Applied Cryptography and Network Security

Adam J. Lee

adamlee@cs.pitt.edu

6111 Sennott Square

Lecture #22: Access Controls

March 25, 2014

University of Pittsburgh

Announcements

Exam review: Thursday

Project 4 due on Thursday 4/3

- Some of you have talked to me
- None of you have finished your project writeup

There will be a short quiz in class next Thursday 4/3

Authentication is only the beginning…

During earlier lectures, we've looked at several ways in which we can authenticate users

Typical policy models typically manage permissions on a per-user basis

- This is clearly the case in DAC models
- In MAC models, we need to user ID to determine clearance level
- RBAC wouldn't work without User-Role bindings

Today, we'll look at how operating systems enforce access control policies at runtime

Overview

Quick overview of the access control matrix model

Access control lists

- Unix
- AIX
- Windows 2000/NT

Capabilities

- Unix file system
- Case study: Amoeba
- Revocation of capabilities

Differences between capabilities and ACLs

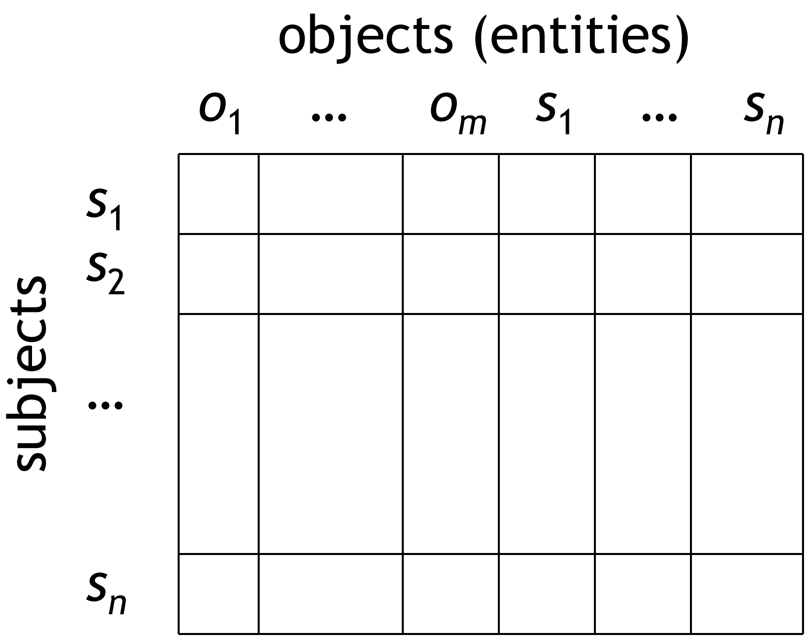
The subset of a system's state describing the protection of resources within the system is called its protection state

The Access Control Matrix (ACM) model is the simplest way to describe a system's current protection state

Formally:

- Subjects $S = \{ s_1,\ldots,s_n \}$
- Objects $O = \{ o_1,\ldots,o_m \}$
- Rights $R = \{ r_1,\ldots,r_k \}$
- Entries $A[s_i, o_j] \subseteq R$
- $A[s_i, o_j] = \{ r_x, \ldots, r_y \}$ means that subject s_i has rights $r_x, \ldots, r_y$ over object o_j

Let's look at an example…



A Sample Access Control Matrix

	/etc/passwd	Alice_priv.txt	recipes.html	/etc/shadow
Alice	read	read, write, own	read	
Bob	read		read, write, own	
Charlie	read		read	

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}

An access control matrix is the simplest way to describe a system's protection state

Formally:

- Subjects $S = \{ s_1,\ldots,s_n \}$
- Objects $O = \{ o_1,\ldots,o_m \}$
- Rights $R = \{ r_1,\ldots,r_k \}$
- Entries $A[s_i, o_j] \subseteq R$
- $A[s_i, o_j] = \{ r_x, \ldots, r_y \}$ means that subject s_i has rights $r_x, \ldots, r_y$ over object o_j

objects (entities)

subjects

$$\begin{array}{c|c|c|c|c|c|c|}
 & o_1 & \ldots & o_m & s_1 & \ldots & s_n \\
\hline
s_1 & & & & & & \\
\hline
s_2 & & & & & & \\
\hline
\ldots & & & & & & \\
\hline
s_n & & & & & & \\
\hline
\end{array}$$

Note: Specific rights in the matrix are system dependent!

- E.g., {read, write, execute, own} for files, {pass, drop} for firewalls

Although simple, this model has some problems

- Storing the entire matrix can be expensive (typically sparse)
- Need to be careful about how the matrix is updated

Access control lists make the access control process more decentralized

Informally, an access control list (ACL) is a column of the ACM

- Each object is associated with a list of (subject, rights) tuples
- At access time, the subject's request is checked against the ACL

Essentially, bouncers at expensive clubs follow the ACL model

More formally, an access control list can be represented as a function $acl : O \rightarrow P(S \times P(R))$. Given an object o, $acl(o)$ returns a set of (s_i, r_i) pairs such that subject s_i can access object o using any right in the set r_i.

An ACL example…

	/etc/passwd	Alice_priv.txt	recipes.html	/etc/shadow
Alice	read	read, write, own	read	
Bob	read		read, write, own	
Charlie	read		read	

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, ∅)

Question: Can you think of any problems with ACLs?

- Common permissions shared by groups of users
- Default permissions

To prevent ACL bloat, many systems use abbreviated versions of the ACL model

Example: In Unix, users are classified into groups

The file `/etc/passwd` contains entries describing each user

- adamlee : x : 87 : 7 : Adam J. Lee : /afs/cs.pitt.edu/usr0/adamlee : /bin/tcsh

Username uid default gid

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

- srg : x : 7 : adamlee, janedoe, johnsmith, tomthomas

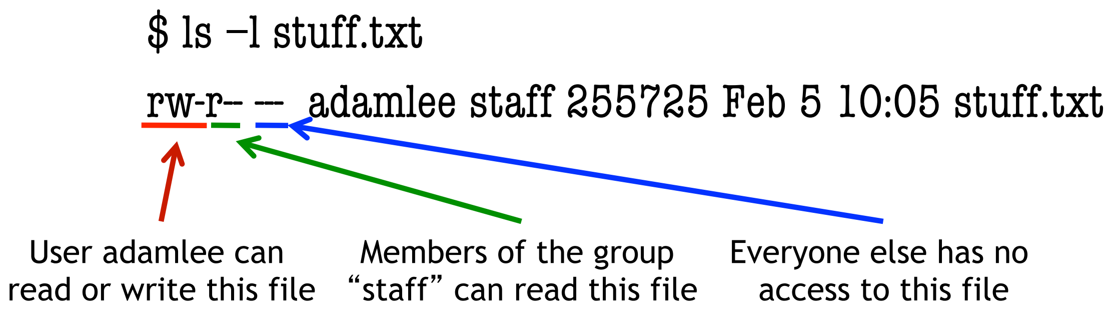
Group name gid List of users belonging to this group

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

Unix provides three levels of access control: User, Group, and Other

This can all be done using a mere 9 “mode” bits stored in the file’s inode!



```
$ ls –l stuff.txt
rw-r-- --  adamlee staff 255725 Feb 5 10:05 stuff.txt
```

User adamlee can read or write this file
Members of the group “staff” can read this file
Everyone else has no access to this file

This approach has several strengths

- Much simpler than full ACL management
- Multiple levels of control
- Minimal storage in kernel data structures

However, a weakness of this approach is that it is not terribly expressive

- Suppose Alice has a file that she wants {r,w,x} access to
- She also wants Bob to have {r, w} access, Charlie to have {r, x} access, and everyone else to be denied access
- There is no way to do this!

Windows NT/2000 also allows a richer access control model than Unix

In the NTFS file system, ACLs can be attached to any file or directory

Each ACL entry refers to a single user or a group

- Users can be members of multiple groups
- Groups can be defined locally or elsewhere in a workgroup

Each ACL entry permits or denies access to the six generic rights: read, write, execute, delete, change permissions, take ownership

How are a user's permissions decided?

- Not in the ACL? Denied.
- Denied by any rule (or rules) in the ACL? Denied.
- Otherwise, you get the set of rights specified by all applicable rules.

This model is richer than that of Unix, but not fundamentally different.

When dealing with ACLs, several important questions deserve thoughtful consideration

1. Which subjects can modify an ACL?
   - *Unix*: The file owner
   - *Windows*: Any user with the "change permission" privilege

2. Is there a privileged user? If so, do the ACLs apply to that user?
   - *Unix and Windows:* ACLs do not apply to root/Administrator

3. Does the ACL support groups, wildcards, and/or pattern matching?
   - *Unix and Windows:* Groups, yes. Wildcards, no.
   - The UNICOS operating system supports wildcards

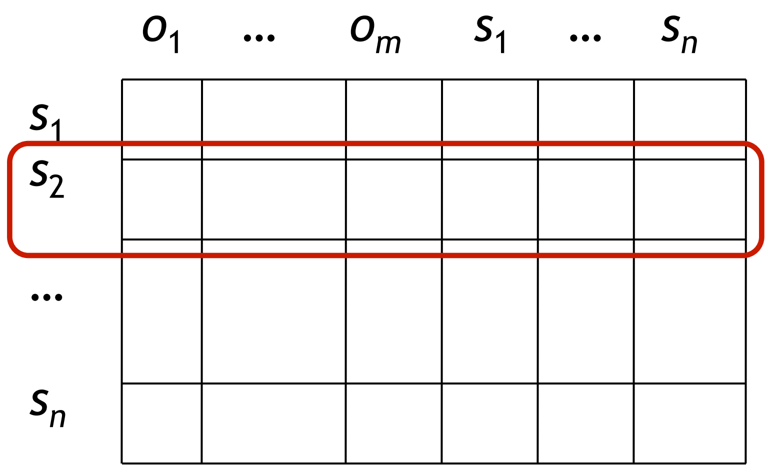
4. How are contradictory access control permissions handled?
   - *Unix and Windows:* Deny overrides

5. If a default setting is allowed, do ACL entries modify or override this default?

Capability lists are effectively the dual of access control lists

Informally, a capability list is a row in an access control matrix

As a result, users (not objects) manage their own set of permissions



More formally, a capability list is a set of pairs $c = \{(o, r) \mid o \in O, r \subseteq R\}$. Let $cap : S \rightarrow P(O \times P(R))$. Given an subject s, $cap(s)$ returns a set of (o_i, r_i) pairs such that subject s can access object o_i using any right in the set r_i.

In most applications, each capability $c = (o, r)$ is managed as a discrete object. The capability list is mainly a conceptual construct.

A capability example...

	/etc/passwd	Alice_priv.txt	recipes.html	/etc/shadow
Alice	read	read, write, own	read	
Bob	read		read, write, own	
Charlie	read		read	

From this access matrix, we have:

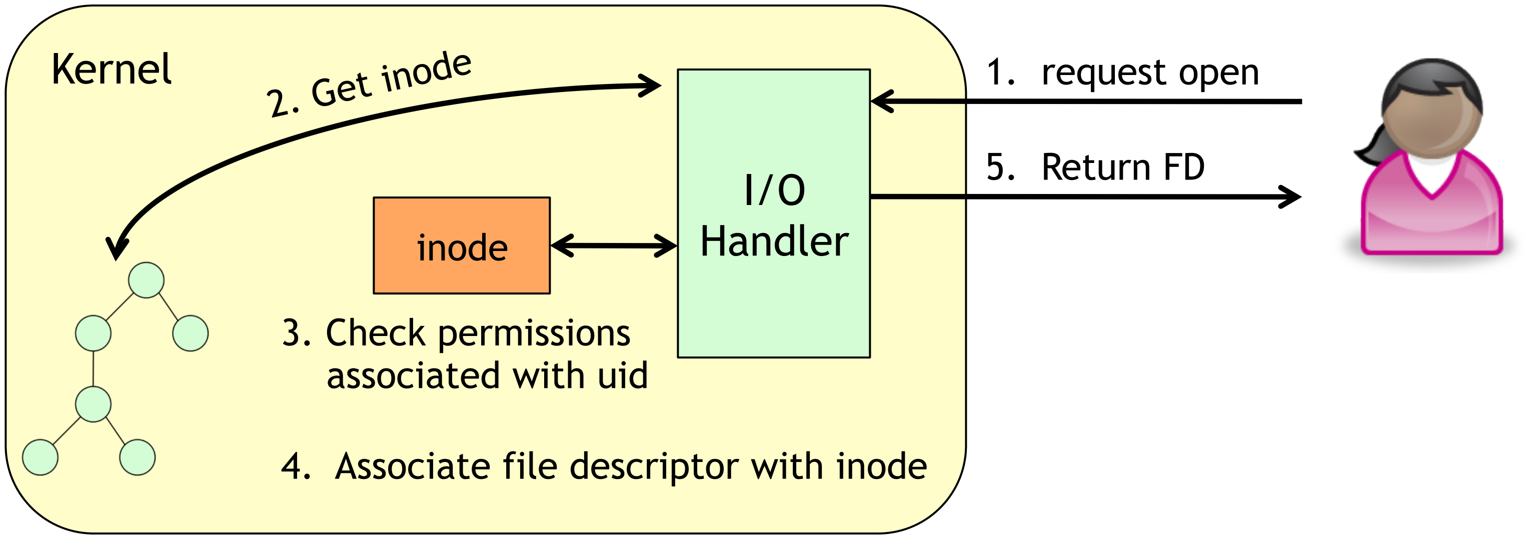
- *cap*(Alice) = {(/etc/passwd, {r}), (Alice_priv.txt, {r,w,x}), (recipes.html, {r})}
- *cap*(Bob) = {(/etc/passwd, {r}), (recipes.html, {r,w,x})}
- *cap*(Charlie) = {(/etc/passwd, {r}), (recipes.html, {r})}

Question: Can you think of any potential problems with capabilities?

- How do we keep users from modifying their own permissions?!?!
- How can the object owner revoke or change a user's permissions?

At one time or another, you have all used a capability system

File access in Unix systems is based on capabilities!


Kernel
2. Get inode
I/O Handler
inode
3. Check permissions associated with uid
4. Associate file descriptor with inode
1. request open
5. Return FD

In this model, the file descriptor---which is really just a random number---acts as a capability that allows some process access to a file

This is a neat idea for (at least) two reasons

- Optimizes the file access process by only consulting ACL once
- Access permissions will not "carry over" if the original file is deleted and another file with the same name is later created

Case Study: The Amoeba Distributed OS

Amoeba is a distributed operating system developed in the 80s and 90s at the Vrije Universiteit in the Netherlands

Amoeba uses an object oriented microkernel

- Clients see many machines as a single computational resource
- Minimal function is placed in the kernel itself
- User-space server objects are allocated resources by the kernel, but manage these resources themselves

Example: The Amoeba file system

- Block server allocates physical storage blocks
- File server allocates blocks for files
- Directory server manages collections of files

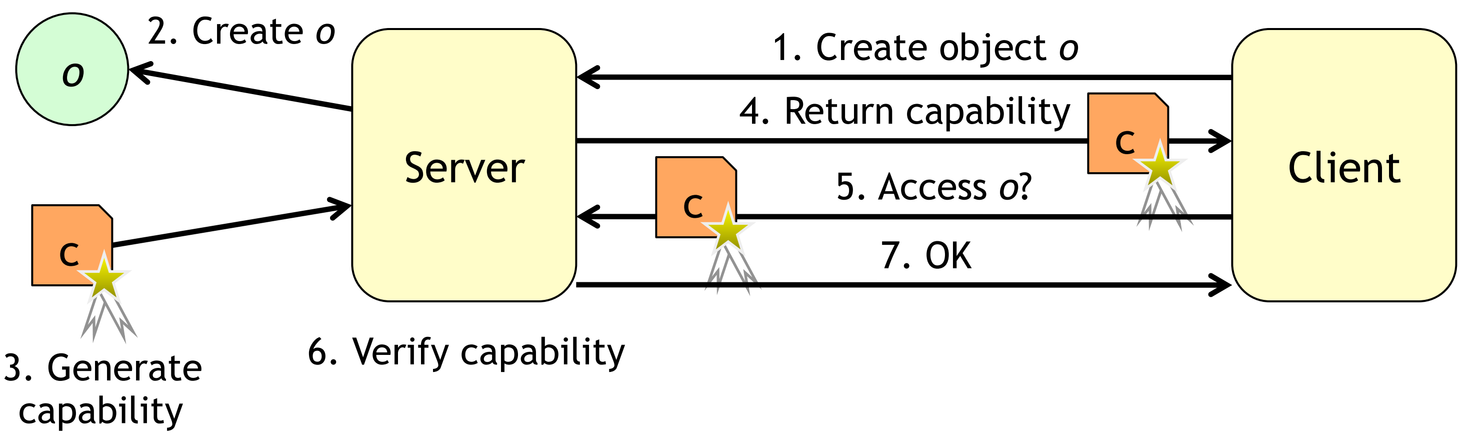
Client processes talk to server objects using (essentially) RPC

Amoeba web page: http://www.cs.vu.nl/pub/amoeba/

Andrew S. Tanenbaum, Sape J. Mullender, Robbert van Renesse, "Using Sparse Capabilities in a Distributed Operating System," ICDCS 1986, pages 558-563.

Without a monolithic kernel to manage permissions, how can Amoeba be secured?

By using capabilities, of course!



What does an Amoeba capability look like?

- Server port (48 bits): Think of this as a network port
- Object number (24 bits): Basically a file descriptor
- Rights field (8 bits): Defines the type of access granted by this capability
- Check field (48 bits): A cryptographic checksum

How are capabilities created?

When an object is created in Amoeba, it is associated with a random 24-bit object number

- This number is used by the server process to manage its objects
- Think of this as an inode number in the Unix file system

The server then generates a random key and associates this key with the object number corresponding to the new object

The capability is then set to contain the following fields

- The server's 48-bit port number
- The randomly generated object number
- The rights allowed to the owner of the object
- H(port || object number || rights $\oplus$ key)

The resulting capability is returned to the client application

How exactly are capabilities used?

To use a capability, the server

- Checks to make sure the capability is set for its port
- Looks up the key corresponding to the object number in the capability
- Verifies that checksum = H(port || object number || rights ⊕ key)

Can a malicious user edit a capability to use it on another server?

- No, because forging the port field will invalidate the checksum

Can a malicious user edit a capability to access another object?

- No, because the wrong key will be looked up

Can a malicious user edit the rights field of her capability?

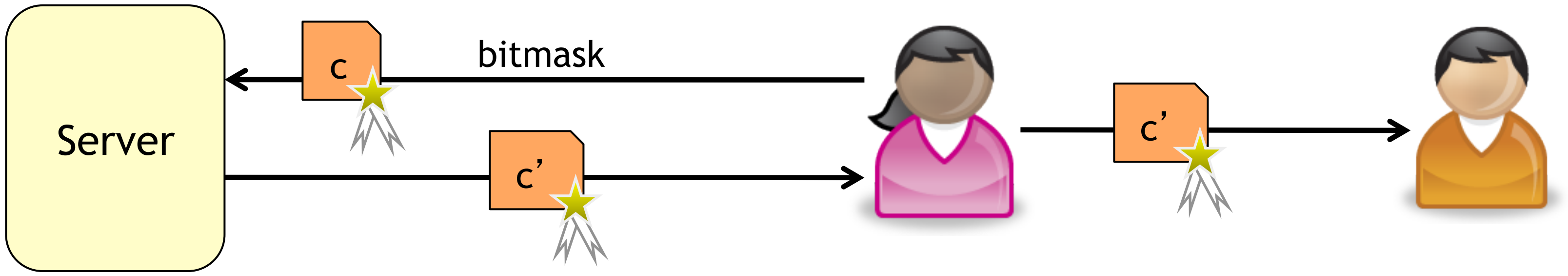
- No, because the XOR will produce an incorrect value

One interesting feature is that capabilities can be delegated to others by their owner

If the owner wants to delegate all of his rights to another user, he can simply send her a copy of the capability!

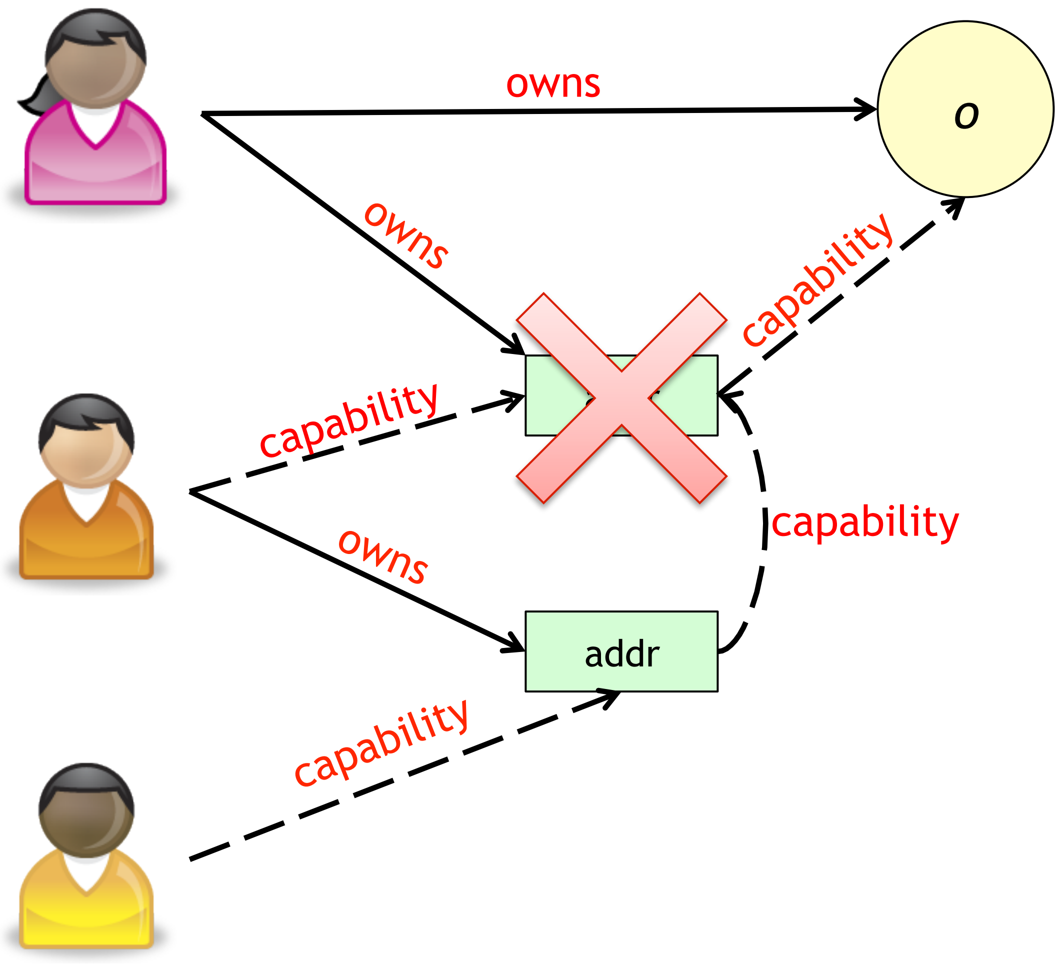
The owner can also delegate partial rights to another user

- Client sends the original capability to the server, along with a bitmask used to attenuate the rights contained in the capability
- Server creates a new capability
  - rights' = rights $\wedge$ bitmask
  - checksum = H(port || object number || rights' $\oplus$ key)
- This new capability is returned to the user, who can then pass it along



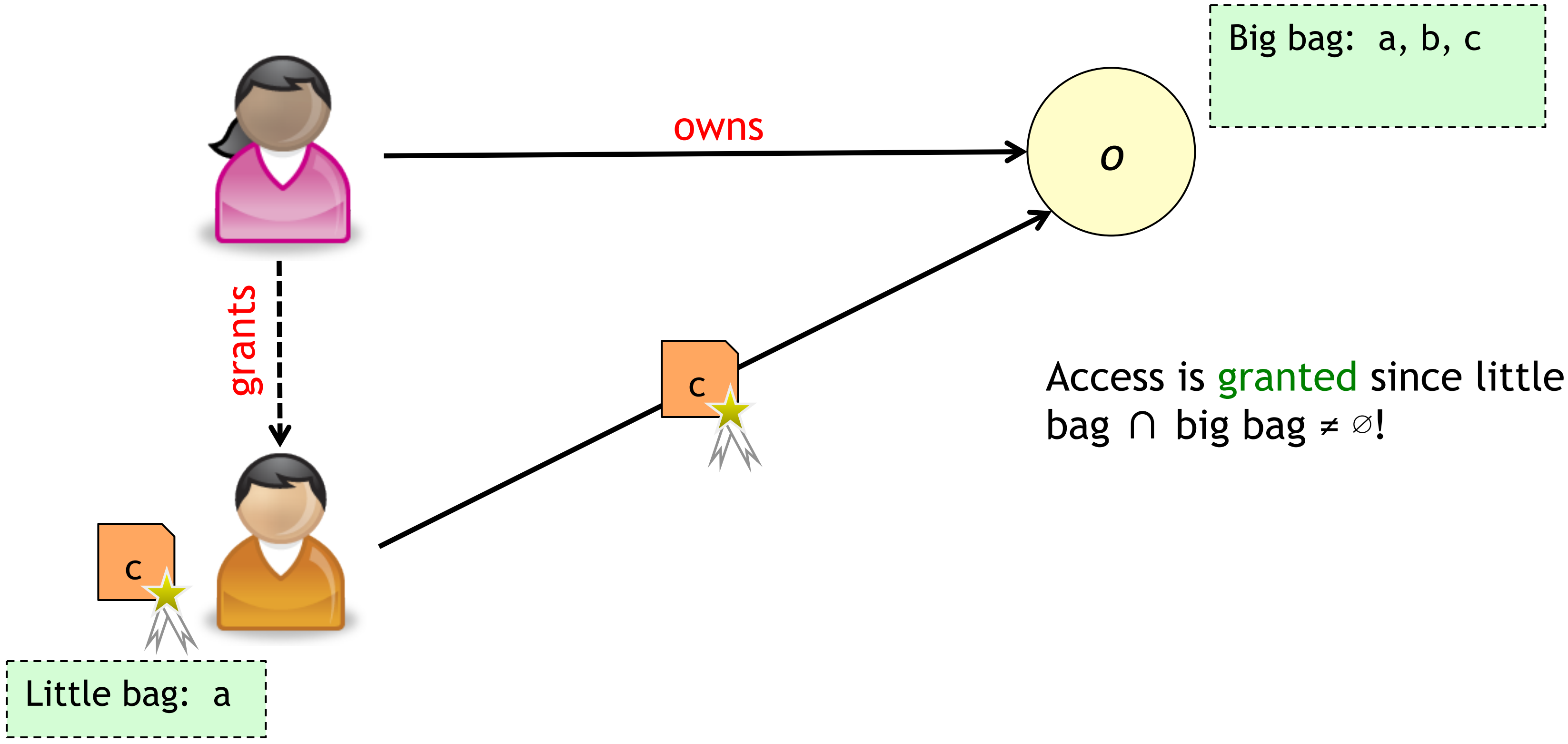
How can capabilities be revoked?

The simplest method is to use indirection by treating capabilities as addresses



Another approach to revocation is to use “bags”

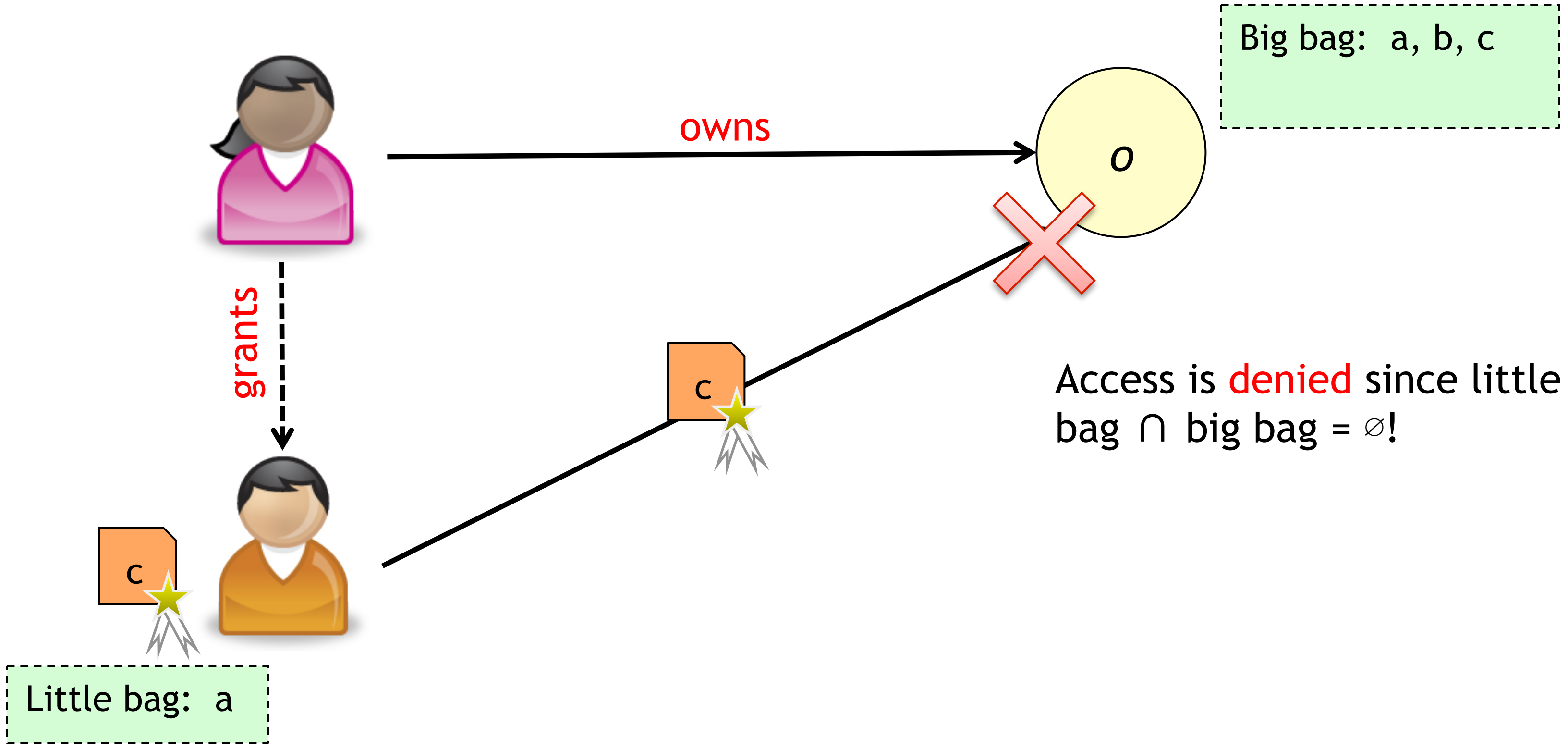
Intuition: We can simplify revocation by including extra state information



Access is granted since little bag ∩ big bag ≠ ∅!

Another approach to revocation is to use "bags"

To revoke Bob's access, Alice simply removes items from the big bag of the object *o*, which she can do since she is the owner



This approach is very simple to implement, but does not allow for constrained delegation without the involvement of the object's owner

How does Amoeba handle revocation?

Amoeba uses a variation on the indirection approach

Recall that all capabilities refer to a specific object ID

To revoke a capability:

- The object owner makes a request to the server to change the object ID
- The server changes the object ID and creates a new capability
- The new capability is returned to the object owner

Question: How does this process accomplish revocation?

- The object ID of any old capability cannot be changed, as this would invalidate the cryptographic checksum!

Question: What are the limitations of this approach?

ACLs vs. Capabilities

When dealing with access controls, two questions are often asked:

1. Given a subject, which objects can it access and how?
2. Given an object, which subjects can access it and how?

In theory, either ACLs or capability lists can be used to answer either of these questions

In practice, the first question is more easily answered using capabilities

- Everything is centralized with the user

The second question is more easily answered using ACLs

- The difficulties of capability revocation also apply to capability review, which is needed to answer this question

In the end, the decision of whether to use ACLs or capabilities will be influenced (to some degree) by which of these two questions needs to be answered more often

Conclusions

In the access control matrix model, access control is performed based upon the rights assigned to an authenticated user

ACLs and capabilities simplify the management of the ACM

- ACLs are essentially columns of the ACM
- Capability lists are rows of the ACM

ACLs are used more often than capabilities, but both are widely used

Trade-offs between ACLs and capability lists

- Ease of management
- Revocation and review
- Efficiency
- …