CS 1699
Privacy in the Electronic Society

William Garrison
bill@cs.pitt.edu
6311 Sennott Square
http://cs.pitt.edu/~bill/1699

14: Access control
How do access control systems determine which users can access which resources?

**Recall:** Authentication binds identity to session
- Assume we have already authenticated users
- Once we know who they are, what can they access?

Today, we’ll talk about ways to store and enforce an access control protection state
- Basics: Access control matrix
- Splitting the matrix: ACLs and capabilities
- Indirection: RBAC
- For more flexibility: Logic-based policies
- Looking ahead: Special cases for social networks, health, etc.
What data is stored to enforce access control?

Protection state: Current snapshot of the access control data structures

The most basic form of protection state is the access control matrix

- Set of subjects, $S = \{s_1, s_2, \ldots, s_n\}$
- Set of objects, $O = \{o_1, o_2, \ldots, o_m\}$
- Set of rights, $R = \{r_1, r_2, \ldots, r_k\}$ (say, $\{r, w, a, x\}$)
- Each entry in $A$ is a subset of $R$, where $A[s_i, o_j] = \{r_x, \ldots, r_y\}$ means that subject $s_i$ has rights $r_x, \ldots, r_y$ over object $o_j$
# A Sample Access Control Matrix

<table>
<thead>
<tr>
<th></th>
<th>/etc/passwd</th>
<th>Alice_priv.txt</th>
<th>recipes.html</th>
<th>/etc/shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>read</td>
<td>read, write, own</td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>read</td>
<td></td>
<td>read, write, own</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>read</td>
<td></td>
<td>read</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Specific rights in the matrix are system dependent!

- In this case {read, write, execute, own} for files

**What about a matrix describing a (simple) firewall?**

- Subjects: (IP, port) tuples “outside” the firewall
- Objects: (IP, port) tuples “inside” the firewall
- Rights: {pass, drop}

**Problems?**
Idea: Break the matrix up into columns

Encode each resource’s access control list (ACL) in its metadata

- Each object is associated with a list of \((\text{subject}, \text{rights})\) pairs
- At access time, check if subject is in ACL for the object
- What real-life scenarios look like this?

More formally, we can view ACL as a function from object to privileges

- \(acl: O \rightarrow \mathcal{P}(S \times \mathcal{P}(R))\)
- Given an object \(o\), \(acl(o)\) returns a set of \((s_i, R_i)\) pairs
  - Each one says that subject \(s_i\) can access \(o\) using any right in \(R_i\)
An ACL example

<table>
<thead>
<tr>
<th></th>
<th>/etc/passwd</th>
<th>Alice_priv.txt</th>
<th>recipes.html</th>
<th>/etc/shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>read</td>
<td>read, write, own</td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>read</td>
<td>read, write, own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>read</td>
<td></td>
<td>read</td>
<td></td>
</tr>
</tbody>
</table>

From this access matrix, we have:

- `acl(/etc/passwd) = {(Alice, {r}), (Bob, {r}), (Charlie, {r})}`
- `acl(Alice_priv.txt) = {(Alice, {r,w,o}), (Bob, ∅), (Charlie, ∅)}`
- `acl(recipes.html) = {(Alice, {r}), (Bob, {r, w, o}), (Charlie, {r})}`
- `acl(/etc/shadow) = {(Alice, ∅), (Bob, ∅), (Charlie, ∅)}`
Capability lists can be thought of as the dual of ACLs

Informally, a row in the access matrix
  • Permissions managed per user, not per file

Formally, a capability list is a set of pairs \( c = \{(o, r) \mid o \in O, r \subseteq R\} \)
  • \( \text{cap}: S \rightarrow \wp(O \times \wp(R)) \)
  • Given a subject \( s \), \( \text{cap}(s) \) returns a set of \((o_i, R_j)\) pairs
    • Each one says that subject \( s \) can access object \( o_i \) using any right in the set \( R_j \)
    • In practice, each capability \( c = (o, r) \) is managed as a discrete object
      • Given to user on grant
      • Used to prove access when needed

Concerns?
  • Delegation, capability modification, revocation...
Amoeba OS uses capabilities in its object model

To access a resource, a capability is needed:

- **Server port** (48 bits): Say, directory server (handles calls to filesystem)
- **Object number** (24 bits): A unique ID for the object
- **Rights** (8 bits): What accesses are allowed?
- **Check field** (48 bits): $H(port || object\ number || rights \oplus key)$
  - Key created per file, stored in directory server

Checking a capability:

- Is the server port my own?
- Determine key for this object number  
  - **Delegation**?
- Verify check field
- Grant requested privileges
In some scenarios, access controls are defined by system administrator rather than owner

Up to now, we’ve largely been assuming discretionary access control (DAC)

- Access controls for a file are set by the owner
- Consider a military scenario; different requirements!

Bell–LaPadula (BLP), a mandatory access control (MAC) system

- Label each file with a classification level
  - e.g., “public”, “classified”, “secret”, “top secret”
- Label each session with a clearance level (same set)
- Only allow read if session dominates file
- Only allow write if file dominates session
  - Why?
- In practice: Includes categories (need-to-know) and DAC component
Levels of indirection are useful when groups of permissions are often assigned together

Example: In Unix, /etc/passwd contains entries describing each user

- bill : x : 87 : 7 : William Garrison : /afs/cs.pitt.edu/usr0/bill : /bin/dash

The file /etc/group contains entries describing each group in the system

- lec : x : 7 : bill, phil, will, jill, drphil, mcgill, shill, dill

All files are assigned an owner (user) and group

- The chown command allows the owner of a file to transfer ownership
- The chgrp command allows the file owner to change the file’s group
- Request is considered first match in order: user, group, other
Role-based access control provides a more flexible approach to indirection

Intuition: Assign users to roles, assign permissions to roles
• In what ways might this be an improvement?

Formally:
• Set of subjects, $S = \{s_1, s_2, \ldots, s_n\}$
• Set of permissions, $P = \{p_1, p_2, \ldots, p_m\}$
  • Maybe of the form (object, right)
• Set of roles, $R = \{r_1, r_2, \ldots, r_l\}$
  • Typically describe job functions
• Subject assignment, $SA \subseteq S \times R$
• Permission assignment, $PA \subseteq R \times P$

Subject $s$ can be granted access to permission $p$ if there exists a role $r$ such that $(s, r) \in SA$ and $(r, p) \in PA$

Optionally, role hierarchy to allow inheriting permissions
For greater flexibility, label entities with attributes and construct expressive policies!

Attribute-based access control (ABAC): Encode features of users, files, rights, etc. as attributes
- Free text fields that generalize roles, etc.

Types of attributes
- **Subject attributes** describe users
  - age, clearance, role, organization
- **Action attributes** describe rights
  - read, delete, approve
- **Resource attributes** describe objects
  - classification, topic, department
- **Environment attributes** describe dynamic aspects of the scenario
  - time, location
Once attributes are assigned, construct policies over attributes

ABAC is sometimes called **policy-based** access control

- Construct protection state using expressive logical language
- Language supports features such as IF...THEN

Examples for our department

- If user is a CS student and time is M–F 7:00–22:00, access to enter Sennott Square is granted
- If user is CS faculty or staff, access to enter Sennott Square is granted
- If user is CS faculty teaching $c$, and current time is within 15 minutes of $c$’s scheduled time, access to enter $c$’s reserved classroom is granted

Different approaches encode policy differently

- **Expressiveness** describes the range of policies that can be represented
Conclusions

OS generally enforces access controls

How does the OS store protection state and enforce policy?

- Matrix vs. ACL vs. caps
- DAC vs. MAC
- RBAC and levels of indirection
- ABAC and other logic-based policies

Next: Access control in unique scenarios