Secret>
<Secret, army>
<Secret, Nuclear>
<Top Secret>
<Top Secret, army>
<Top Secret, Nuclear>
<Top Secret, {Nuclear, army}>
```
BLP Model - Axioms

- The state of the system is described by the pair \((A, L)\), where:
  - \(A\) is the *set of current accesses*: triples of the form \((s,o,m)\) denoting that subject \(s\) is exercising access \(m\) on object \(o\) - example \((Bob, o_1, \text{read})\)
  - \(L\) is the *level function*: it associates with each element in the system its access class
BLP Model - Axioms

Axioms for security (mandatory)

• No read up (user cannot read something has higher security level)

• No write down (user with higher security level cannot write something with lower security level)

Example: a subject with access class \((C, \{\text{Army}\})\) is not allowed to read objects with access classes \((C, \{\text{Navy, Air Force}\})\) or \((U, \{\text{Air Force}\})\)

Note: Axioms ensure that users have access only to information for which they have the necessary access class and that no information is leaked
MLS in the world

- Classification level added by tagging files
  - Just like DTE
- Everything tends to flow up
  - Ends up being very cumbersome in practice
- Declassification
  - Don’t know how to automate this, human in the loop
  - Becomes a bottle neck in critical situations
- Almost no one uses this in practice
- Recall: most real systems combine ideas of DAC and MAC
Unix access control

- Based on DAC
- Access control list
Access Control Lists: ACL

- Store the matrix one column at a time
  - Each object has a set of \(<user, right>\) tuples
  - Accounting data \{\<Alice, rw\>, \<Accounts program, rw\>, \<Dave, r\>\}

- Properties
  - Simple: good for few users and for many applications (file systems) where the policy is decided centrally
  - Can grow quite large and long in dynamic environments users enter and exit frequently
Capabilities

Store the matrix one row at a time
  - Alice \{<A,r/w>, <B,w>, <C,r>\}

  • Natural model for delegation (rights coupled to object): Bob can create a certificate saying “here is my capability and I delegate David to read file X, signed Bob”
  • Each tuple can be viewed as a handle to an object
  • Similarity with Certification authorities
Unix Resource

• Files
  • Includes devices, IPC (domain sockets, named pipes, mmap shared memory)

• Network
  • TCP/UDP port address space

• Other
  - Sys V IPC has its own name space
  - ability to load kernel modules
Unix: File system access control

10 characters: first directory, rest info for owner/group/ other

- owner group other
  - `-rw-` `rwx` `rwx` general form for a file
  - "`rwx-r-----` Alice Accounts" (Alice is file owner, she can read execute and write, people in the group Accounts can read, others cannot read or write)
  - root can do everything
  - root access is given to system administrator; therefore system administrator can write every file (security hole)
  - List refer to users not programs: problem
  - New files created according to "umask"
  - setuid bit for executables - changes uid to uid of file owner on exec() of file.
  - Execute on directory implies access to directory.
Unix Identities

- Each process has set of identities used to make access control decisions.
  - uid (user identity), gid (group identity) most important
  - euid used for access control decisions
  - Allows process to drop and resume privilege temporarily
- Lots of hard coded rules
- Highest privilege uid is root (uid = 0), has most privilege,
  - Access privileged ports, override file permissions, load kernel modules, etc.
  - Once you have root, you own the system
  - this is a source of many problems.
Unix: delegation

- ACL make it difficult to realize delegation
- File descriptors are capabilities
- Ability to delegate files by passing descriptors over a unix domain socket (`sendmsg()`, `recvmsg()`)
- Supports privilege separation
  - One process runs as root, opens a unix domain socketpair.
  - `Forks()` another process, drops privilege
  - More privilege process can pass descriptors to less privilege process, also do other stuff e.g. manage crypto keys.
  - Can apply this paradigm ad nauseam i.e. multiple unprivileged processes.
Dangers of bad access control model

- **Too much privilege!**
  - Original Unix architecture suffers severely
  - Windows world even worse
  - Once ISV’s (developers) expect this, very hard to go back!

- **Wrong granularity**
  - Too coarse grain => too much privilege
  - Too fine grain => too many knobs, no one knows how to use it
    - (SELinux struggles with this, its gotten better, but still not for the faint of heart)
  - Compromise
    - Protection Profiles, Roles, etc. - use fine grain permissions
File system ACL’s

• Basic Problem
  - I want to do a project with a new group, how do I share files with only them without involving my sysadmin?

• Ghetto style
  - Create a directory accessible but not readable by everyone
  - Make directory (or files) in that directory with secret name
  - Hand out name as a capability
  - Gross and many limitations...

• ACL’s a better solution
  - User created groups, user settable list of groups/users on files/directories

• Almost everywhere now
  - AFS, NTFS, NFSv4, FreeBSD, Solaris, ZFS, etc.
Unix: conclusions

• Good
  - Works decently well on multi-user system where programs are not malicious -- shortcomings can be fixed with file system ACL's

• Bad
  - Lot of rules; in large systems errors are common
  - Very difficult/sometimes impossible to contain damage of malicious root process
  - Lots of stuff needs root
  - Users can't protect themselves from bad apps, i.e. apps totally trusted!!

• SELinux: add Mac to Unix
Windows: NT

• Similar to Unix
• More sophisticated rules for delegation:
  - More attributes: besides r/w/x there are “takeownership”, “change permission”, “delete”
  - Attributes are not on/off but there are more complex values: access to data is given according to the
• Partitioning into domains with distinct administrators
Biba-Integrity Model

• Cool Observation: Integrity the opposite of secrecy -- low integrity data flowing upwards leads to system compromise.

• Biba Rules
  - No read down
  - No write up

• Again, tends to be kind of unwieldly, can make sense for some systems.
Conclusions

Most of you will go off to companies to write code

• Many attacks seen in the wild could have been rendered moot if software used access controls properly.
• Learn the access control system of your language runtime and OS
  – Learn their strengths and pitfalls.
• Build systems according to saltzer and schroders principles.
  – Be conservative in your design, assume things will fail, attempt to minimize harm
• Building things right requires a bit more effort
• Fixing things later requires far more
  – sometimes impossible or impractical.
Conclusions/2

- Express information access policies.
  - Who can access my files, who can't, how.

- Sandboxing: Minimize damage inflicted by malicious program
  - Java virtual machine

- Privilege Separation: Allow programs to be written to minimize damage in case of failure.
  - Think about compartments in a ship

- Defense-in-Depth
  - Programmers make mistakes, protection systems fail, redundancy always a good thing!!