Applied Cryptography and Network Security CS 1653

Summer 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

Announcements

- Please schedule Project Phase 3 Demo with Pratik as soon as possible
- Phase 3 Peer Evaluation Survey is up on CATME
  - Due by next Friday
- Homework 8 due this Friday @ 11:59 pm
- Homework 9 due next Friday @ 11:59 pm
- Programming Assignment 2 due next Friday
- Project Phase 4 Due on 7/31 @ 11:59 pm
  - Teams must meet with me on or before Thursday 7/27
- Midterm, homework, and phase 1 and 2 grades will be posted by the end of this week

Breaking Crypto!

- Breaking PKI by breaking collision resistance of MD5
- Brute force attacks without using brute force
  - SSL key generation
  - Kerberos v4
- Brute force attacks against symmetric key ciphers
  - massively parallel attack
  - Deep Crack
- Subverting public key cryptography protocols
- Attacking real world implementations
  - Timing analysis of RSA
  - Power analysis of DES
  - Keyjacking cryptographic APIs

Case study: Kerberos v4

Kerberos is a popular network authentication protocol

- Initially developed at MIT as part of project Athena
- Used within the department for AFS “tokens”

Note: Kerberos is available as a widely-used and widely-studied open-source software package

The security of Kerberos v4 depends on 56-bit DES or 112-bit TDES keys

In 1997, a group of grad students at Purdue showed that Kerberos v4 used the XOR of several 32-bit quantities to seed its PRNG

- Timing information
- PID and PPID
- Count of keys generated
- Host ID

So what does this tell us?

Observation: The XOR of any number of 32-bit quantities is 32 bits long!

- In 1997, brute forcing this 32-bit seed space only took about 28 hours!

As if that wasn't bad enough, many of these quantities are predictable

- Timing information can be guessed
- Process IDs can be read if the attacker has an account on the system
- The host ID is known
- …

The result of the above is that the actual entropy of the 56- or 112-bit keyspace is reduced to 20 bits!

- This can be brute forced in a few seconds…

The moral of the story: Not even open-source software that has been scrutinized by lots of smart people is totally free of errors

An interesting aside…

This weakness was first discovered nearly a decade earlier (1988)

After this initial discovery, a stronger PRNG was written and checked into the Kerberos source tree, but was never actually used!

- The PRNG was written as a new function (not a replacement)
- Extensive use of `#define` statements in the code obscured which PRNG was actually being used

The result: Good code was available, but bad code was used…

The lesson? Don't get too cute and write incomprehensible code!

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- massively parallel attack
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Surely, breaking *some* systems must require a brute force search of their keyspace, right?

If all else fails, any cryptosystem can be broken by "simply" decrypting a ciphertext with all possible keys

Recall: NSA reduced Lucifer's keyspace when developing DES

- Lucifer was a 64-bit cipher, DES is a 56-bit cipher
- Many cryptographers began to speculate as to why this was done

Certainly, brute forcing a 56-bit space on a single computer would take a very long time, but why use just a single computer?

Quisquater and Desmendt discussed the possibility of a massively parallel attack

- Key-cracking hardware could be built into TVs and radios
- State-run media could farm out searches to these devices
- A back of the envelope example:
  - Keys tested at 1 million/second
  - 1 Billion TVs/radios
  - Break a DES key in about a minute…

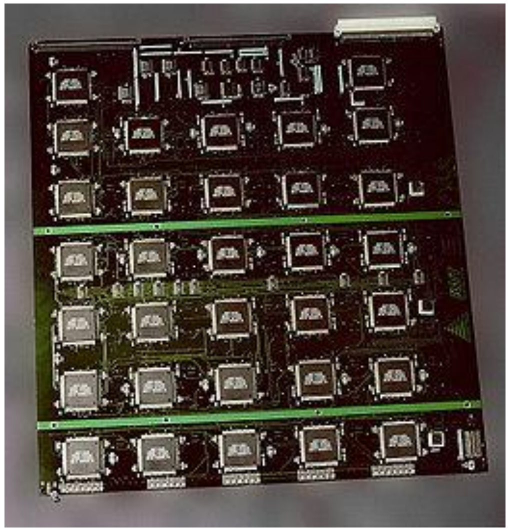
Deep crack is a slightly less "big brother" version of this principle

The Electronic Frontier Foundation (EFF) is a digital rights advocacy group based in the US

To demonstrate that 56-bit keys are not secure enough for widespread use (despite the government's claims to the contrary), the EFF built a machine called Deep Crack

Deep Crack is an extremely parallel machine (esp. by 1997 standards)

- 29 individual circuit boards
- 64 specially-designed DES-cracking chips per board
- 1,856 chips running in parallel could test about 90 billion keys per second!
- The entire DES key space could be tested in about 9 hours



With a price tag of about $250,000, Deep Crack was well within the budget of many criminal organizations and/or enemy governments

Take-home message

Definitions of words like “big”, “infeasible”, and “difficult” are rooted in hardware, software, or algorithmic assumptions. Make sure to revisit these assumptions often!

e.g., check keylength.com

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- massively parallel attack
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Is it possible to subvert public key systems without attacking the mathematics?

Question: Say that Alice runs a digital notary. We want Alice to sign the message M for us, but do not want her to see M.

Surprisingly, we can trick Alice into doing this!

- Let (n, e) be Alice's public key
- Let d be Alice's private key
- Let X be a random number that we pick
- Let $Y = X^e \bmod n$
- Let $M_D = YM \bmod n$ be a "decoy message"

M_D is a "randomized" version of M

Now, we get Alice to sign M_D, returning $U = M_D^d \bmod n$

Note:
$$\begin{aligned}
UX^{-1} \bmod n &= M_D^d X^{-1} \bmod n && // \; U = M_D^d \\
&= Y^d M^d X^{-1} \bmod n && // \; M_D^d = (YM)^d = Y^d M^d \\
&= X^{ed} M^d X^{-1} \bmod n && // \; Y = X^e \\
&= X M^d X^{-1} \bmod n && // \; X^{ed} = X \\
&= M^d \bmod n && // \; XX^{-1} = 1
\end{aligned}$$

Alice signed M without knowing it!

If Alice also decrypts using her signing key, we can read her private messages!

How? Let:

- $C = M^e \bmod n$ be a message encrypted to Alice
- R be a random number
- $X = R^e \bmod n$
- $Y = XC \bmod n$

Now, we ask Alice to sign Y, which gives us $U = Y^d \bmod n$

Note: $R^{-1}U \bmod n = R^{-1}Y^d \bmod n$ // $U = Y^d \bmod n$

$= R^{-1}X^dC^d \bmod n$ // $Y = XC \bmod n$

$= R^{-1}R^{ed}C^d \bmod n$ // $X = R^e \bmod n$

$= R^{-1}RC^d \bmod n$ // $R^{ed} \bmod n = R \bmod n$

$= M^{ed} \bmod n$ // $RR^{-1} = 1, C = M^e \bmod n$

$= M \bmod n$ // $M^{ed} \bmod n = M \bmod n$

This is probably not a good thing…

Why do these attacks work?!

RSA has what is known as a multiplicative homomorphism

- $(X^Z \bmod n)(Y^Z \bmod n) = XY^Z \bmod n$
- i.e., $E(x)E(y) = E(xy)$ if E is the RSA encryption function

The attacks that we just talked about used RSA to operate on raw data

- Data is represented as plain old numbers
- Each number encrypted directly

In real life, RSA is not used this way! Padding functions like OAEP (aka PKCS#1) are used to randomly pad message prior to encryption or signing

- i.e., M becomes $P(M)$
- $P^{-1}(P(M)) = M$
- $P^{-1}\left(D\left(E(P(M))\right)\right) = P^{-1}(P(M)) = M$

Note: Given $E(P(M_1))$ and $E(P(M_2))$, we can calculate $E(P(M_1)P(M_2))$.

However, $P^{-1}\left(D\left(E(P(M_1)P(M_2))\right)\right) = P^{-1}(P(M_1)P(M_2)) \neq M_1M_2$

Take home message

Breaking public cryptography does not need to involve "breaking" mathematics. More often than not, misusing a protocol or two can be just as effective!

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- massively parallel attack
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Successive squaring or Square-and-Multiply

```
// Goal:  Calculate m^d mod n
int pow(m, d, n)
    bitvector b[] = binary representation of d
    int r = 1
    for(int i=0; i <= b.length(); i++)
        r = r*r mod n
        if(b[i] == 1) then r = r * m
    return r
```

MSB in b[0]

Note that $5_{10} = 101_2$

Example: Computing pow(2, 5, 64)

Iteration	r
0	1
1	$1^2 \times 2 = 2$
2	$2^2 = 4$
3	$4^2 \times 2 = 32$

Observation: This algorithm does more work when bits are set to 1

We can leverage this observation to launch an attack on implementations of RSA!

In 1995, Paul Kocher described and demonstrated a timing attack against square-and-multiply implementations of RSA

The gist: If we know C, we don't know d, and we can measure the time that the operation C^d takes, we can eventually learn d by observing many such operations

So, why does this work?

- We can make a guess for the first bit of d
- If our guess is correct, we can predict
  - The intermediate result r
  - How long the algorithm will take
- If our guess is incorrect, we don't learn anything

That only explains one round, how does this work for a large key?

Details, details, details…

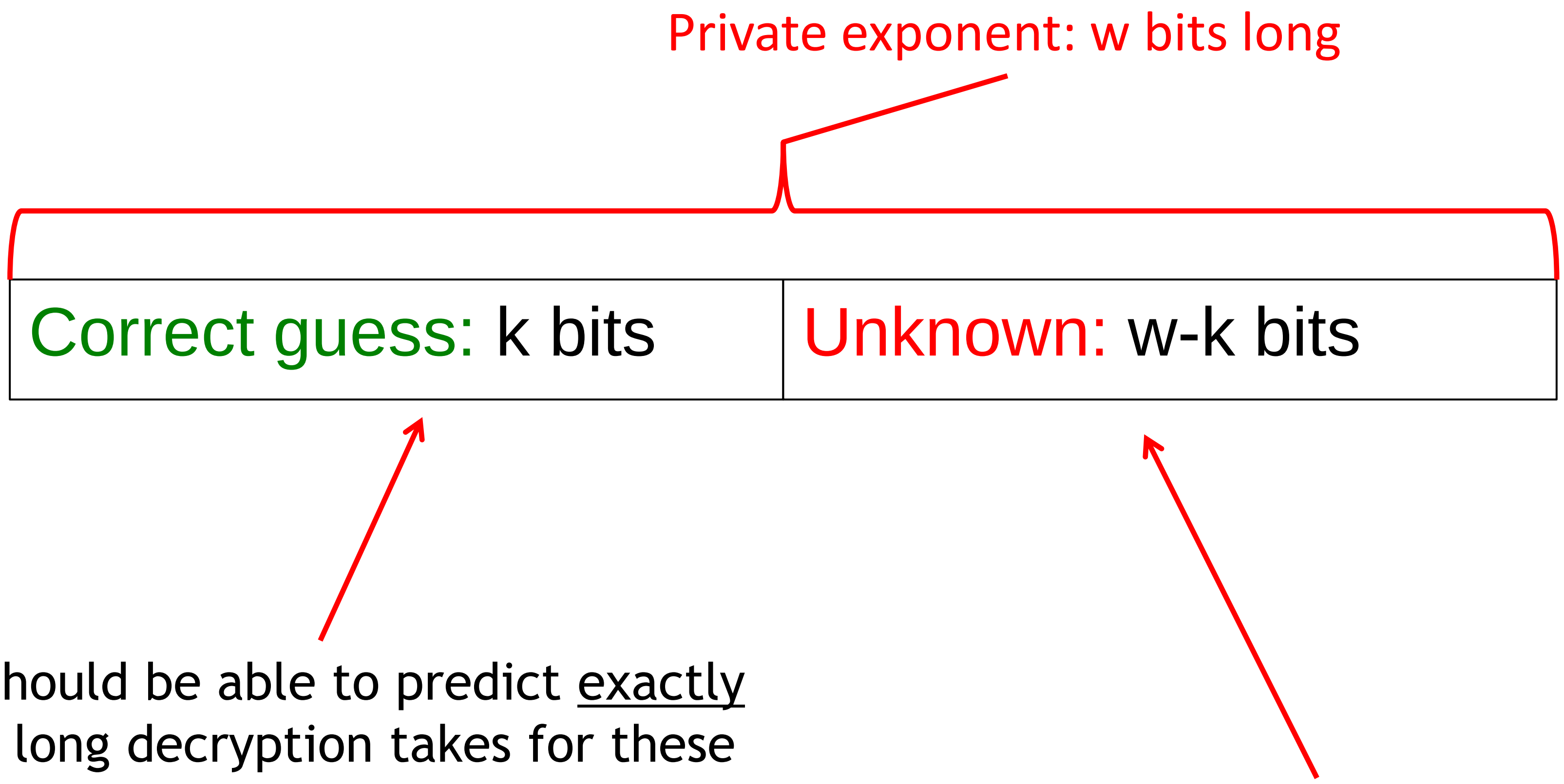
Consider the following observations, assuming a w-bit key:

- The time a decryption operation takes is the sum of the individual times for each of the w bits
- If we can correctly guess the first k bits of the key
  - We can calculate the time required for the first k operations
  - The remaining terms in the sum are essentially random
- Over a large enough sample of C, the difference between how long we calculate C^d to take versus how long C^d actually takes is a distribution whose variance is proportional to w-k

In short, by guessing correctly, we can reduce the variance of this distribution and learn the key k

Furthermore, this method allows us to learn the private key in O(w) time, rather than $O(2^w)$ time

Graphically…



We should be able to predict <u>exactly</u> how long decryption takes for these bits of the private exponent.

Our error in predicting the runtime of the algorithm for these bits is the sum of w-k random variables (one per bit)

This seems really esoteric…

Sure, I'm convinced that this is possible, but how useful is this attack likely to be in practice?

In 2003, Brumley and Boneh showed how to successfully launch timing attacks against OpenSSL-based Apache web server over the Internet!

Check also the attack on the Enigma Machine in WWII (https://www.youtube.com/watch?v=RzWB5jL5RX0)

The lesson? Over time, attacks only get better!

If timing information is useful, what can we do with other sources of information?

Obvious statement: Software runs on hardware, hardware requires power

- Algorithms leak information via the power that they consume
- By monitoring the power consumption of hardware, we may be able to learn something about the inputs to the algorithms running
- Keys are important inputs to cryptographic algorithms!

Kocher et al. demonstrated a viable attack for learning the DES keys used by certain types of smart cards

Their attack was very clever

- Recall that DES keys are 56 bits long
- They observed an odd series of 56 spikes in the power trace
  - Why? Software checking the parity of the DES key
  - Spikes were at one of two heights, corresponding to 1 or 0 in the key
- By watching one encryption or decryption, they learned the key!
- This is simple power analysis

Video-based Cryptanalysis of Power LEDs (https://www.nassiben.com/video-based-crypta)

Even a good implementation and solid hardware can be subverted by a bad API…

Most often, cryptographic routines are not built directly into applications, but rather exist as a library that is used by applications

So-called keyjacking attacks work as follows:

1. Run a malicious executable on the victim's machine
2. Use this executable to begin intercepting calls to the victim's crypto API

This allows the attacker to do all kinds of nasty stuff

- Export secret/private keys to other applications
- Decrypt private messages
- Forge signatures
- …

These attacks are very hard to defend against, as once the system is compromised (e.g., via a buffer overflow exploit), there is nothing much that can be done!

Take home message

Cryptographic hardware exists where people can observe it. Cryptographic software may leak information about the inputs that it uses. Our nice little black boxes need to exist in the real world, and are thus subject to scrutiny and attack.

Breaking Crypto Conclusions

Treating cryptography as a black block simplifies our lives as engineers, but gives us a reason to ignore certain types of threats

Today we saw that

- Sometimes keys can be guessed without a brute force attack
- Sometimes brute force attacks are tractable by thinking outside of the box
- Naive use of public key cryptography can lead to undesirable outcomes
- Engineering decisions when building cryptosystems can be used to break these cryptosystem when they are deployed

As a result, it is important to keep up to date on how to safely use modern cryptography

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 Discretionary Access Control (DAC) models
- In Mandatory Access Control (MAC) models, we need the 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

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

Let's look at an example…

objects (entities)

subjects

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

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) network firewall?

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

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

Formally:

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

objects (entities)

subjects

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

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 (typically sparse) matrix can be expensive
- 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

Movie bouncers are typically shown enforcing by ACL

- "Sorry, you're not on the list."

More formally, an access control list can be represented as a function $acl: O \rightarrow \wp(S \times \wp(R))$. 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 more efficient versions of the ACL model

Example: In Unix, users are classified into groups

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

- skhattab : x : 87 : 7 : Sherif Khattab : /afs/cs.pitt.edu/usr0/skhattab : /bin/dash

Username uid default gid

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

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

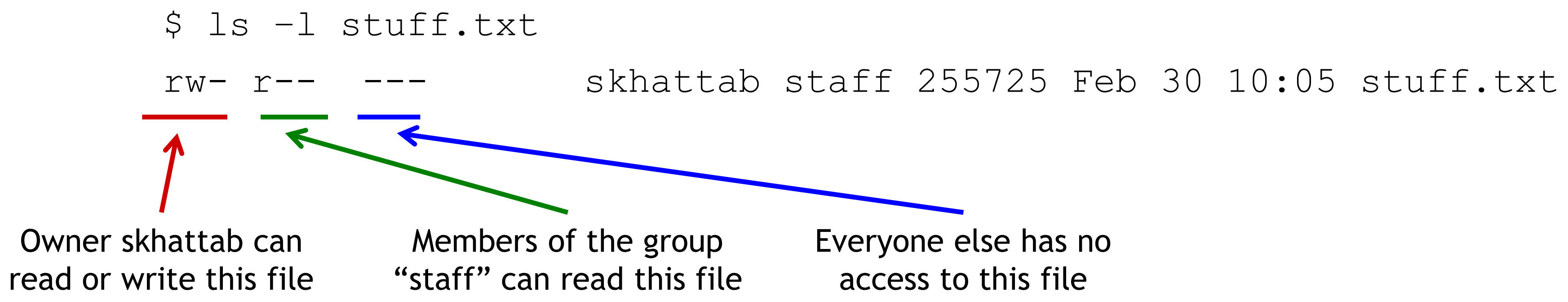
Group name gid List of users belonging to this group

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--  ---       skhattab staff 255725 Feb 30 10:05 stuff.txt
```

Owner skhattab 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 allows a richer access control model than traditional 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 basic Unix permissions, but not fundamentally different.

- Modern Unix/Linux systems now also have more expressive ACLs

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

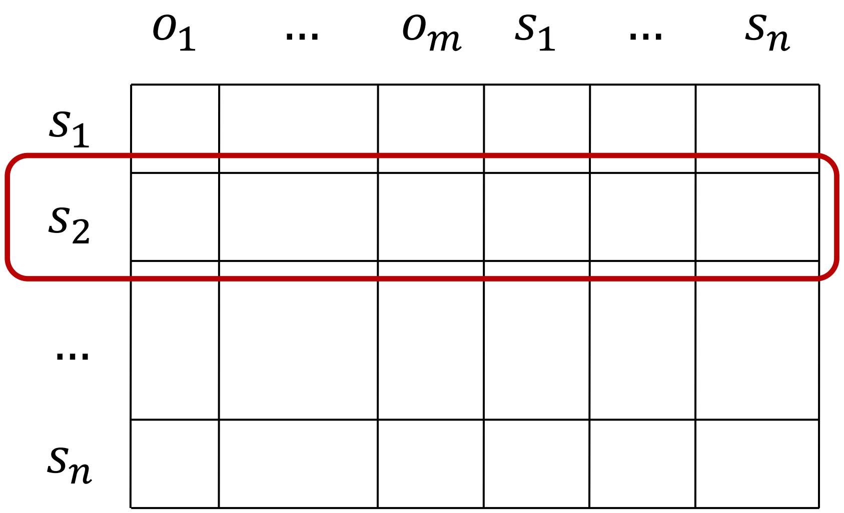
4. How are contradictory access control permissions handled?
   - *Traditional Unix*: First match applies
   - *Windows*: Deny overrides

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 $cap : S \rightarrow \wp\big(O \times \wp(R)\big)$. Given an subject s, $cap(s)$ returns a set of (o_i, r_j) pairs such that subject s can access object o_i using any right in the set r_j.

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:

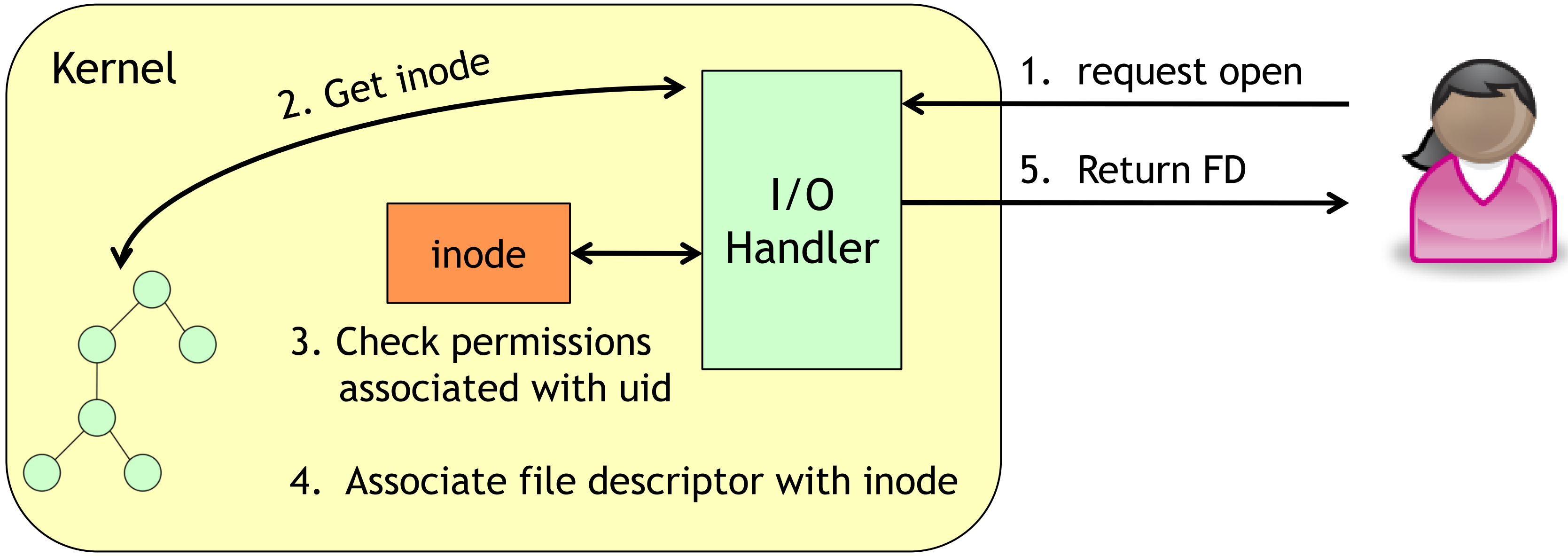
- *cap*(Alice) = {(/etc/passwd, {r}), (Alice_priv.txt, {r,w,x}), (recipes.html, {r})}
- *cap*(Bob) = {(/etc/passwd, {r}), (recipes.html, {r,w,x})}
- *cap*(Charlie) = {(/etc/passwd, {r}), (recipes.html, {r})}

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

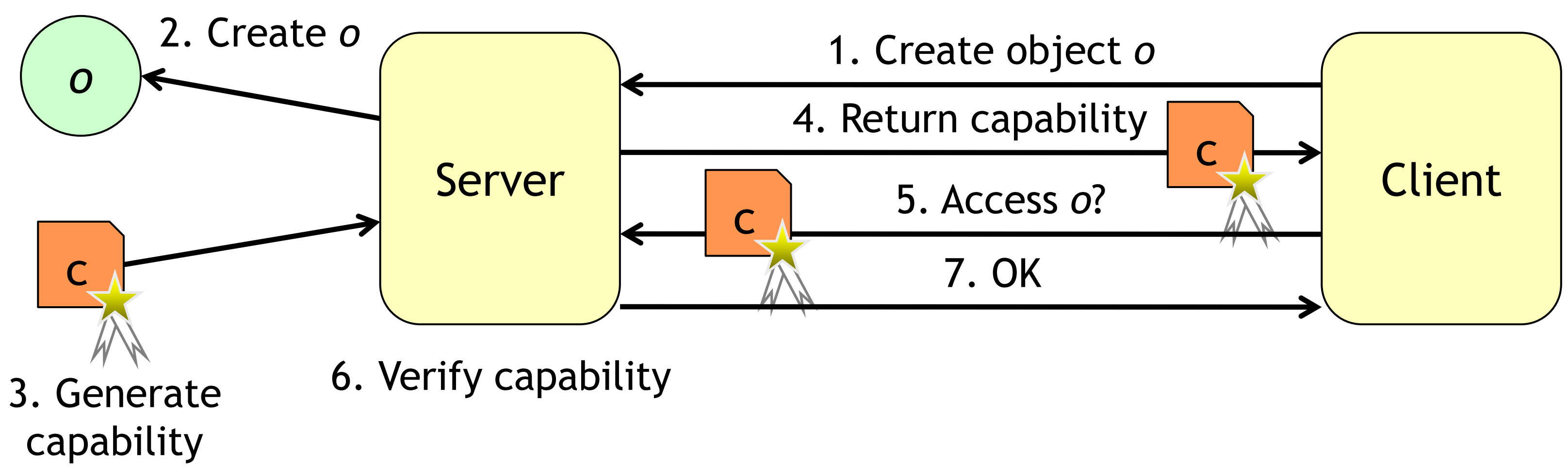
Client processes talk to server objects using (essentially) RPC

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
- $H(\text{port} \mid\mid \text{object number} \mid\mid \text{rights} \oplus \text{key})$

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 $\oplus$ 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?

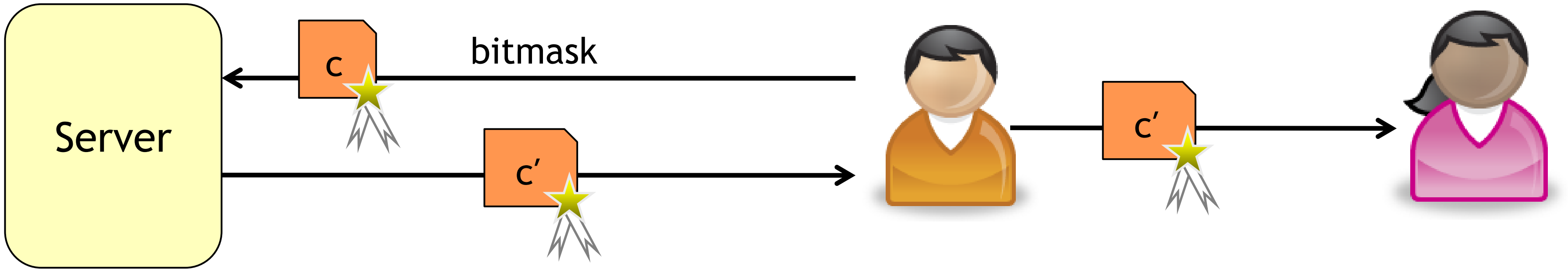
- No, because the XOR will produce an incorrect value

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!

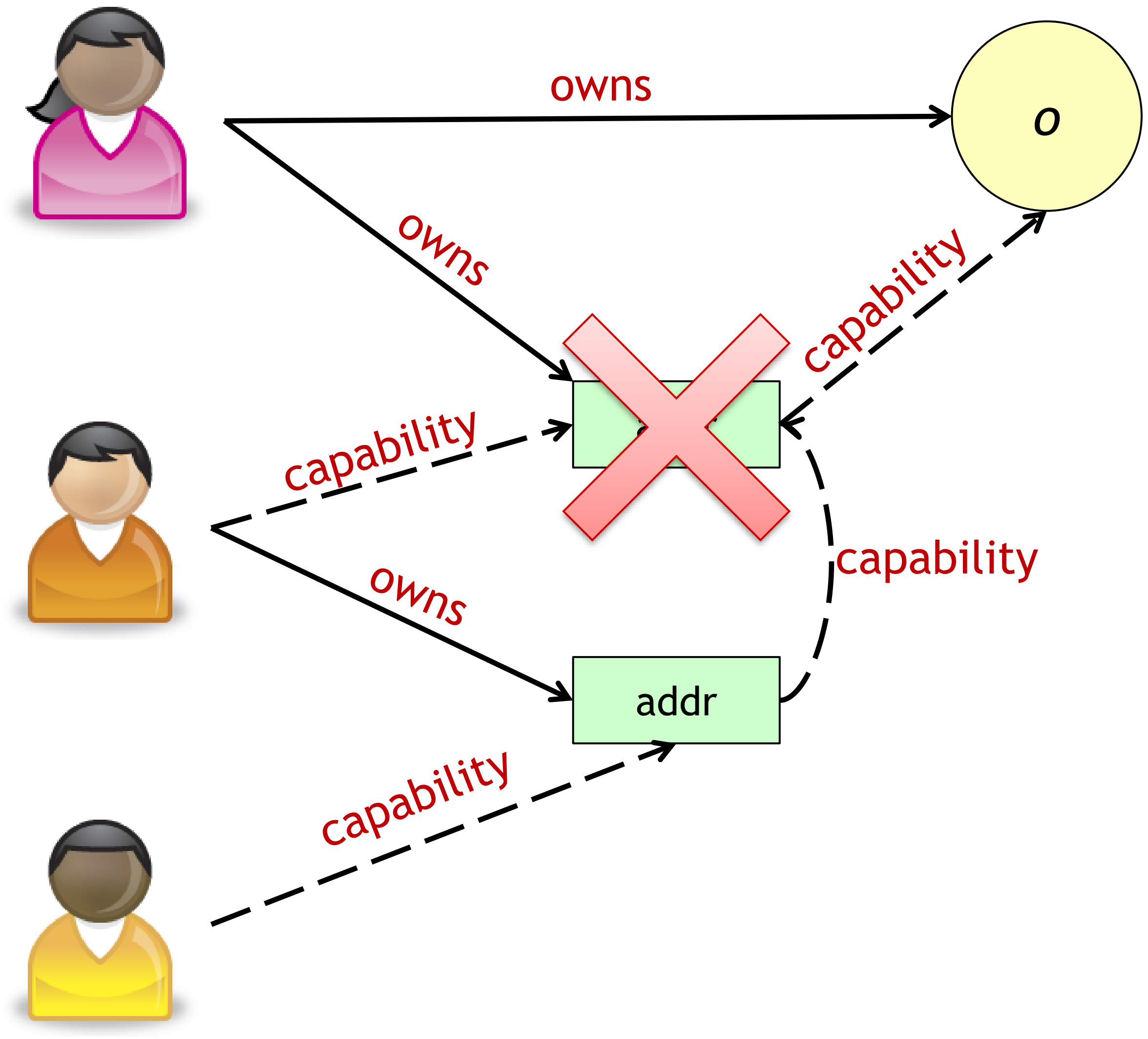
The owner can also delegate partial rights to another user

- 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



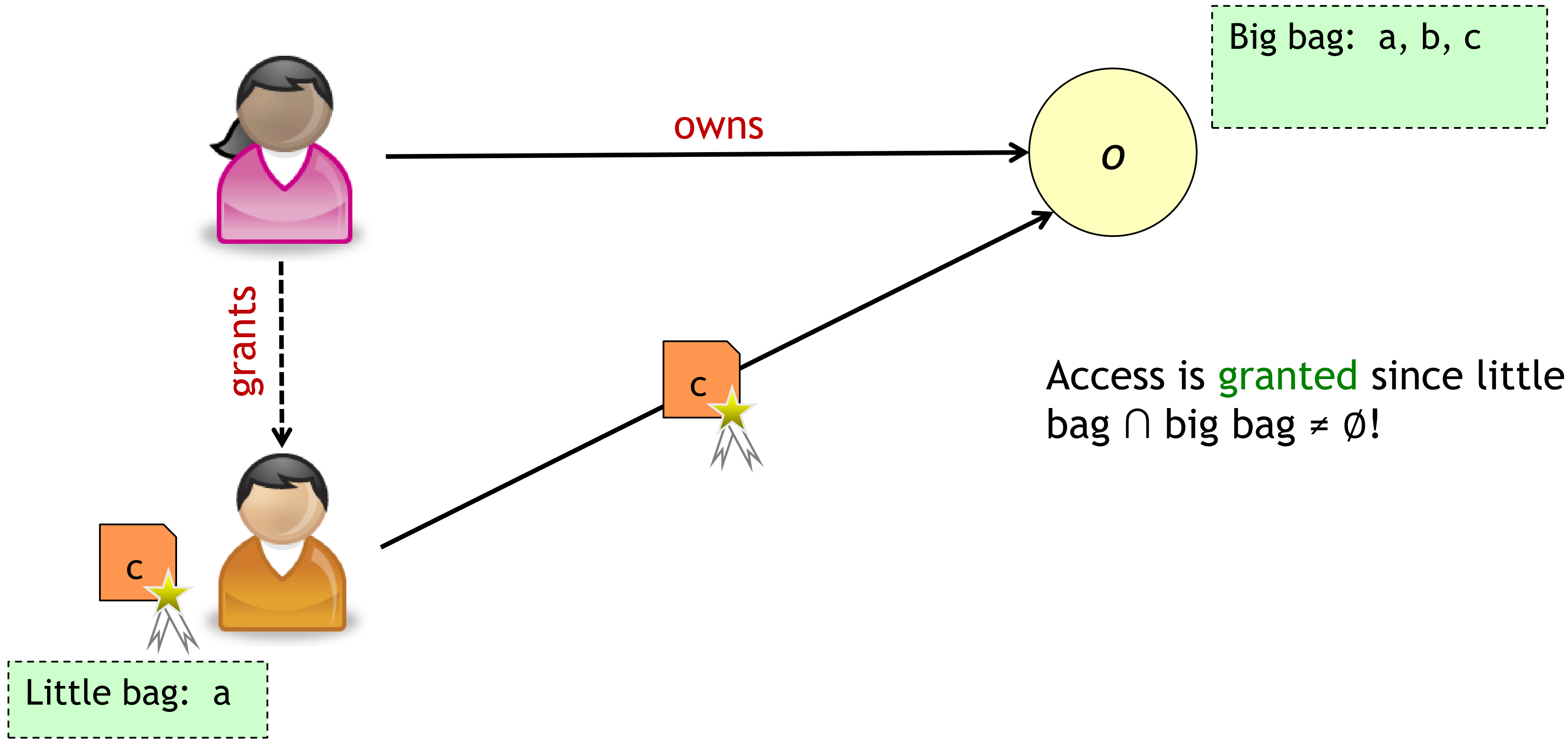
How can capabilities be revoked?

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



Another approach to revocation is to use “bags”

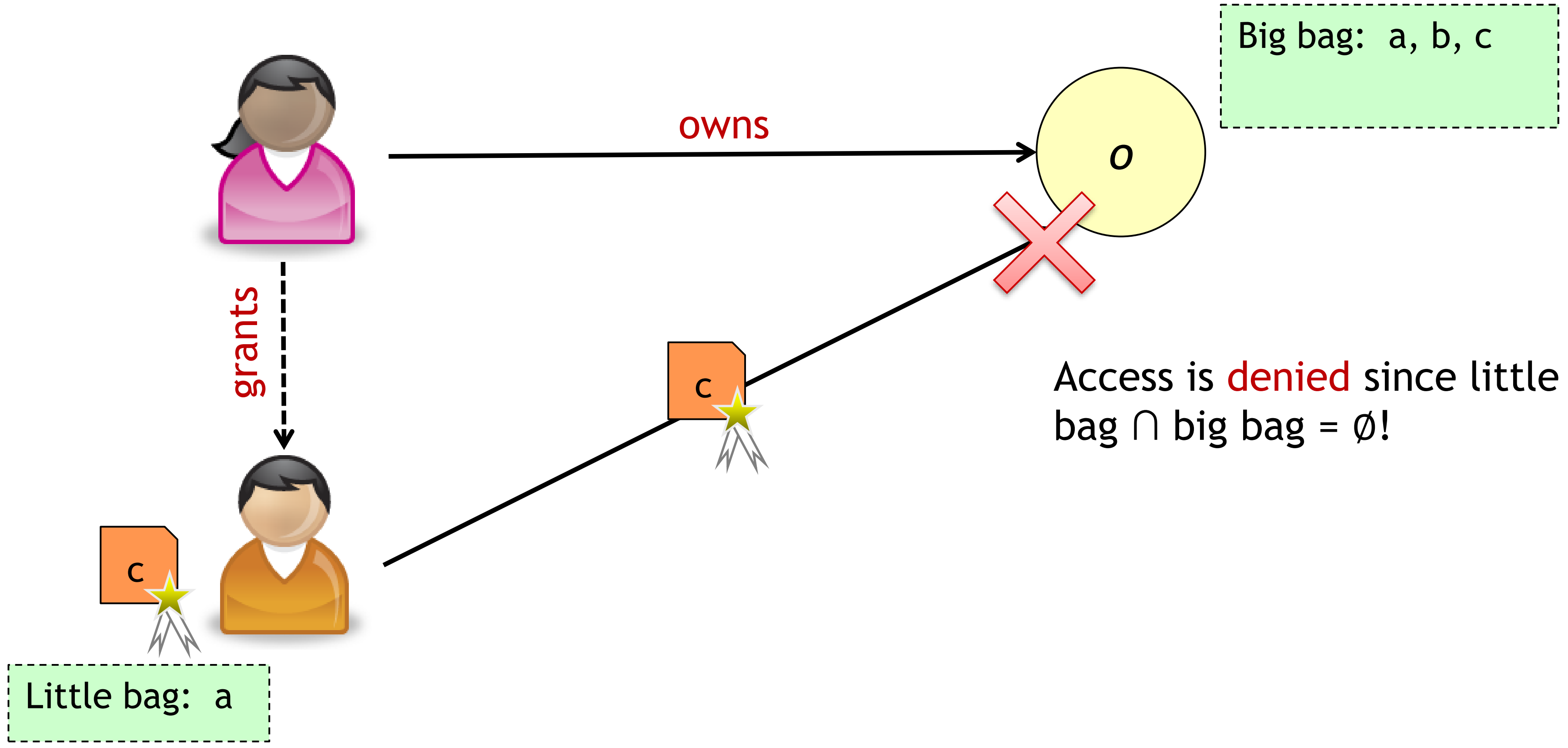
Intuition: We can simplify revocation by including extra state information



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



Access is denied since little bag ∩ big bag = ∅!

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

Summary

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
- …

Next: OS and application security; Viruses and worms