

Access control

Access control

Access control is prominent at many levels:

- memory-management hardware,
- operating systems, file systems, and the like,
- middleware,
- applications,
- firewalls,

and also in physical protection.



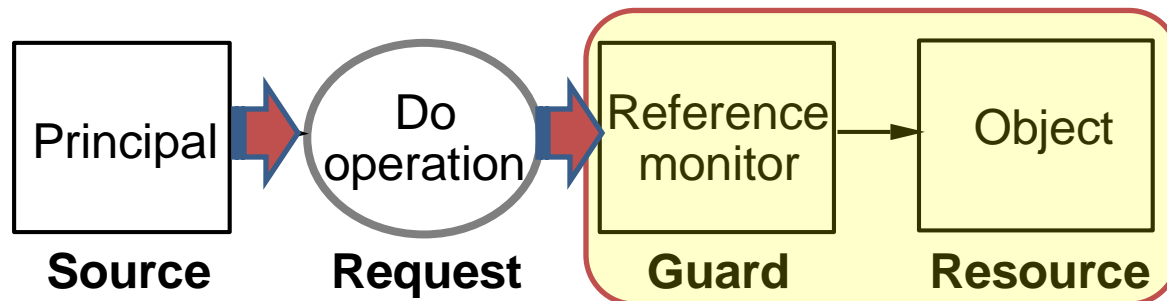
Access control (cont.)

- Access control is a **mechanism**.
 - It aims to guarantee secrecy, integrity, and availability properties, and more.
- Access control can also be seen as a **model**, as specification for lower-level mechanisms.
 - (Higher-level policies are often not explicit.)



The access control model

- Elements:
 - **Objects** or resources
 - **Requests**
 - Sources for requests, called **principals** (or **subjects**)
 - A **reference monitor** to decide on requests



An access control matrix

[Lampson, 1971]

objects principals	file1	file2	file3	file4
user1	rwX	rw	r	X
user2	r	r		X
user3	r	r		X

Implementing access control

Authentication

Access control depends on **authentication**:

- Access control (authorization):
 - Is principal A trusted on statement s ?
 - If A requests s , is s granted?
- Authentication:
 - Who says s ?

Other machinery

- Auditing
- Recovery
- ...

The reference monitor and mediation

The principle of complete mediation

[Saltzer and Schroeder, 1975]

Every access to every object must be checked for authority.

This principle can be enforced in several ways:

- The OS intercepts some of the requests. The hardware catches others.
- A software wrapper / interpreter intercepts some of the requests. (E.g., as in VMs.)

Strategies for representing an access control matrix

In practice, a matrix is typically represented in terms of ACLs and capabilities.

- **ACL**: a column of an access control matrix, attached to an object.
- **Capability**: (basically) a pair of an object and an operation, for a given principal.
It means that the principal may perform the operation on the object.

More on ACLs

objects	file1	file2	file3	file4
principals				
user1	rwX	rw	r	x
user2	r	r		x
user3	r	r		x

- An ACL says which principals can access a particular object.
 - It is a column of an access control matrix,
 - typically maintained “near” the object that it protects.
- ACLs can be compact and easy to review.
- Revoking a principal can be painful.

More on capabilities

objects \ principals	file1	file2	file3	file4
user1	rwX	rw	r	x
user2	r	r		x
user3	r	r		x

- An alternative is to associate capabilities with each principal.
 - A capability means that the principal can perform an operation on an object.
- These capabilities form a row of an access control matrix for the principal
- Capabilities are often easy to pass around (so they enable delegation).
- They can be hard to review and to confine.

Implementing capabilities

⇒ *Principals should not be allowed to forge capabilities.*

This leads to implementations of capabilities

- stored in a protected address space, or
- with special tags with hardware support, or
- as references in a typed language, or
- with a secret, or
- with cryptography, e.g., certificates.

ACLs vs. capabilities

- ACLs and capabilities are dual.
- Both can yield practical implementations of access matrices.
- In actual systems, they are often combined.

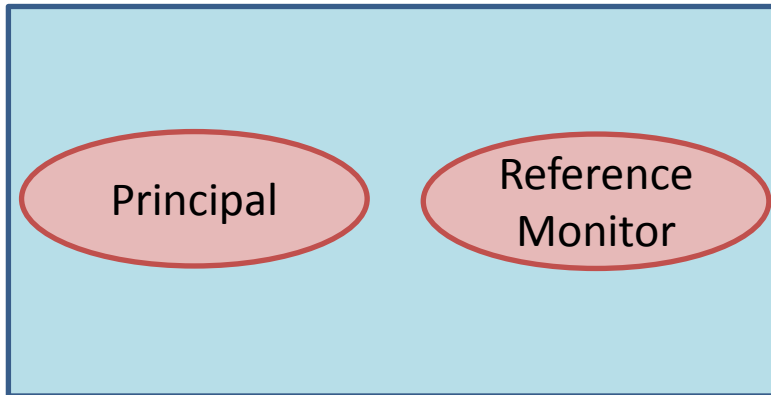
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

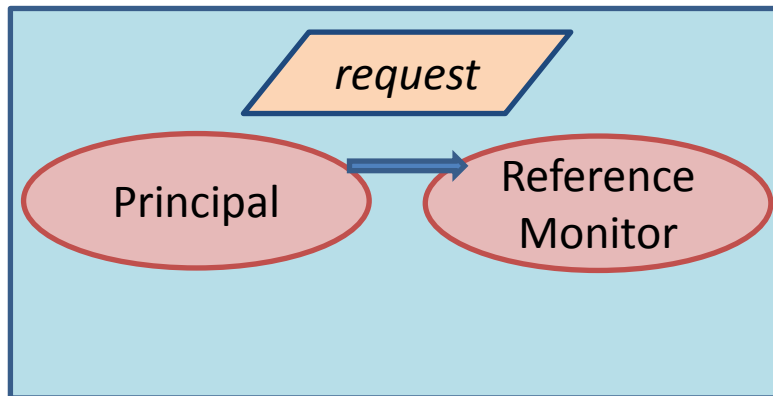
push: Principals present evidence
with requests



push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

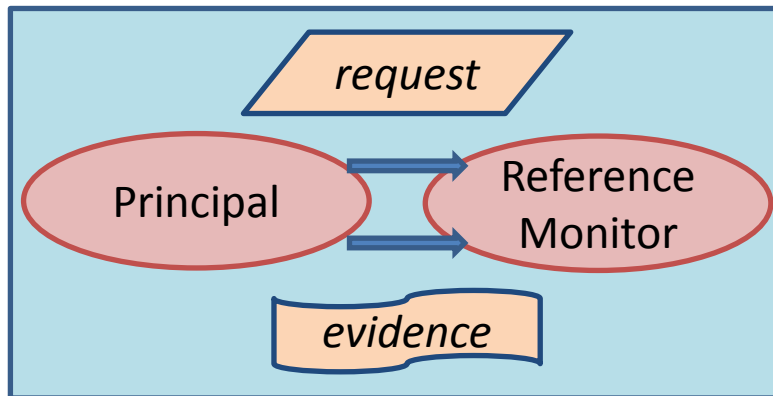
push: Principals present evidence
with requests



push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

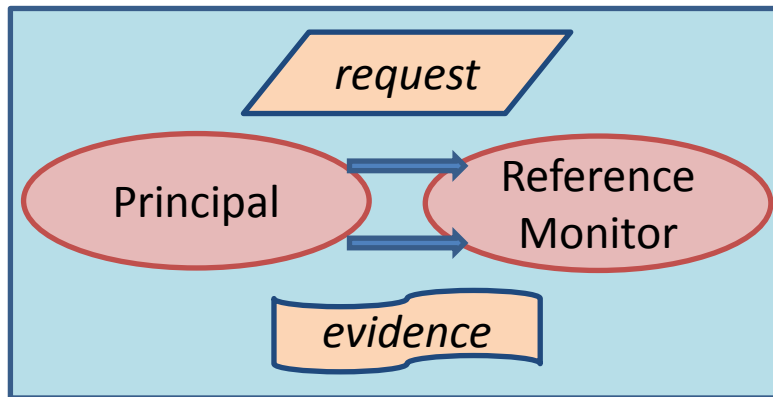
push: Principals present evidence
with requests



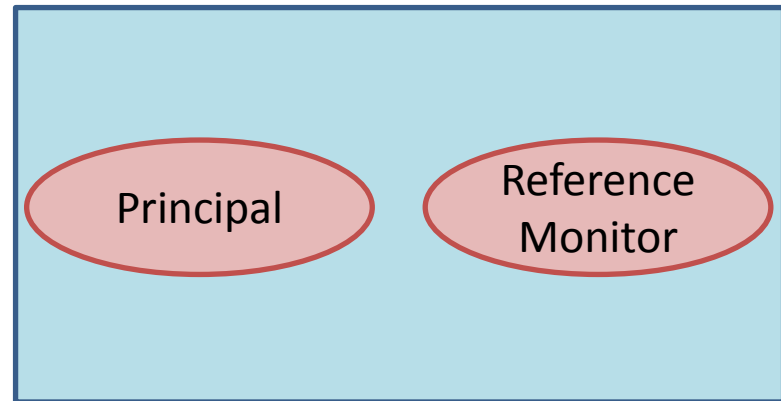
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



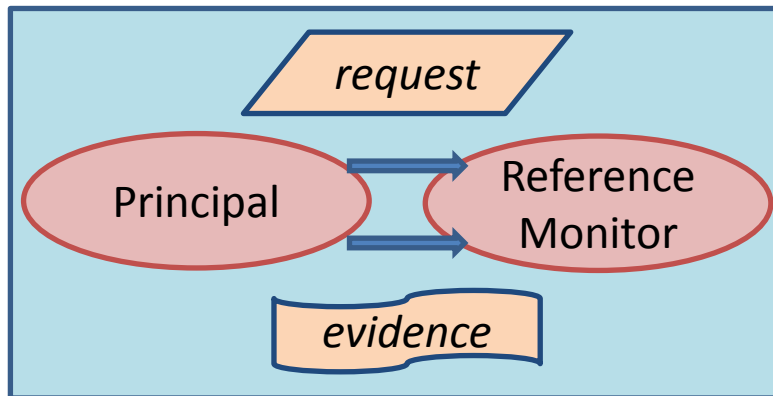
pull: Reference monitors gather evidence, with help from others.



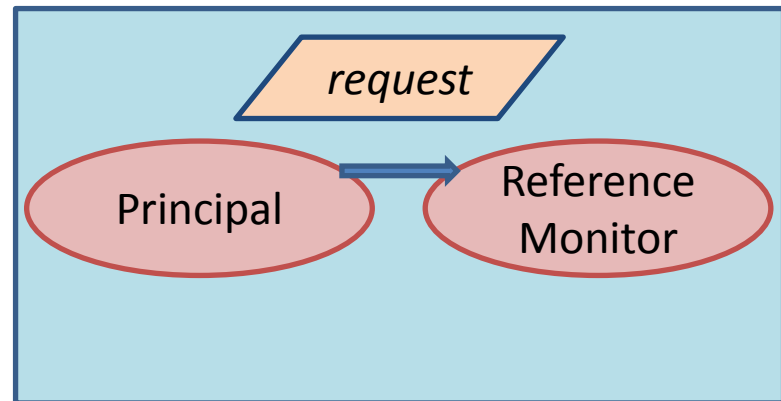
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



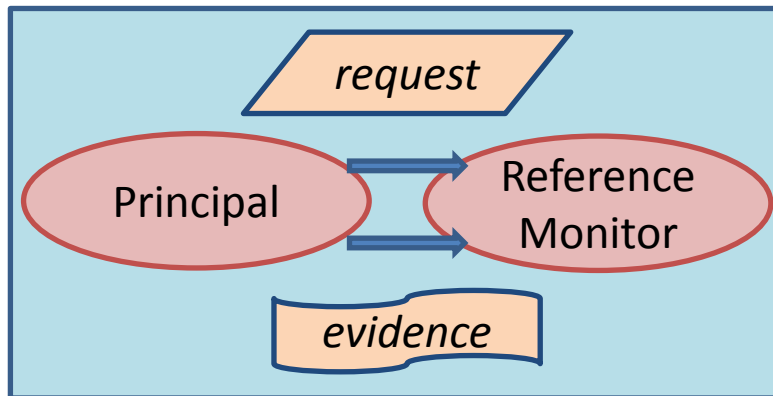
pull: Reference monitors gather evidence, with help from others.



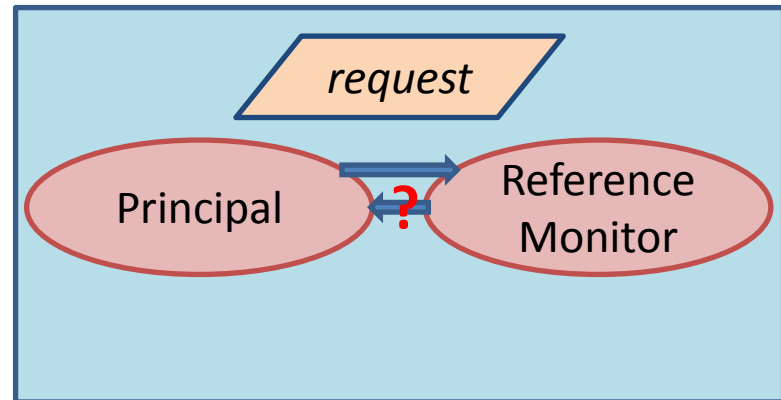
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



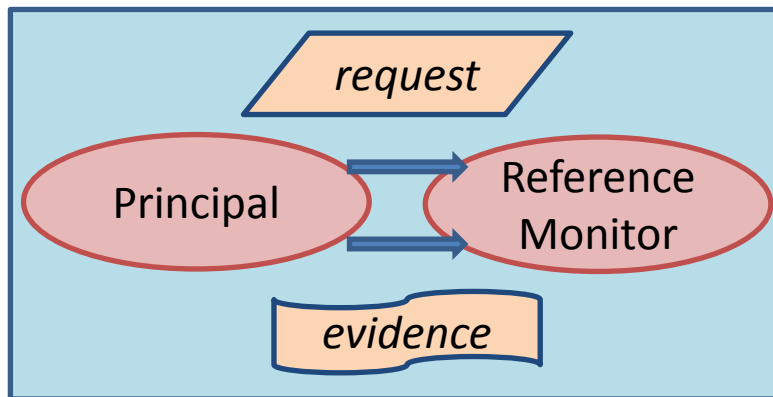
pull: Reference monitors gather evidence, with help from others.



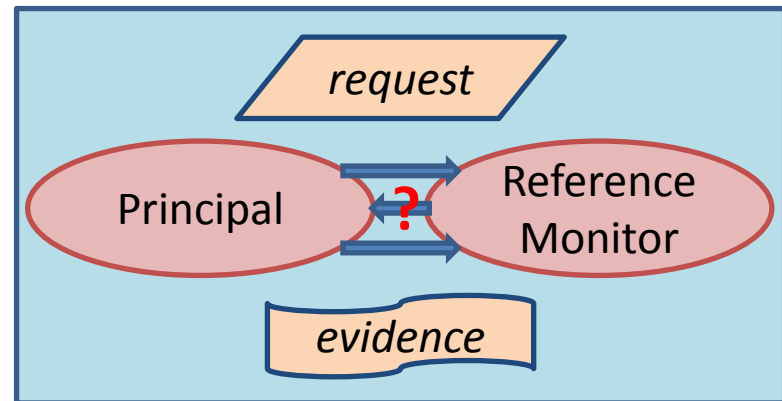
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



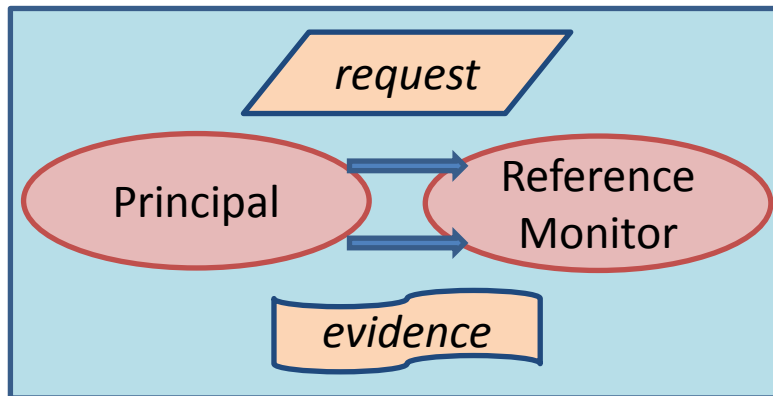
pull: Reference monitors gather evidence, with help from others.



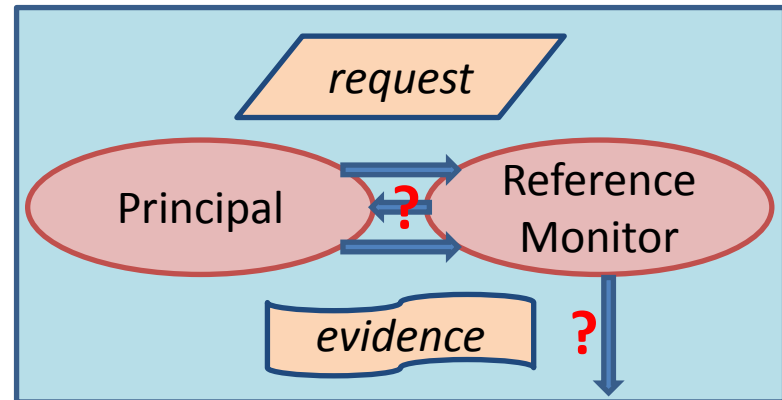
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



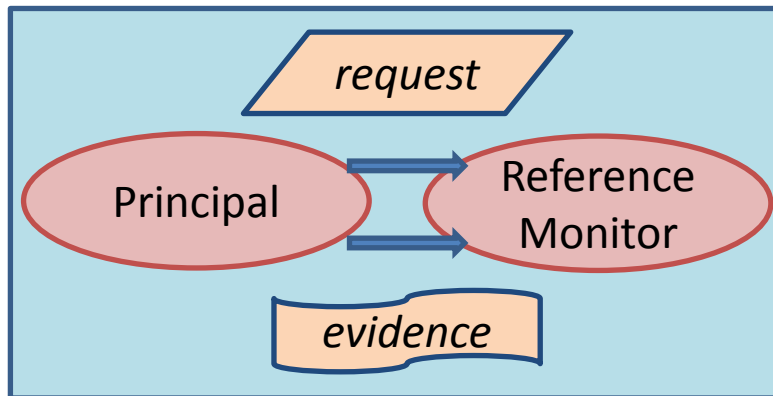
pull: Reference monitors gather evidence, with help from others.



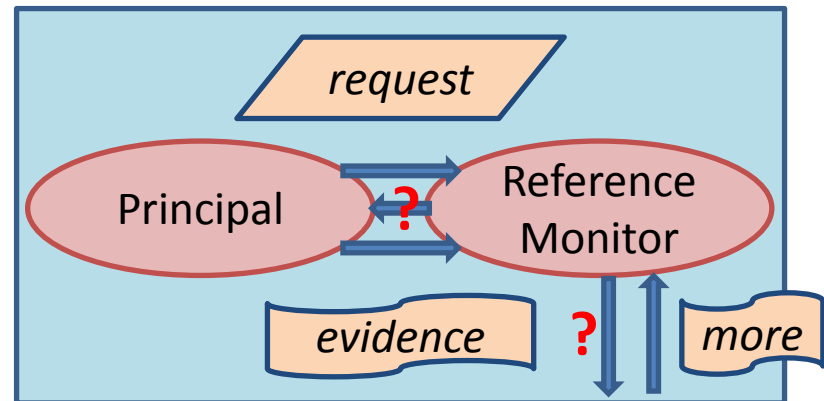
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



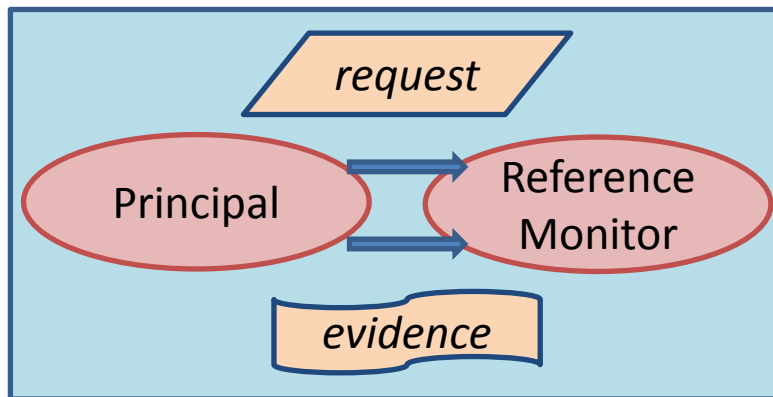
pull: Reference monitors gather evidence, with help from others.



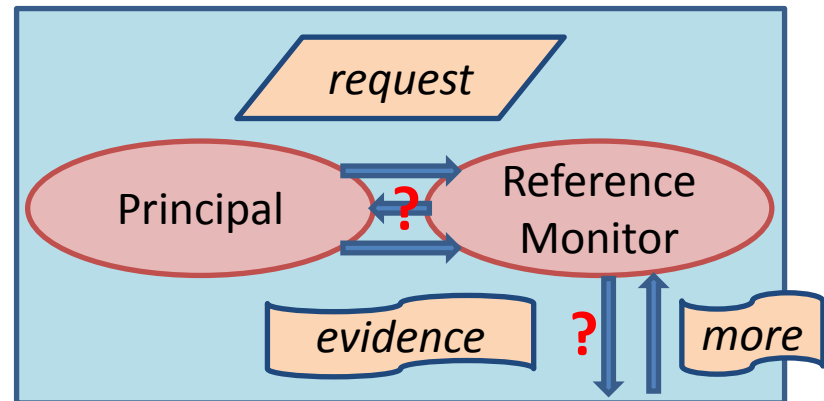
push vs. pull

- The reference monitor relies on proofs of identity, the access policy, and other evidence.
- It can gather this evidence by two methods:

push: Principals present evidence with requests



pull: Reference monitors gather evidence, with help from others.



- Concerns: completeness, efficiency, privacy.

*Embellishments and
complications*

Principals

Principals may be

- users,
- programs,
- computers,
- origins (in browsers),
- their combinations,
- ...

On principals

The notion of principal varies (dangerously) across systems and abstraction layers.

For example, one should not confuse

- IP addresses (e.g., 118.214.218.135),
- domains (e.g., whitehouse.gov),
- the computers at those addresses,
- the people who control the computers.



Some further elaborations

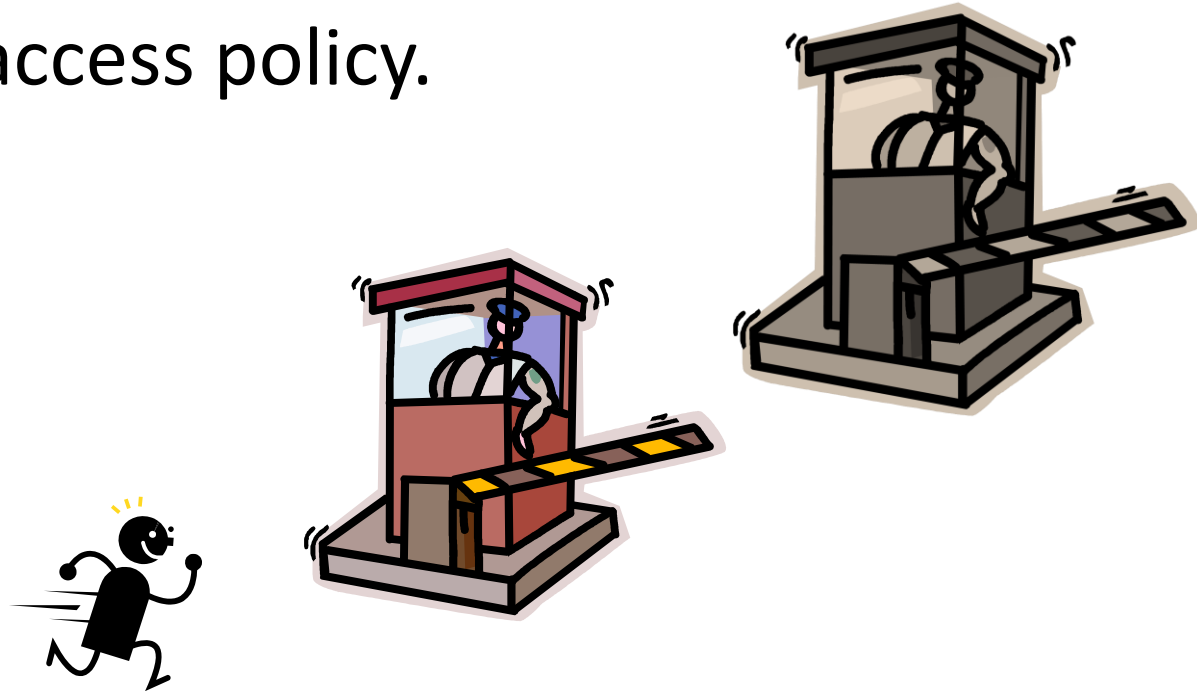
- Joint requests
- Groups
- Roles
- Negation
- Delegation
- Running programs

Conjunctions

- Sometimes a request should be granted only if it is made jointly by several principals.

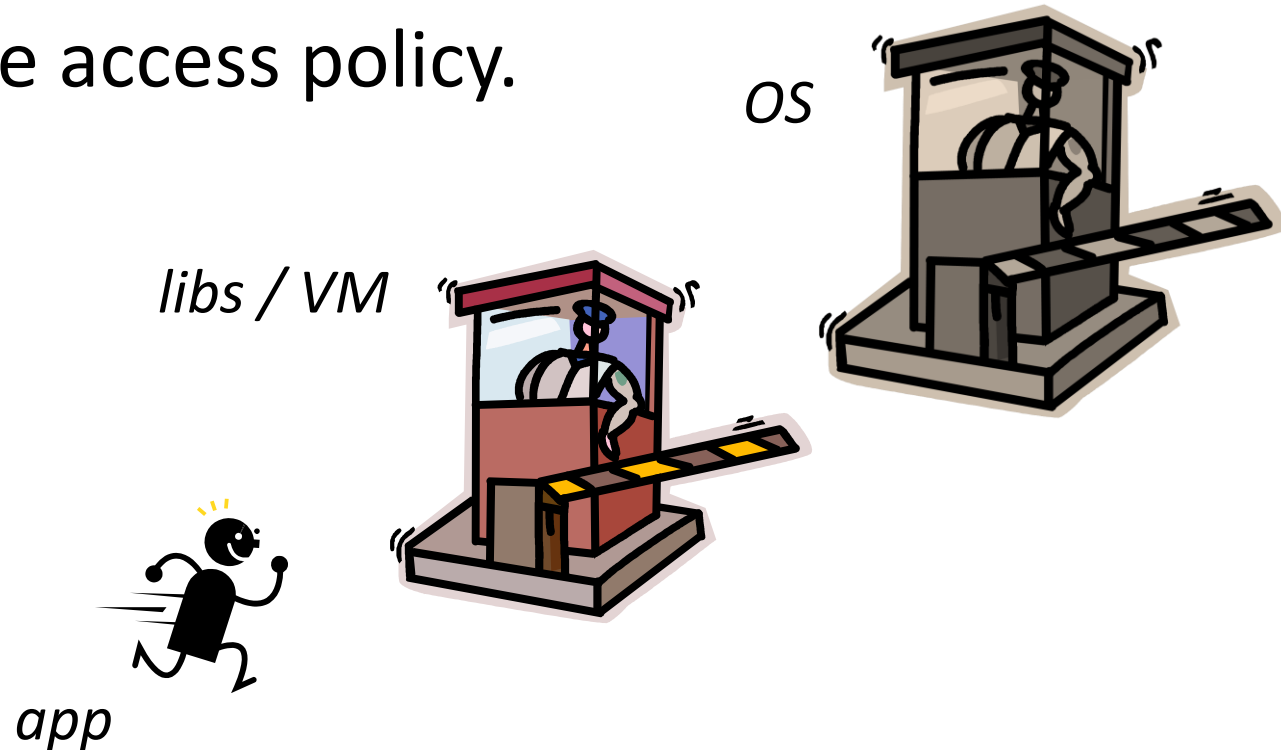
Conjunctions

- Sometimes a request should be granted only if it is made jointly by several principals.
- A conjunction may or may not be made explicit in the access policy.



Conjunctions

- Sometimes a request should be granted only if it is made jointly by several principals.
- A conjunction may or may not be made explicit in the access policy.



Groups and roles

- Principals can be organized into groups.
- Principals can play roles.
- These groups and roles may be used as a level of indirection in access control.
 - E.g., any member of a group G may access a file f .
 - E.g., anyone who can adopt the role R may then access a file f .



Groups and roles (cont.)

- Suppose that any member of group G may access file f owned by Alice.
 - G may be maintained by someone else.
 - G may change over time, without immediate knowledge of Alice.
 - f's ACL should be short and clear.
 - Proofs of memberships resemble (are?) capabilities.
 - Access to f may be partly anonymous.
 - Still, Alice may require a proof of identity at each f access, for auditing.

ACL for f (owned by Alice)
G

Members of G (owned by admin)
Alice
Bob
Charlie

On objects

Similarly, objects may include

- disk blocks,
- files,
- database tables, rows, and columns,
- application-level records, like calendar entries.

Picking objects is also an important part of designing an access control system.



On operations

Similarly, too, there are important choices in defining operations.

In particular, sometimes “small” operations should be bundled to form “bigger” ones.

- E.g.,
 - read a patient's record,
 - write a log record (for auditing).
- A principal may be allowed to do a “big” operation but not each of its components.



More on objects and operations

- Objects and operations may also be put in groups, e.g.,
 - all company files,
 - all write operations (e.g., append) on an object.
- Moreover, some policy may be automatically inherited from object to object.

Advanced Security Settings for try.txt



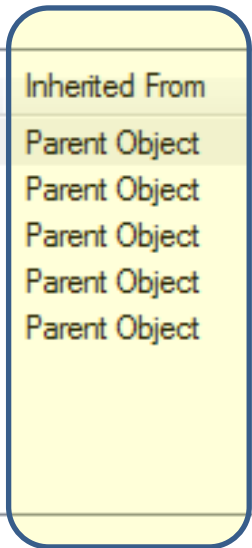
Permissions Auditing Owner Effective Permissions

To view details of a permission entry, double-click the entry. To modify permissions, click Change Permissions.

Object name: Z:\college\raw-material\try.txt

Permission entries:

Type	Name	Permission	Inherited From
Allow	AT Research Backup	Modify	Parent Object
Allow	Martin Abadi (abadi@microsoft.com)	Full control	Parent Object
Allow	Everyone	Read & execute	Parent Object
Allow	MSRSV-ServerAdmin (NORTHAMERI...	Take ownership	Parent Object
Allow	MsrTech (REDMOND\MsrTech)	Modify	Parent Object



Change Permissions...

Include inheritable permissions from this object's parent

[Managing permission entries](#)

OK Cancel Apply

Design choices

- Principals, objects, and operations should have the “right” granularity and be at the “right” level of abstraction
 - for ease of understanding,
 - to avoid giving away too much privilege.

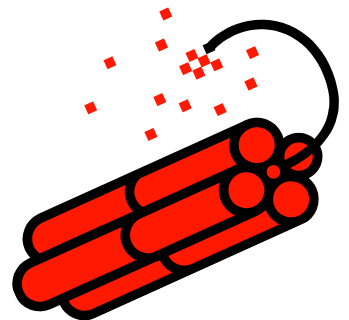
The principle of least privilege

[Saltzer and Schroeder, 1975]

Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Common dangers

- Access control can be insufficient or irrelevant
 - when it is implemented incorrectly,
 - when the underlying operations are implemented incorrectly,
 - when the policy is wrong,
 - when it is circumvented.

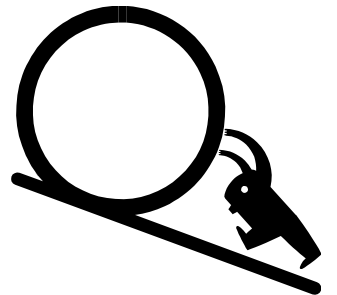


Further issues

- Many characteristics of distributed systems make access control harder:
 - size,
 - faultiness (e.g., revocations may get lost),
 - heterogeneity (e.g., of communication channels and of protection mechanisms),
 - autonomy, lack of central administration and therefore of central trust,
 - ...
- Access control seems difficult to get right.

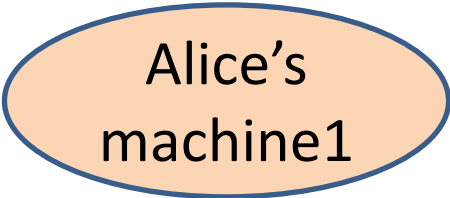
The *snowball effect*

- An illustration of the consequences of bad policies (particularly in distributed systems).
- Not a new problem, but still a problem.
- With a recent precise formulation and some research [Dunagan, Zheng, and Simon].



The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



Alice's
machine1

The snowball effect

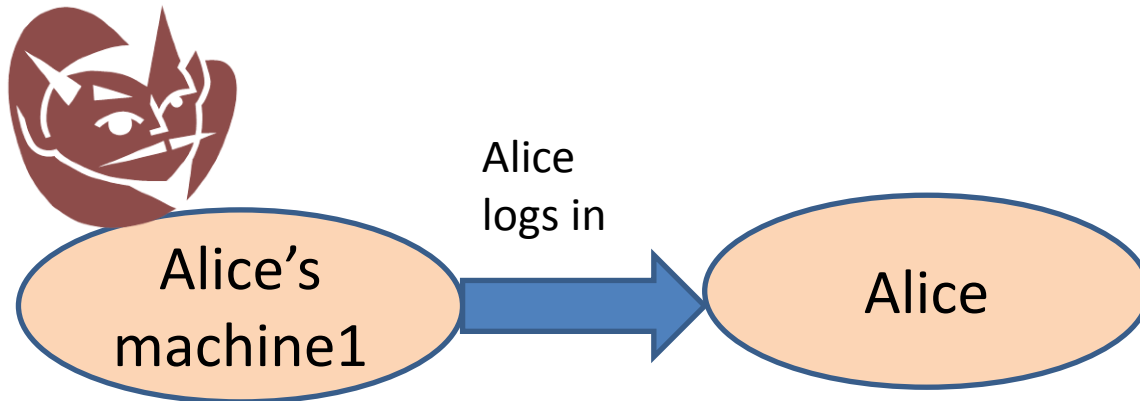
An illustration of the consequences of bad policies (particularly in distributed systems):



Alice's
machine1

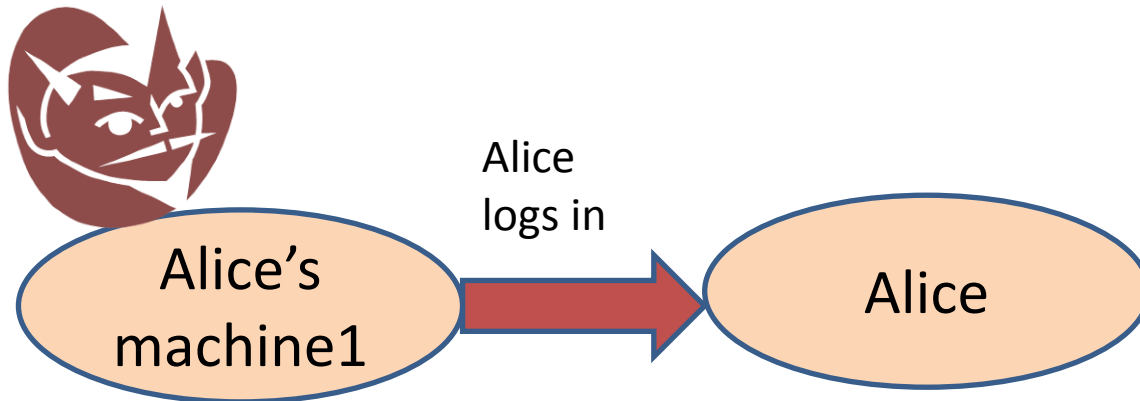
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



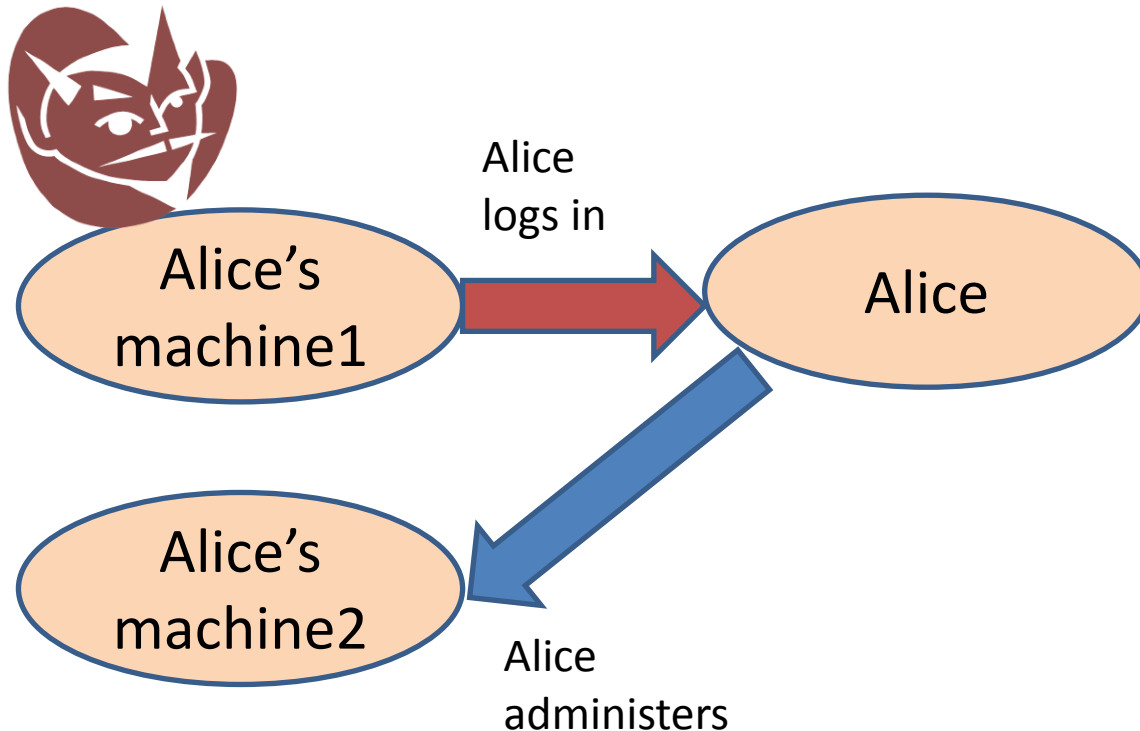
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



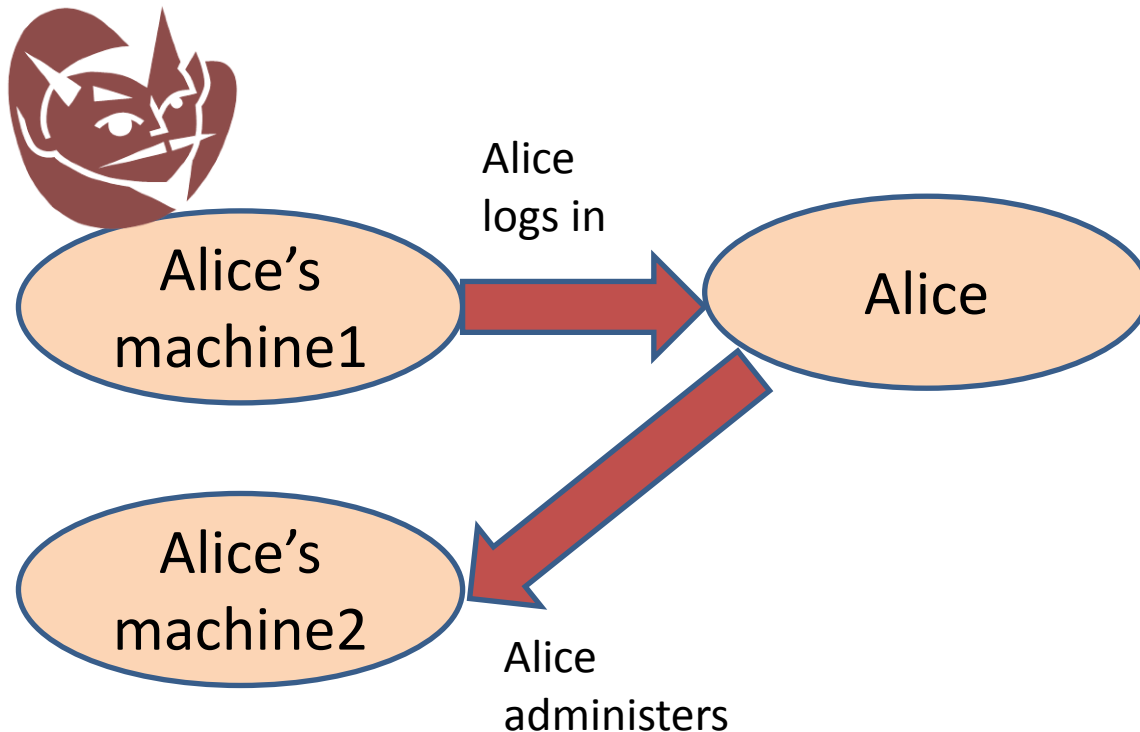
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



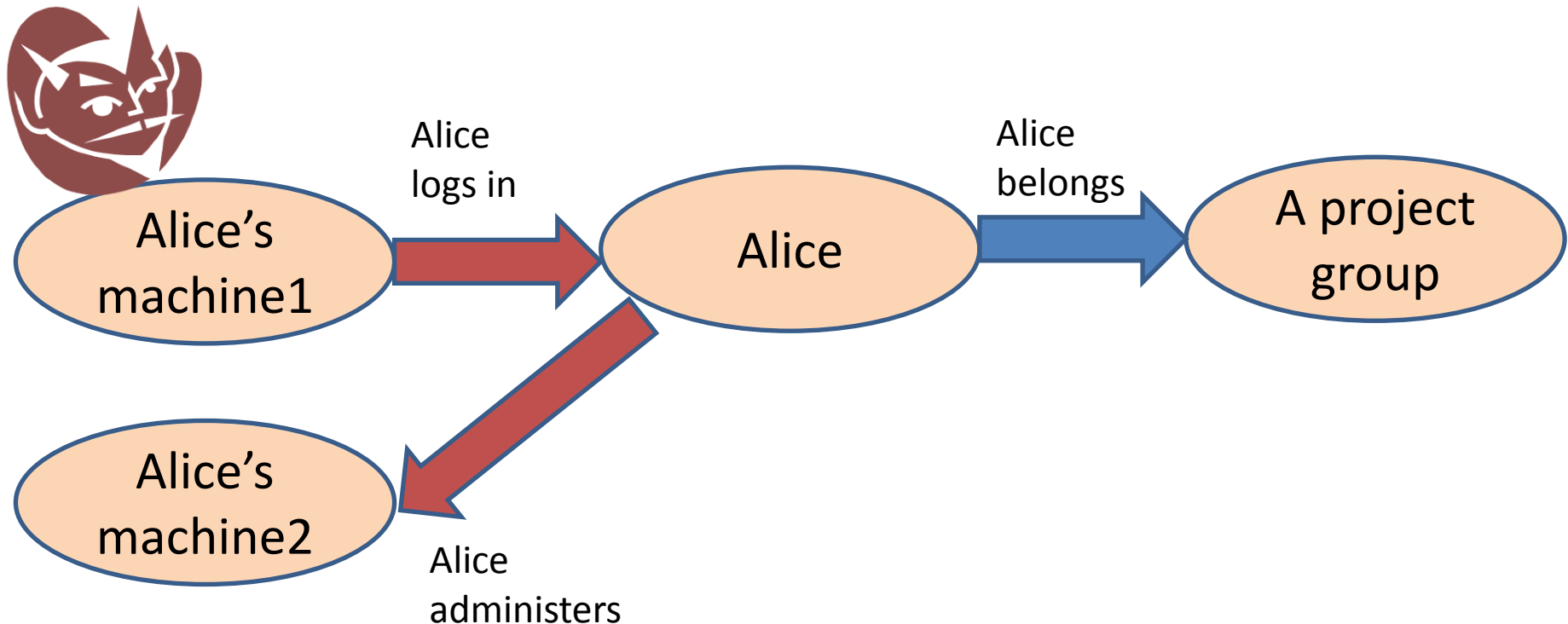
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



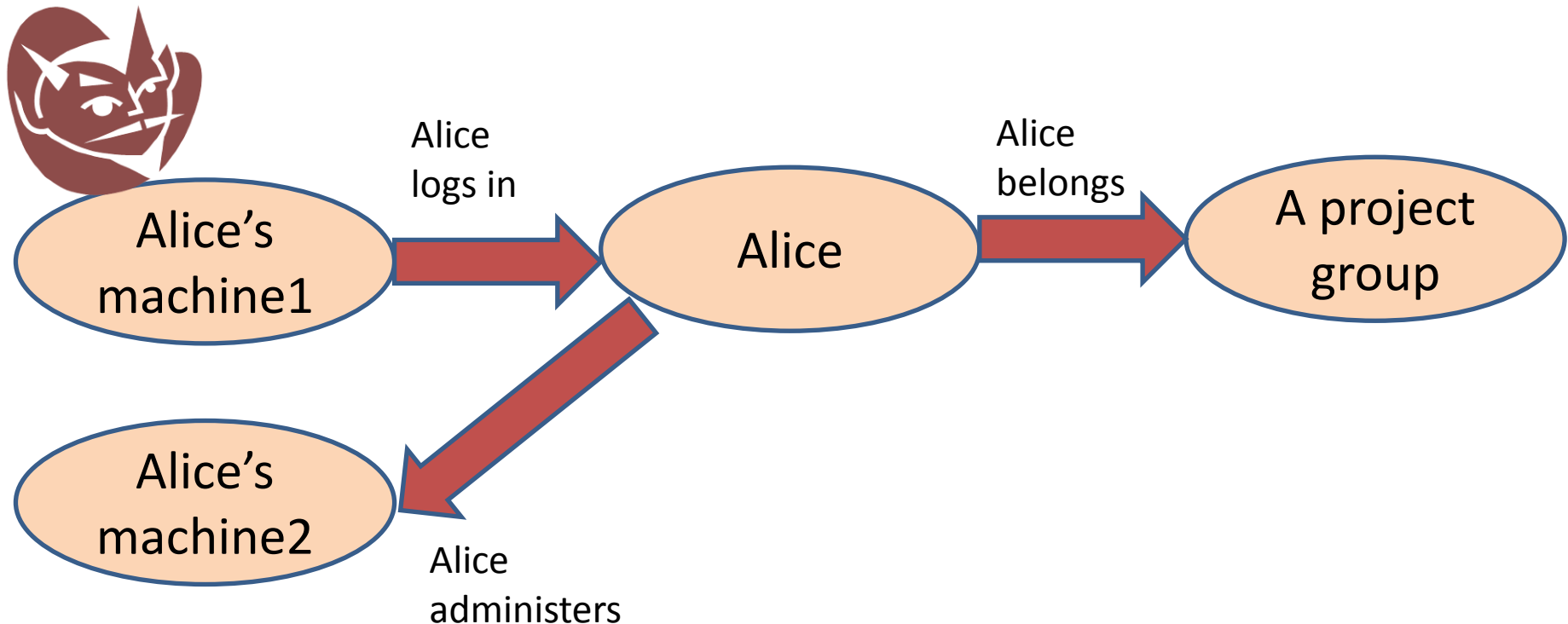
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



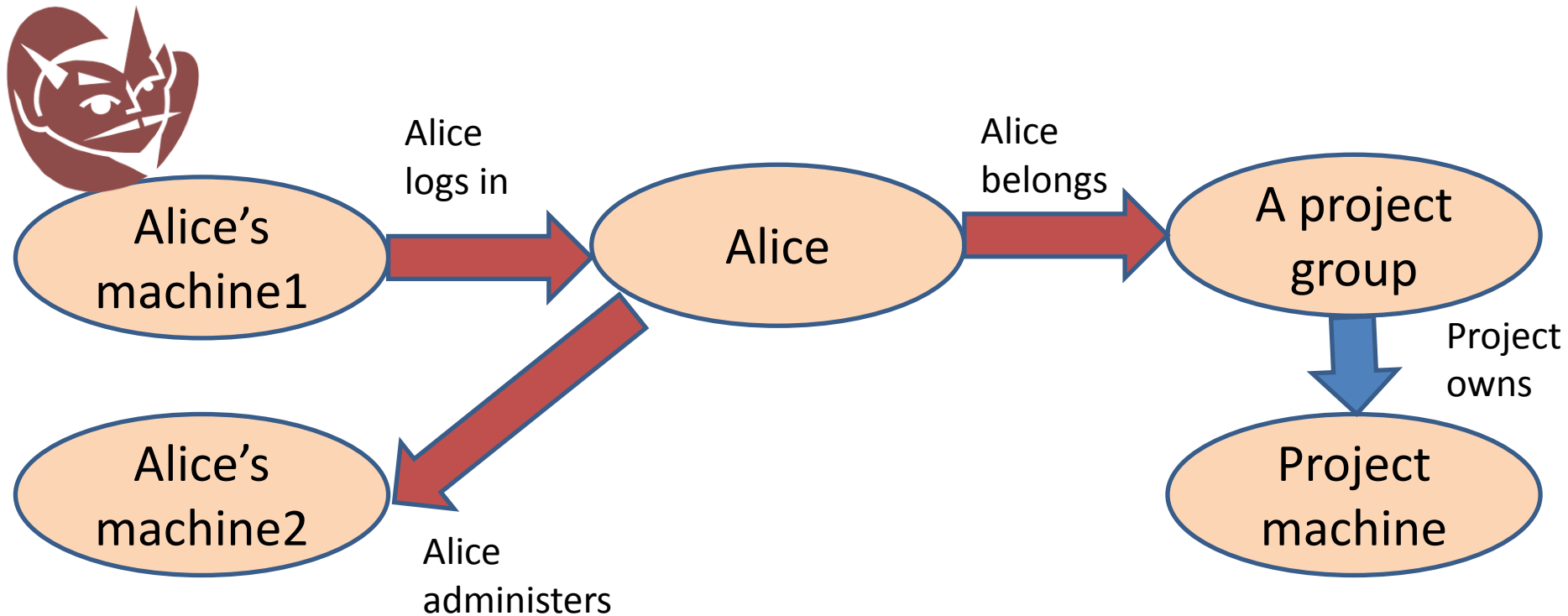
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



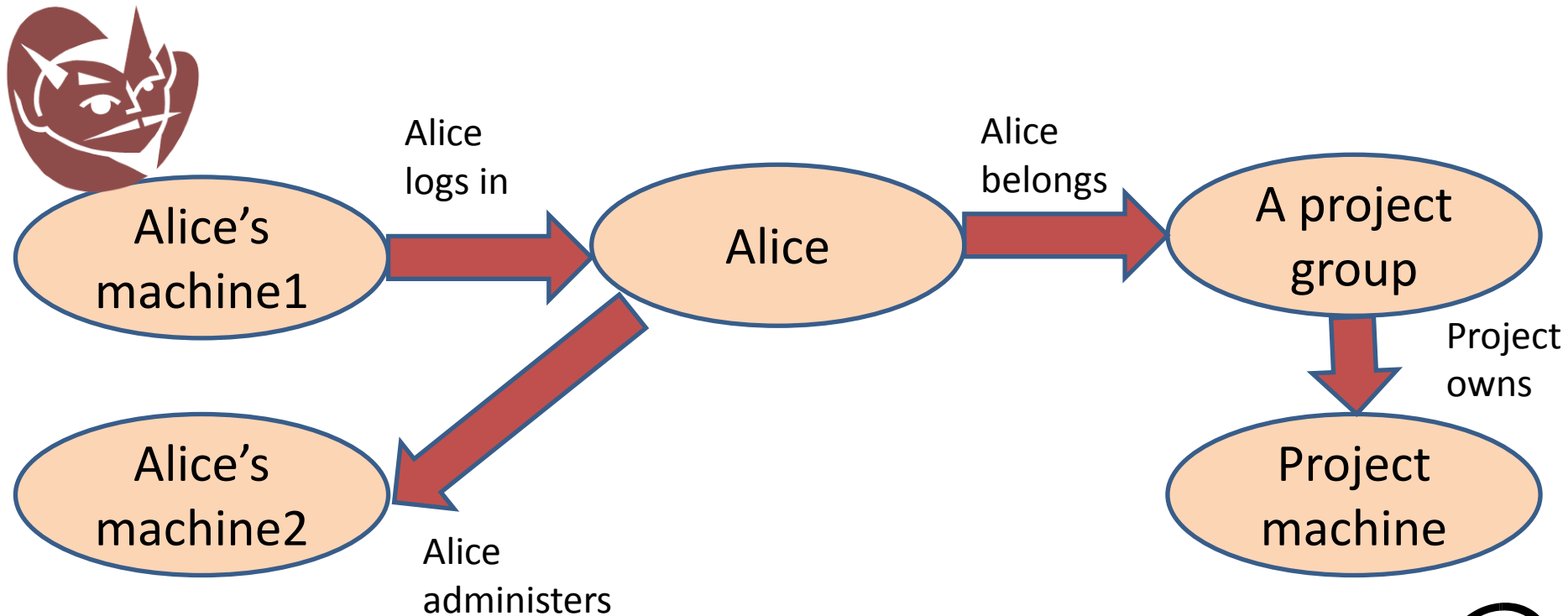
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



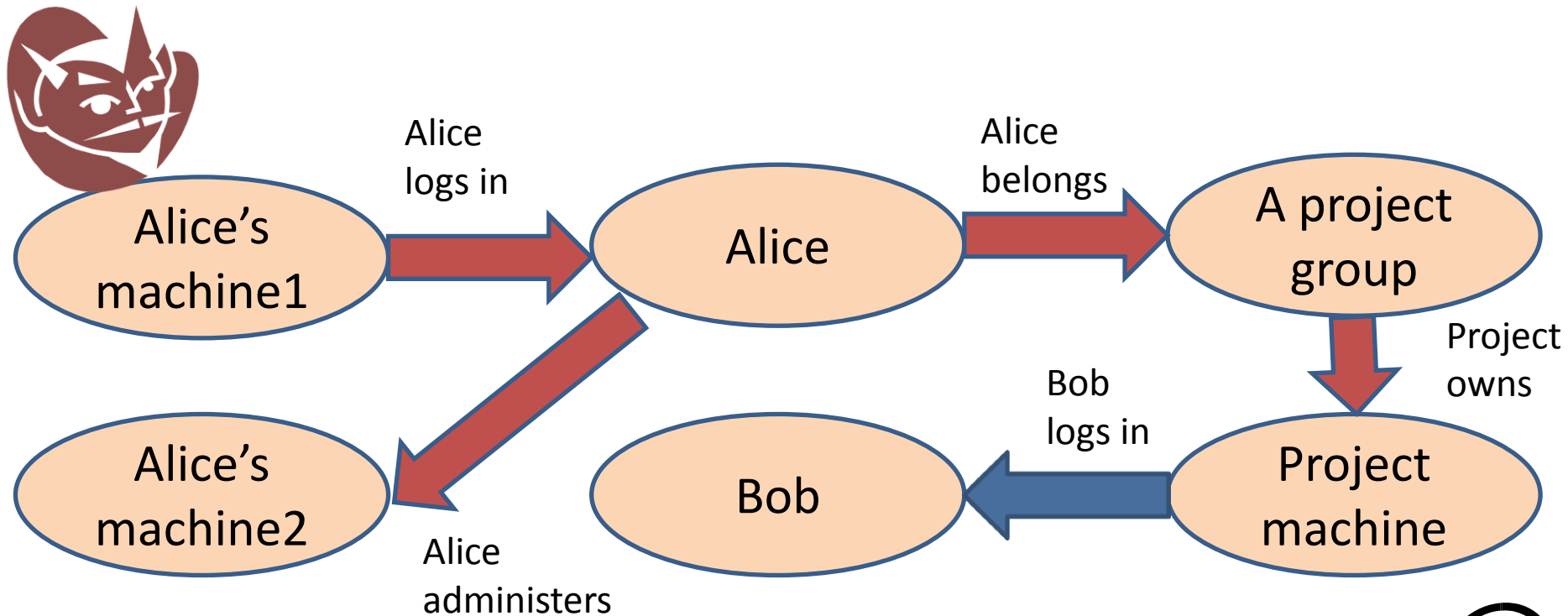
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):



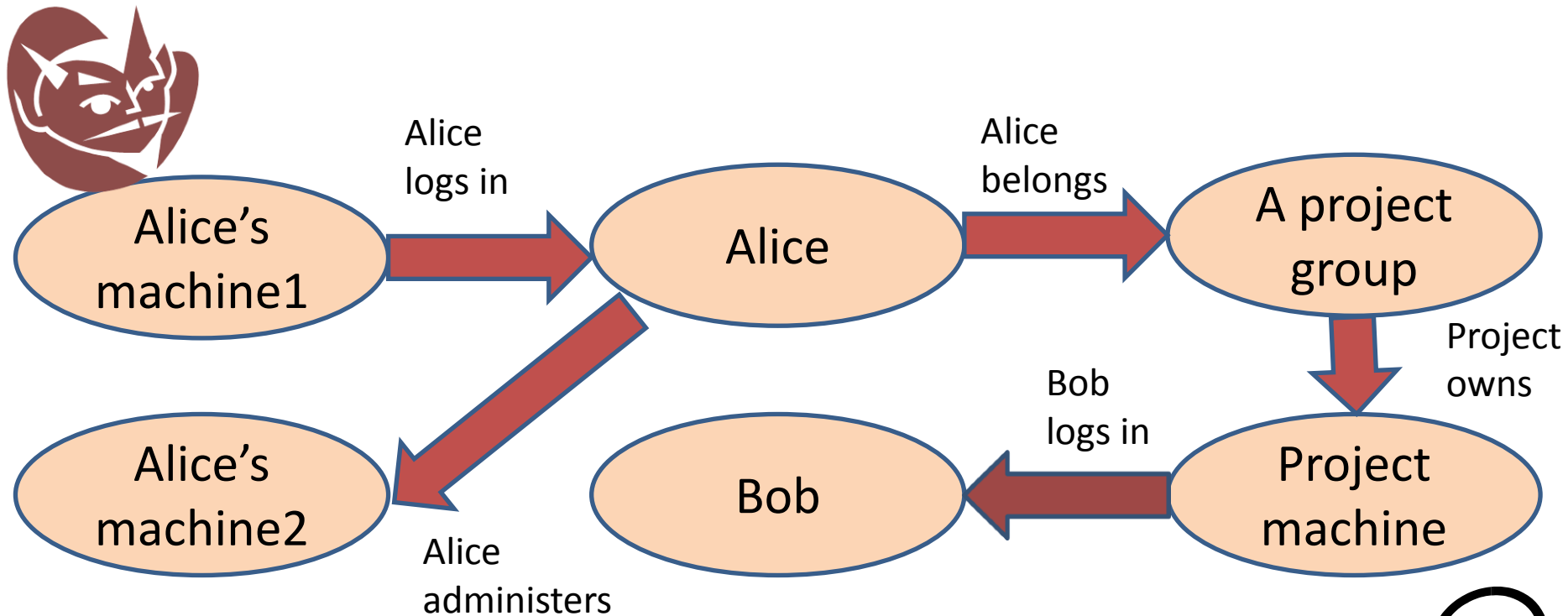
The snowball effect

An illustration of the consequences of bad policies (particularly in distributed systems):




The snowball effect

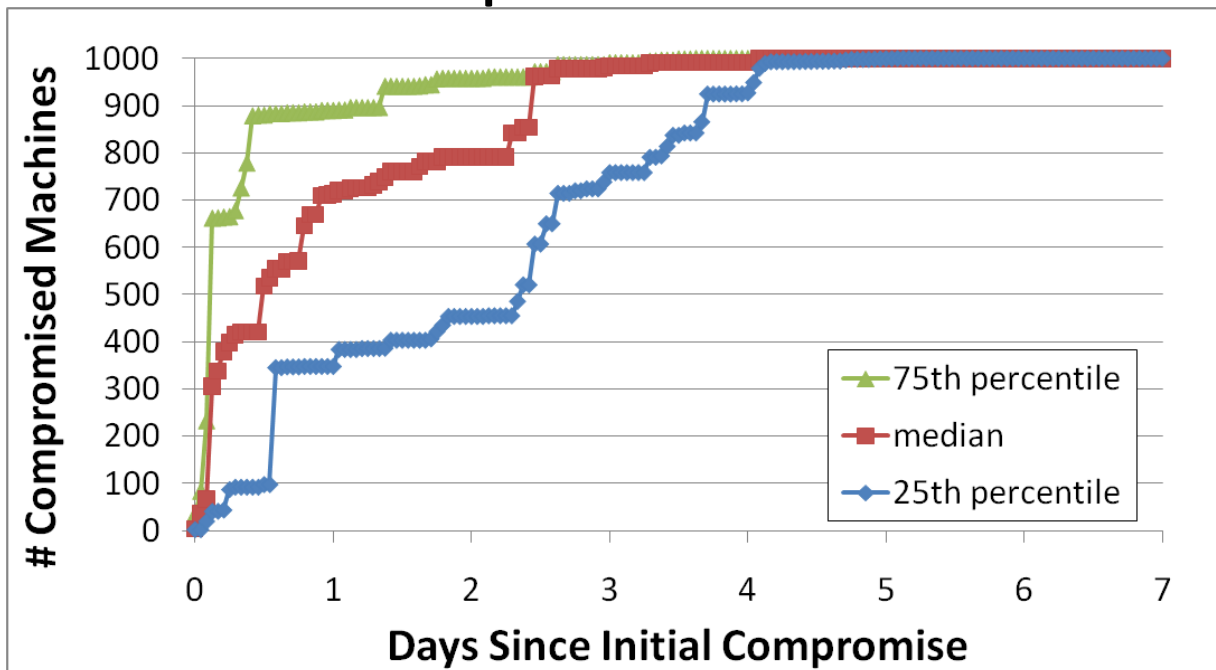
An illustration of the consequences of bad policies (particularly in distributed systems):



Snowball experiment

[Dunagan, Zheng, and Simon]

- Over 1 week, observe “log in”, “administer”, and “member” relations in a system. 
- Then compute the effects of a single random initial compromise.

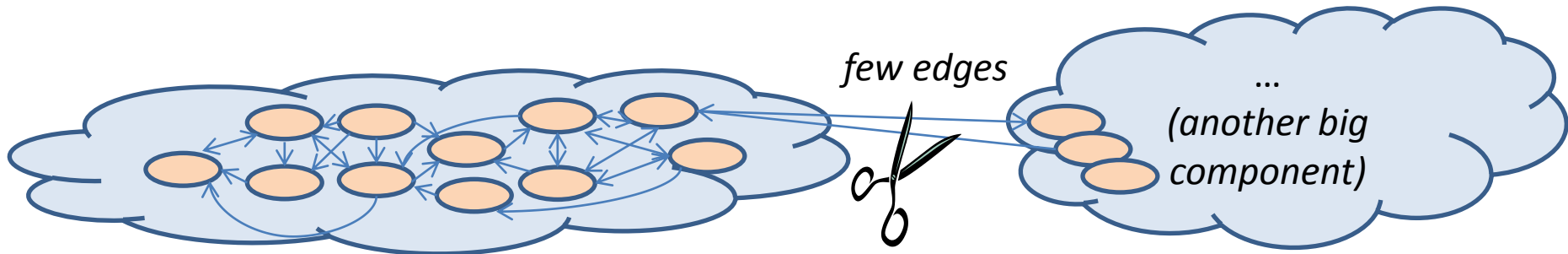


Cutoff at 1,000 for confidentiality reasons.

In an organization with ~100K accounts and ~200K machines.

Defenses

- Having analyzed the relations in a system, one may try to remove some of them.
 - The functioning of the system requires many of these relations!
 - Dunagan et al. find good candidates in *sparse cuts*.

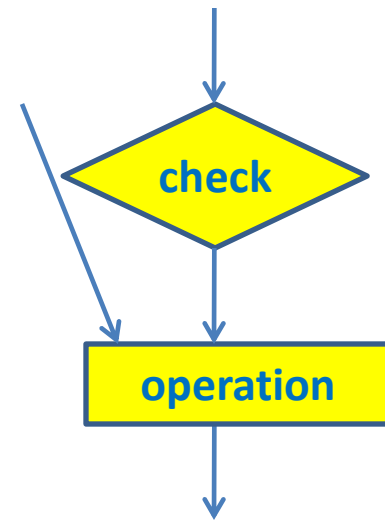


- We can also use stronger building blocks.
 - E.g., making it harder for a compromised machine to impersonate its users.

Circumventing access control

Sometimes the reference monitor does not protect all important objects and operations, for example because of

- hostile platforms (e.g., for DRM systems),
- control-flow subversions (as we will see),
- race conditions,
- data recovery from memory or disks,
- side channels.

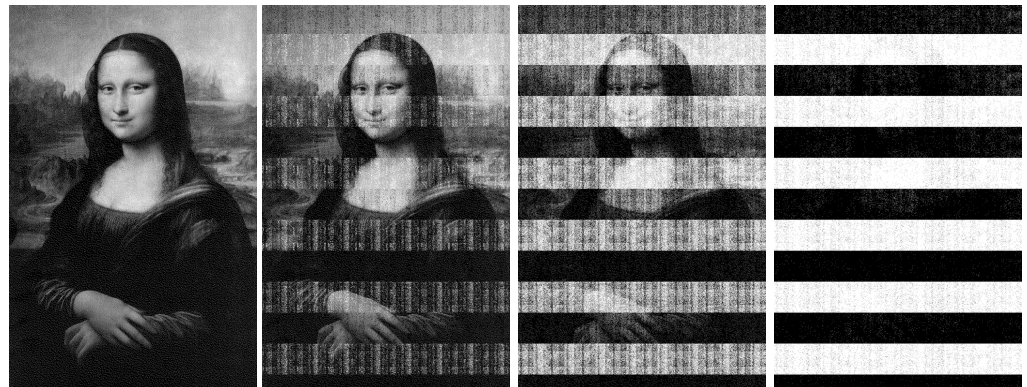


Data recovery from memory

- Memory does not lose data as soon as it is disconnected!
- An attacker must be able to access the memory physically, find secrets in it, and do some error correction.



*Cold
RAM
chips
(-50°C).*



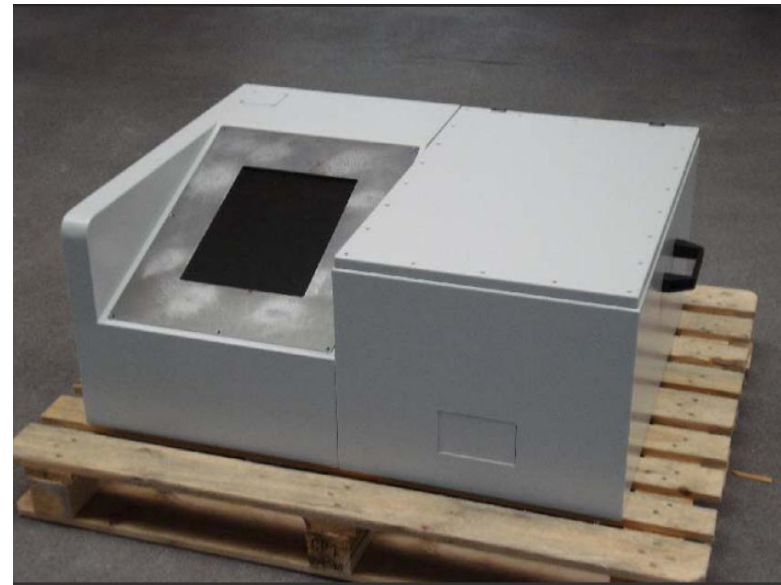
5 secs. 30 secs. 60 secs. 5 mins.

Source: J. A. Halderman et al.

<http://citp.princeton.edu/memory/media/>

“Tempest” in Dutch voting (2006)

- A character in the name of a party caused some voting-machine displays to switch refresh frequencies.
- *The resulting radio emissions were different!*
- This could let someone outside a voting booth identify the party’s name.



Source: B. Jacobs and W. Pieters

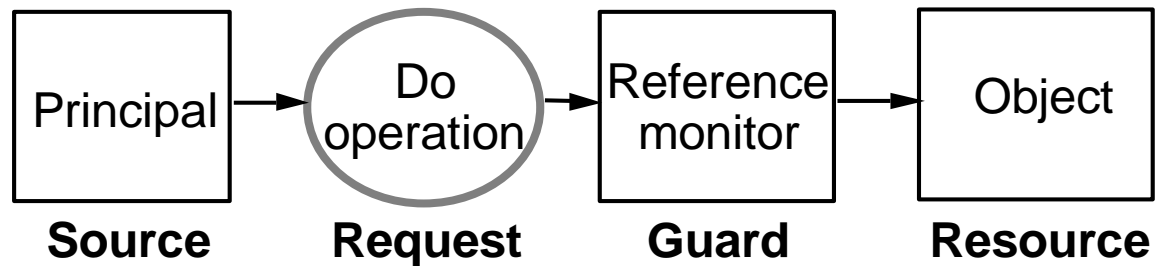
A prototype tempest-shielded vote-printer, with touch screen and protected tray for the printed vote; almost 100kg.

Reading

- Lampson's "Protection"
<http://research.microsoft.com/en-us/um/people/blampson/08-Protection/Acrobat.pdf>
- Saltzer & Schroeder's "The protection of information in computer systems"
<http://web.mit.edu/Saltzer/www/publications/protection/index.html>
- Lampson's "Computer Security in the Real World"
<http://research.microsoft.com/en-us/um/people/blampson/64-SecurityInRealWorld/Acrobat.pdf>
- Dunagan et al.'s "Heat-ray: Combating Identity Snowball Attacks Using Machine Learning, Combinatorial Optimization and Attack Graphs"
<http://research.microsoft.com/en-us/um/people/jdunagan/sosp112-dunagan.pdf>
- Jacobs & Pieters's "Electronic Voting in the Netherlands: from early Adoption to early Abolishment"
<http://www.cs.ru.nl/B.Jacobs/PAPERS/E-votingHistory.pdf>
- Halderman et al.'s "Lest We Remember: Cold Boot Attacks on Encryption Keys"
<https://citp.princeton.edu/research/memory/>

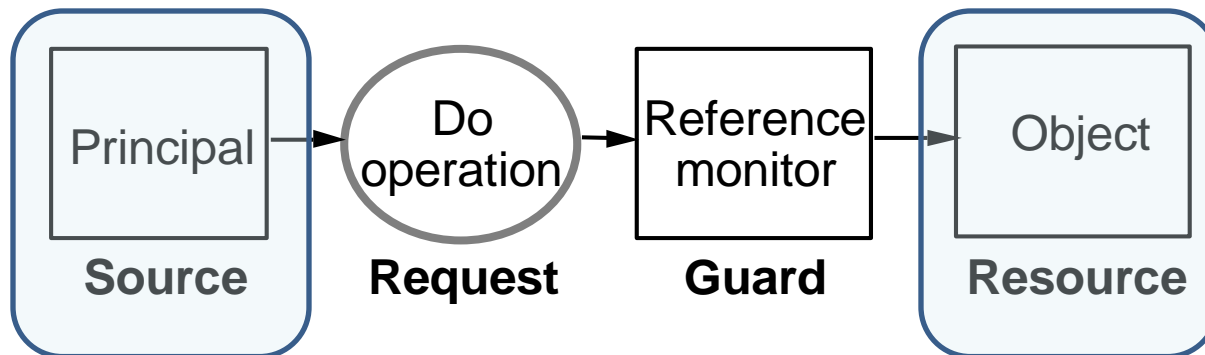
Access control and programs

Programs everywhere!



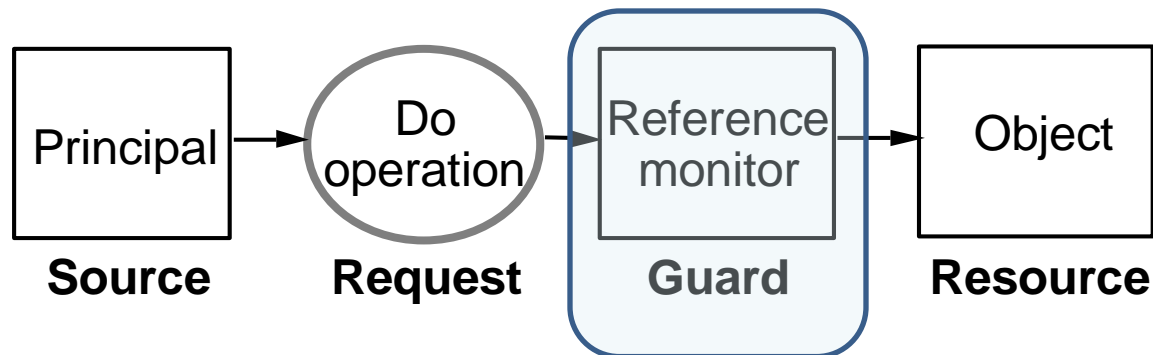
Programs everywhere!

- Programs are principals and objects.



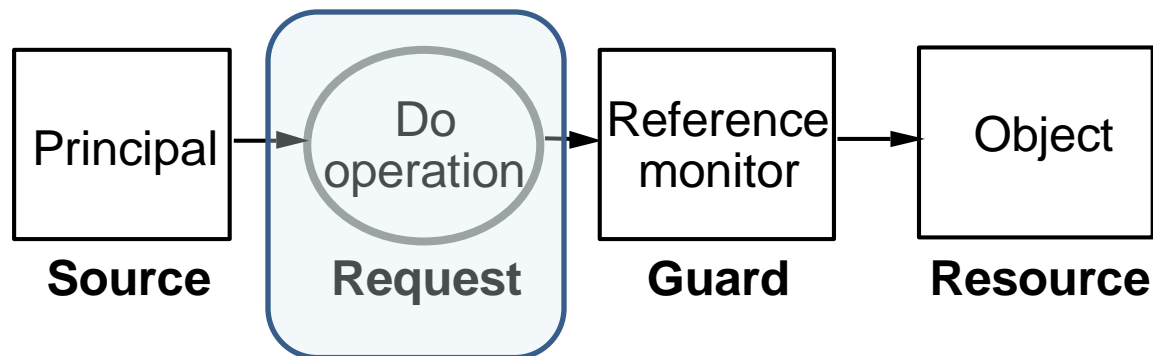
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
 - Often, even some of the access control policy is baked into programs, for better or for worse.



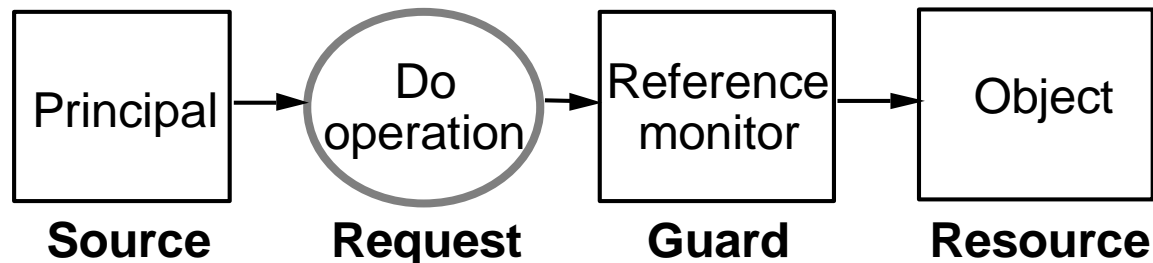
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
 - Often, even some of the access control policy is baked into programs, for better or for worse.
- Programs implement the operations that are the concern of access control.



Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
 - Often, even some of the access control policy is baked into programs, for better or for worse.
- Programs implement the operations that are the concern of access control.



Bundling operations into programs

<i>objects</i> <i>principals</i>	Data file	Log file	Program P's code
Alice			x
auditor		r	r
Program P	rw	rw	x

Conjoining users through programs

<i>objects</i> <i>principals</i>	Data file	Program P's code
Alice		x
Bob		x
Program P	rw	x

where P checks that both Alice and Bob make the same request before forwarding the request.

Conjoining users through programs

(an alternative)

<i>objects</i> principals	Data file	Data file's ACL	Program P's code
Alice			X
Bob			X
Program P		<i>rw</i>	X

where P modifies the matrix so that Alice has access when Bob requests it, and vice versa.

Modifying the matrix

- The access control matrix need not be static.
- It may be modified by programs like:

```
command CONFER (right, user, friend, file)
  if right in matrix[user, file]
    then enter right into matrix[friend,file]
  end
```

How can we ensure safety?

Algorithmic analysis

[starting with Harrison, Ruzzo, and Ullman, 1976]

- A *system* has finite sets of rights and commands.
- A *command* is of the form
“if conditions hold, then perform operations”.
 - The conditions are predicates on the matrix.
 - Operations add/delete rights, principals, objects.

Let A be a principal and f an object.

In general, it is undecidable whether there is a reachable state such that A can access f .

Algorithmic analysis (cont.)

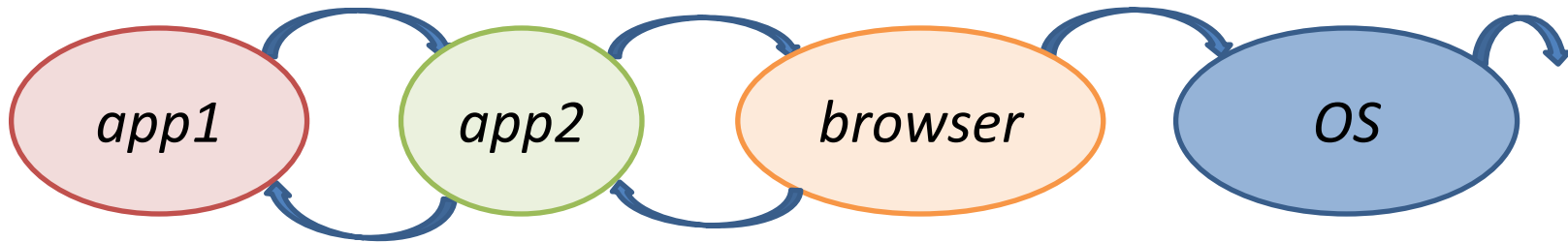
[in particular, Li, Winsborough, and Mitchell, 2003]

- Not all interesting problems are undecidable!
- Consider the containment problem:
In every reachable state, does every principal that has one property (e.g., has access to a resource) also have another property (e.g., being an employee)?

For different classes of systems, this problem is decidable (in coNP or coNEXP).

Programs and other principals

- So, programs may be principals too.
- But then:
 - we need to deal with program combinations,



- we need to connect programs to other principals
 - who write them or edit them,
 - who provide them or install them,
 - who call them.

Running programs

- What are the run-time rights of a program P?
 - those of P's caller, or
 - those of some responsible user, or
 - something else, e.g, because of P's properties, or
 - some combination.
- The same factors appear in deciding whether to run a program.



Running programs (cont.)

Some approaches to combining authorities:

- setuid,
- code access security (with stack inspection or alternatives).

Some approaches to intrinsic properties:

- proofs (and proof-carrying code),
- types,
- dynamic checks (e.g., in sandboxes),
- their combinations (e.g., proofs about sandboxes).

Protection and isolation

- Programs must be protected (always) and limited to communicate on proper interfaces.
- This is often the job of the computing platform (OS + hardware).
 - It can implement address spaces so that programs in separate spaces cannot interact directly (e.g., cannot smash or snoop on one another).
- A language and its run-time system can provide fine-grained control.

More on this later.

Examples

Access control in Unix (basics)

- Principals are users (plus root).
- Objects are files.
- Operations are read, write, and execute.
- Each file has an owner and a group.
- Each file has an ACL, which can be set by its owner and root.
- ACLs specify rights for the owner (“user”), group, and others (e.g., **rw-rw-r--**).

Access control in Unix (cont.)

- If a program file is marked as `suid`, then the program executes with the privilege of its owner (not that of the caller).
 - The usage of `setuid` is error-prone.
 - The details are complex and vary across systems.
- And there are other complications: `sgid`, capabilities in Linux, directories, ...

See “Setuid Demystified”, by Chen, Wagner, and Dean.

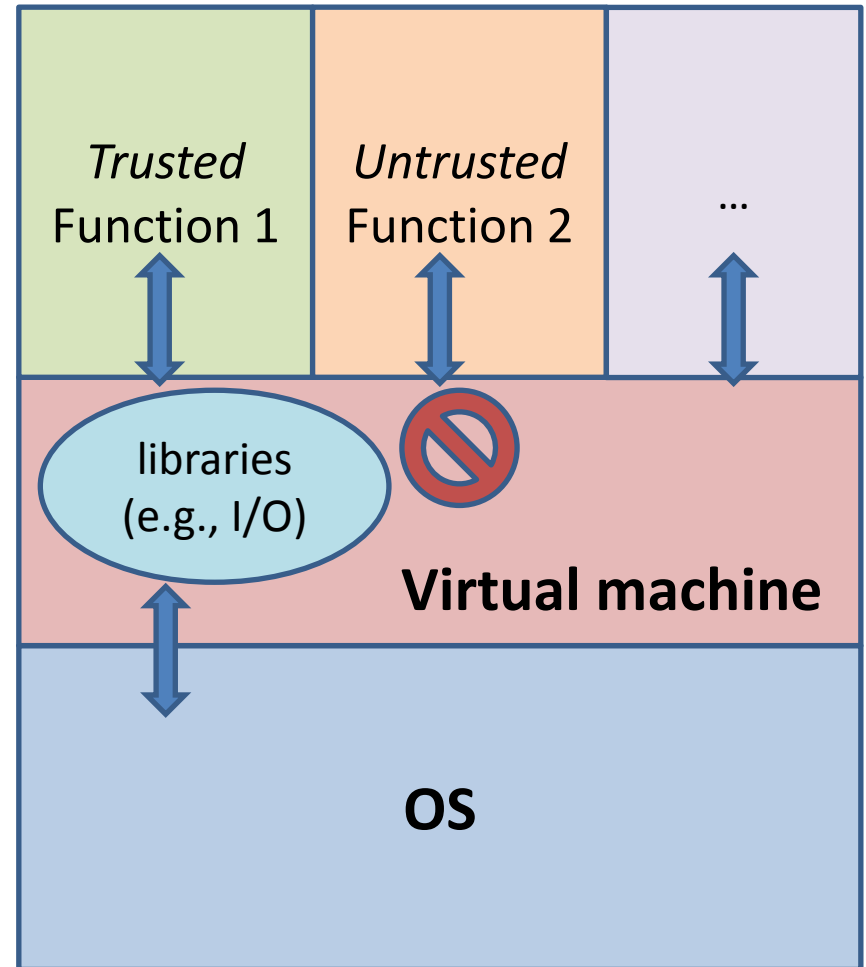
The basic sandbox policy

- *Trusted code* (e.g., local code) has the full power of the user that runs it.
- *Untrusted code* (e.g., foreign code) has very limited rights, e.g.:
 - no direct use of files,
 - network connections only to the code's origin.
- The sandbox is enforced at run-time:
 - A reference monitor (“security manager”) is associated with code when the code is loaded.

The basic sandbox policy

Trusted code can access libraries and thereby the underlying OS services.

Untrusted code mostly cannot.

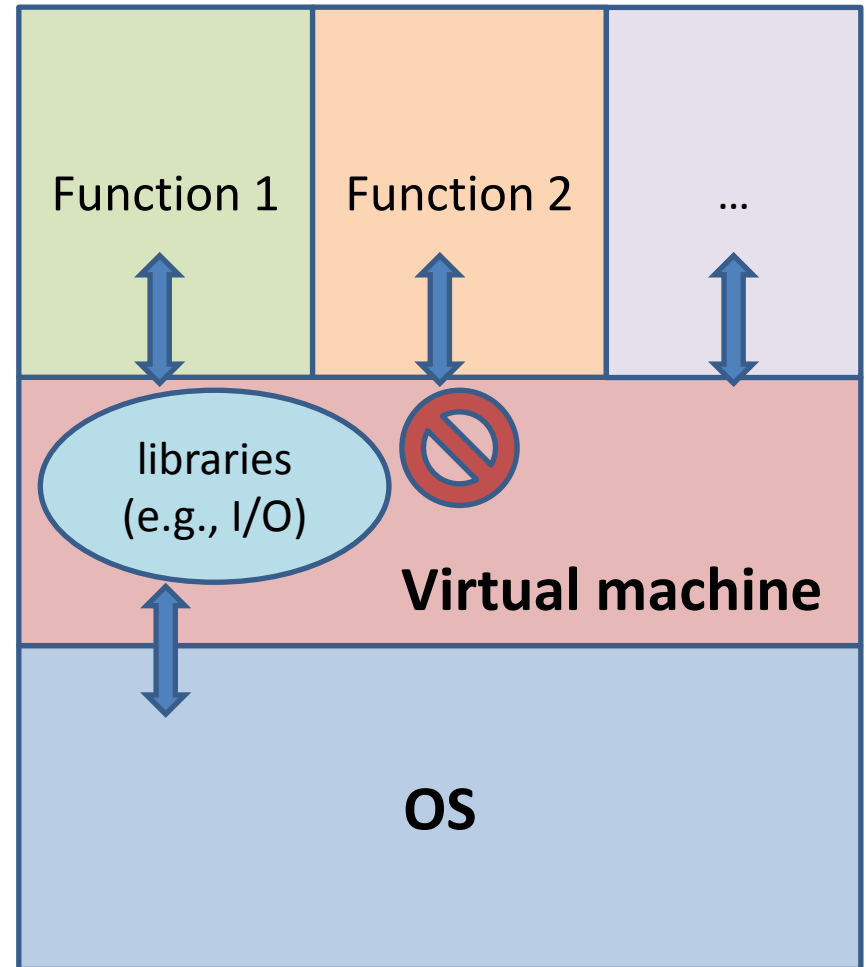


Permissions (as in Java)

Access to resources is expressed in terms of *permissions*, such as “may perform screen I/O”.

Before execution, an annotation on each piece of code (e.g., function) indicates its permissions.

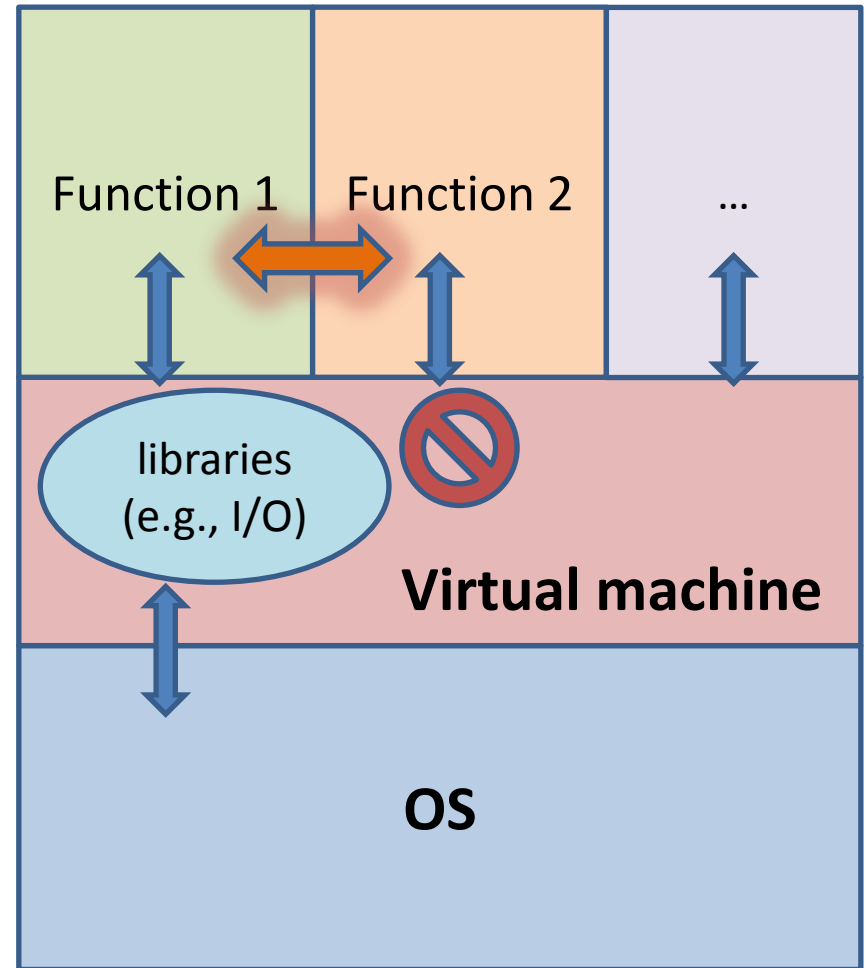
A *configurable policy* determines permissions depending on *code origin*.



Permissions (cont.)

Code with a variety of origins, more or less trusted, may call one another or share data.

Should all of their permissions count in access decisions?



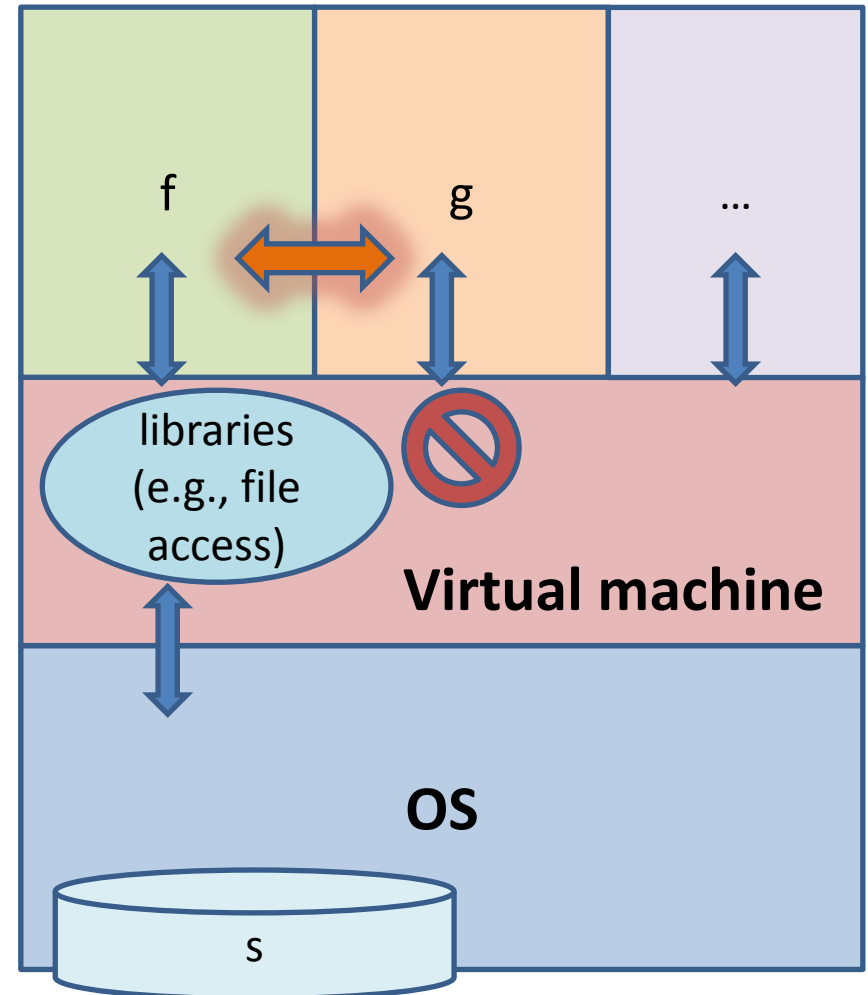
One answer, on a simple example

(also as in Java)

Suppose that $f(s)$ modifies the file named s .

If g calls $f(s)$, **both** should have permission to write to s .

(Otherwise, f may be used as a ***confused deputy***.)



An example where looking at the stack suffices

// **Fully trusted but naive**: has all permissions

```
public class NaiveApp {  
    public static void Write (string s, ... ) {  
        File.Write (s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
class BadApp {  
    public static void Main() {  
        NaiveApp.Write (“..\\password”, ...);  
    }  
}
```



BadApp

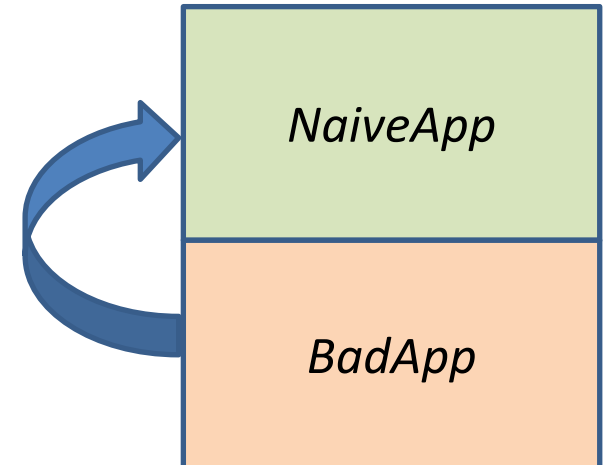
An example where looking at the stack suffices

// **Fully trusted but naive**: has all permissions

```
public class NaiveApp {  
    public static void Write (string s, ... ) {  
        File.Write (s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
class BadApp {  
    public static void Main() {  
        NaiveApp.Write (“..\\password”, ...);  
    }  
}
```



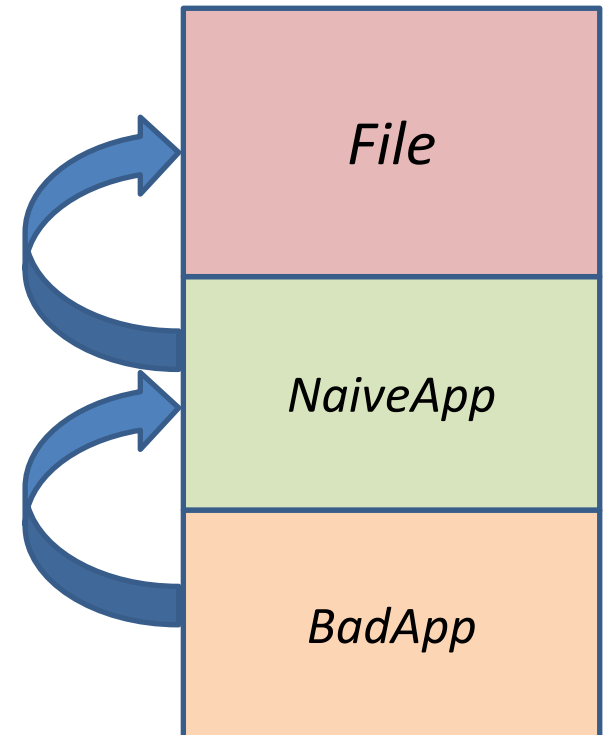
An example where looking at the stack suffices

// **Fully trusted but naive**: has all permissions

```
public class NaiveApp {  
    public static void Write (string s, ... ) {  
        File.Write (s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
class BadApp {  
    public static void Main() {  
        NaiveApp.Write (“..\\password”, ...);  
    }  
}
```

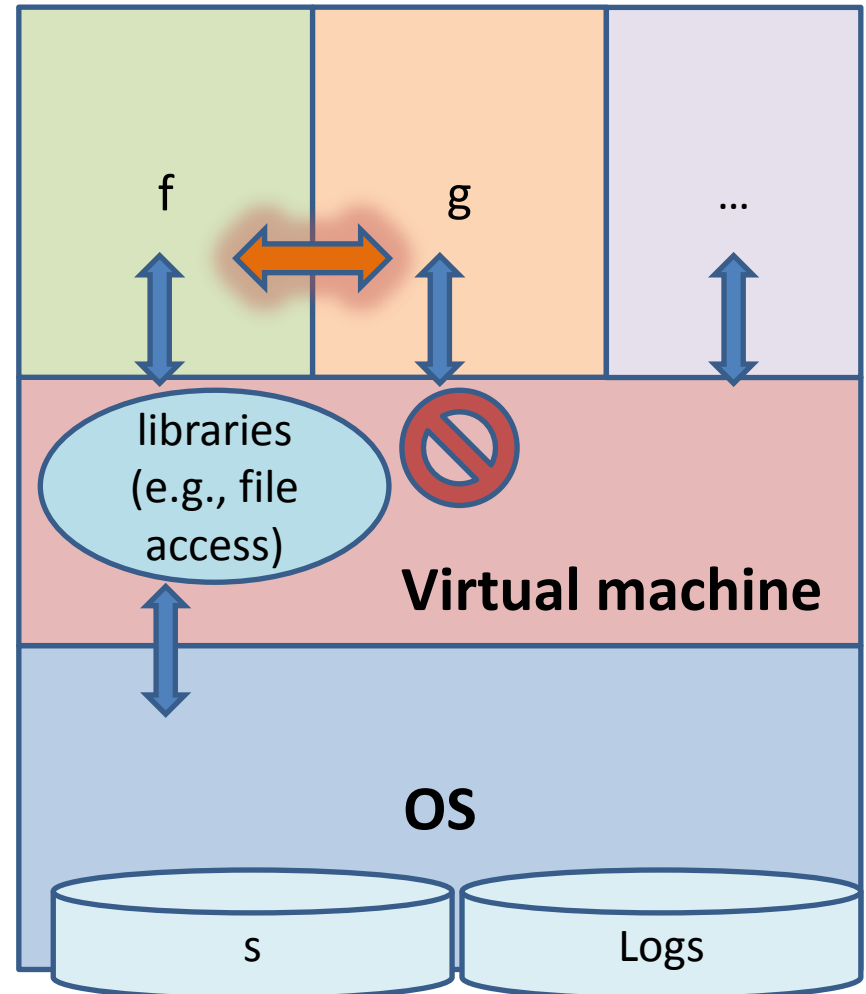


A twist

Suppose that $f(s)$ wants to write to a log that g should not access.

If f is a trusted function, it can check that g 's call is ok, assert it, and then use its own authority for writing to the log.

Afterwards, g 's permissions do not matter, only f 's.



An example where looking at the stack does not suffice

// **Fully trusted but naive**: has all permissions

```
class NaiveApp {  
    public static void Main() {  
        string s = BadPlugIn.TempFile ();  
        File.Write(s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
public class BadPlugIn {  
    public static string TempFile () {  
        return "..\\password";  
    }  
}
```



NaiveApp

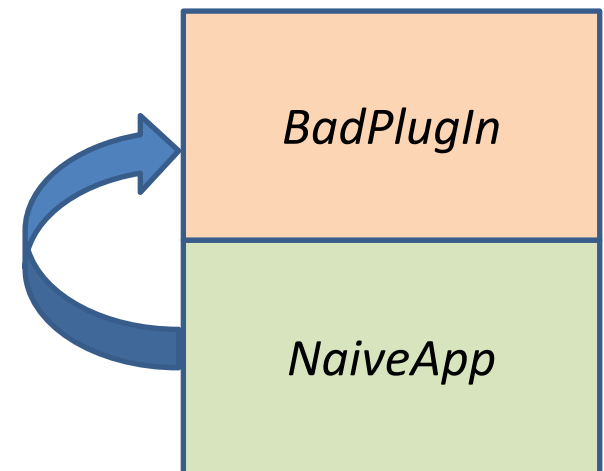
An example where looking at the stack does not suffice

// **Fully trusted but naive**: has all permissions

```
class NaiveApp {  
    public static void Main() {  
        string s = BadPlugIn.TempFile ();  
        File.Write(s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
public class BadPlugIn {  
    public static string TempFile () {  
        return "..\\password";  
    }  
}
```



An example where looking at the stack does not suffice

// **Fully trusted but naive**: has all permissions

```
class NaiveApp {  
    public static void Main() {  
        string s = BadPlugIn.TempFile ();  
        File.Write(s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
public class BadPlugIn {  
    public static string TempFile () {  
        return "..\\password";  
    }  
}
```



NaiveApp

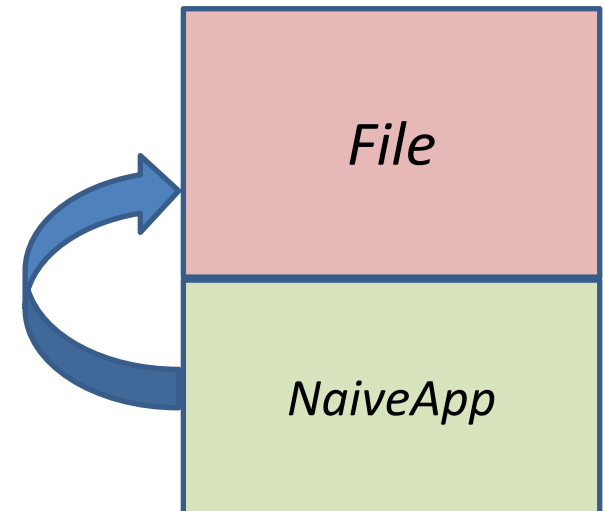
An example where looking at the stack does not suffice

// **Fully trusted but naive**: has all permissions

```
class NaiveApp {  
    public static void Main() {  
        string s = BadPlugIn.TempFile ();  
        File.Write(s, ... );  
    }  
}
```

// **Untrusted**: no FileIOPermission

```
public class BadPlugIn {  
    public static string TempFile () {  
        return "..\\password";  
    }  
}
```



Criticisms

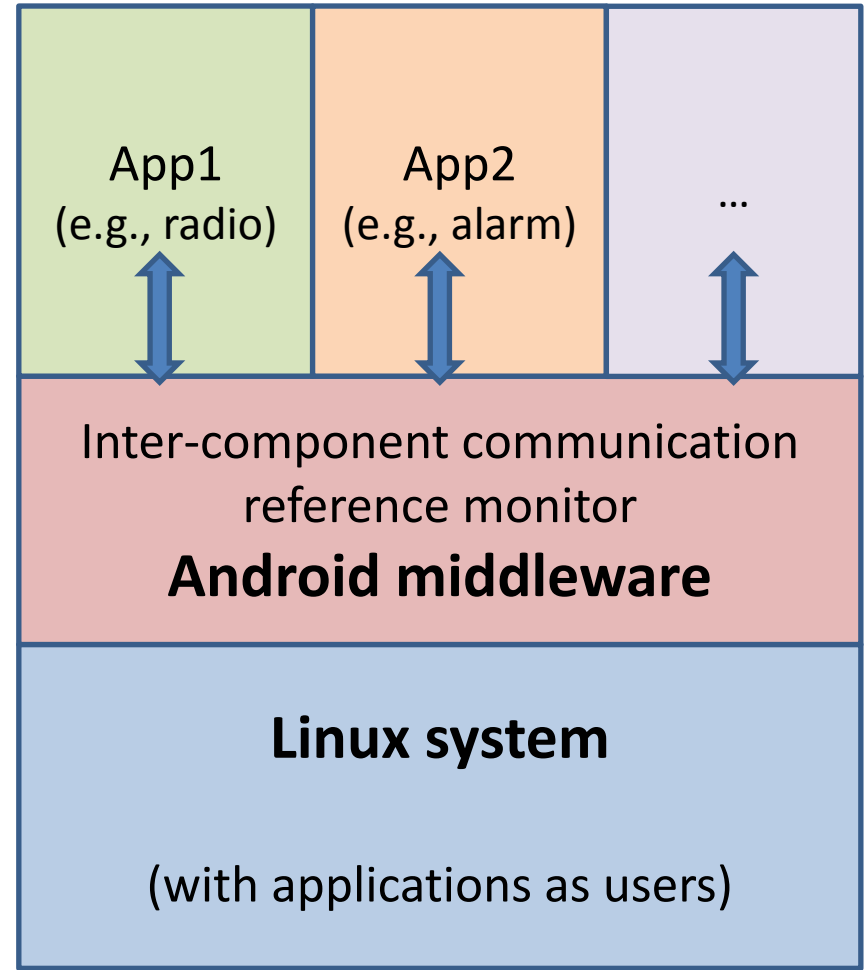
- Does this technique achieve real security?
for what policy?
- Looking at chains of calls is not satisfactory.
 - Some other constructs require careful treatment.
 - A standard formulation (“stack inspection”) is tied to a particular stack implementation.
 - ⇒ It rules out or complicates optimizations.
- It can get hard to understand security.

See “Stack Inspection: Theory and Variants”, by Fournet and Gordon.

Access control in Android

Applications are principals.
Each application comes with fixed permissions

- declared by developer;
- accepted by user at installation time;
- checked at run-time;
- some standard, e.g., access network;
- others defined by developers;
- over 100.



(For many other aspects to Android security, see "Understanding Android Security", by Enck et al..)

*Languages and logics
for access control policies*

From matrices to rules

- An access control matrix may be represented with a ternary predicate symbol **may-access**.
- Other predicates may represent groups, etc..
- We may use standard logical operators.
- We may then write formulas such as:

may-access(Alice, Foo.txt, Rd)

and rules such as:

may-access(p, o, Wr) \rightarrow may-access(p, o, Rd)

good(p) \rightarrow may-access(p, o, Rd)

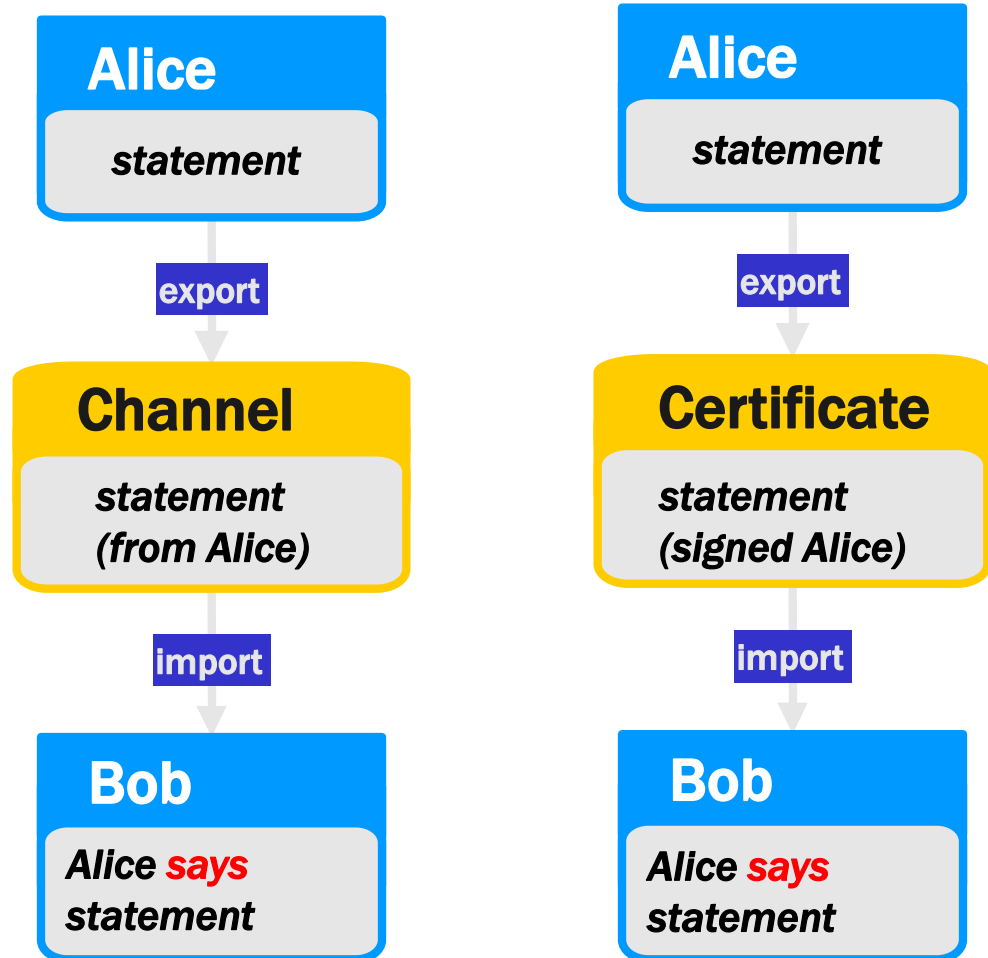
(see **XACML** and the like)

Going further: policies for distributed systems

- In distributed systems, there are multiple sources of assertions, trusted differently.
- This is reflected in some proposed public-key infrastructures, policy languages, and logics.
- One idea is to represent explicitly the principals that make assertions and to reason about them...

Says

- “says” represents communication across contexts.
- It abstracts from the details of authentication.
- The statement may be atomic or a more complex rule.



A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:
 - A says s
 - A speaks for B
- With logical rules, for example:
 - $\vdash A \text{ says } (s \rightarrow t) \rightarrow (A \text{ says } s) \rightarrow (A \text{ says } t)$
 - $\vdash s \rightarrow (A \text{ says } s) \quad \vdash (A \text{ says } A \text{ says } s) \rightarrow (A \text{ says } s)$

A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:
 - A says s
 - A speaks for B
- With logical rules, for example:
 - $\vdash A \text{ says } (s \rightarrow t) \rightarrow (A \text{ says } s) \rightarrow (A \text{ says } t)$
 - $\vdash s \rightarrow (A \text{ says } s) \quad \vdash (A \text{ says } A \text{ says } s) \rightarrow (A \text{ says } s)$
 - $\vdash A \text{ speaks for } B \rightarrow (A \text{ says } s) \rightarrow (B \text{ says } s)$
 - $\vdash A \text{ speaks for } A$
 - $\vdash A \text{ speaks for } B \wedge B \text{ speaks for } C \rightarrow A \text{ speaks for } C$
 - $\vdash (B \text{ says } (A \text{ speaks for } B)) \rightarrow (A \text{ speaks for } B)$

A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:

- A says s

- A speaks for B \equiv for all X . $((A \text{ says } X) \rightarrow (B \text{ says } X))$

- With logical rules, for example:

- $\vdash A \text{ says } (s \rightarrow t) \rightarrow (A \text{ says } s) \rightarrow (A \text{ says } t)$

- $\vdash s \rightarrow (A \text{ says } s)$ $\vdash (A \text{ says } A \text{ says } s) \rightarrow (A \text{ says } s)$

- $\vdash A \text{ speaks for } B \rightarrow (A \text{ says } s) \rightarrow (B \text{ says } s)$

- $\vdash A \text{ speaks for } A$

- $\vdash A \text{ speaks for } B \wedge B \text{ speaks for } C \rightarrow A \text{ speaks for } C$

- $\vdash (B \text{ says } (A \text{ speaks for } B)) \rightarrow (A \text{ speaks for } B)$

A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:

– A says s

– A speaks for B \equiv for all X . $((A \text{ says } X) \rightarrow (B \text{ says } X))$

- With logical rules, for example:

$\vdash A \text{ says } (s \rightarrow t) \rightarrow (A \text{ says } s) \rightarrow (A \text{ says } t)$

$\vdash s \rightarrow (A \text{ says } s) \quad \vdash (A \text{ says } A \text{ says } s) \rightarrow (A \text{ says } s)$

$\vdash A \text{ speaks for } B \rightarrow (A \text{ says } s) \rightarrow (B \text{ says } s)$

$\vdash A \text{ speaks for } A$

$\vdash A \text{ speaks for } B \wedge B \text{ speaks for } C \rightarrow A \text{ speaks for } C$

$\vdash (B \text{ says } (A \text{ speaks for } B)) \rightarrow (A \text{ speaks for } B)$

“same
consequences”

An example

- Let `good-to-delete-file1` be a proposition.
- Let *B controls s* stand for $(B \text{ says } s) \rightarrow s$
- Assume that
 - *B says (A speaks for B)*
 - *B controls good-to-delete-file1*
 - *A says good-to-delete-file1*
- We can derive:
 - *B says good-to-delete-file1*
 - *good-to-delete-file1*

Applications

Several languages rely on logics for access control:

- **D1LP and RT** [Li, Mitchell, et al.]
- **SD3** [Jim] and **Binder** [DeTreville]
- **Daisy** [Cirillo et al.]
- **SecPAL** [Becker, Fournet, and Gordon] and **DKAL** [Gurevich and Neeman]

“says” and “speaks for” play a role in other systems:

- **SDSI and SPKI** [Lampson and Rivest; Ellison et al.]
- **Alpaca** [Lesniewski-Laas et al.] and **Aura** [Vaughan et al.]
- **PCFS (proof-carrying file system)** [Garg and Pfenning]
- ...

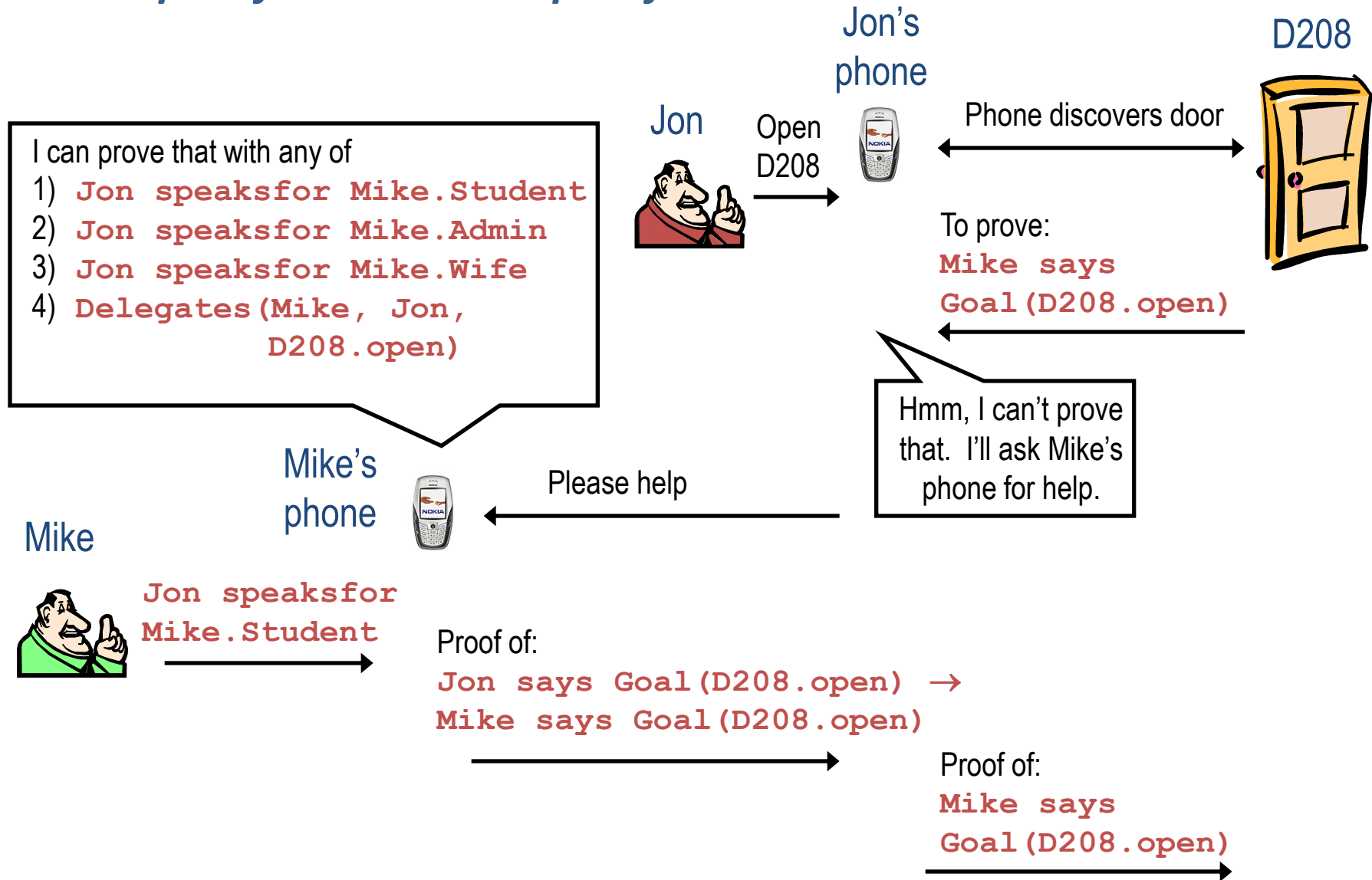
An example system: Grey

[Bauer, Reiter, et al., 2005–2008]

- Turns a cell phone into a tool for delegating and exercising authority.
- Uses cell phones to replace physical locks and key systems.
- Implemented in part of CMU.
- With access control based on logic and distributed proofs.



An example of a distributed proof:



A small language: Binder

- Binder is a relative of Prolog.
- Like Datalog, it lacks function symbols.
- It also includes the special construct says.
- It does not include much else.

- Binder is not the most recent.
- Later systems (e.g., SecPAL) have similarities with Binder, but they are more complex.

An example

- Facts
 - owns(Alice, Foo.txt).
 - Alice says good(Bob).
- Rules
 - may-access(p, o, Rd) :- owns(q, o), blesses(q, p).
 - blesses(Alice, p) :- Alice says good(p).
- Conclusions
 - may-access(Bob, Foo.txt, Rd).

Proof rules

- Binder includes standard logical rules (“resolution”).
- E.g.,
may-access(p, o, Rd) :- owns(q, o), blesses(q, p).
owns(Alice, Foo.txt).
 imply
may-access(p, Foo.txt, Rd) :- Alice says good(p).

Proof rules: importing

- In addition, formulas from a context F can be imported to a context D.
 - This adds “F says” in front of all atoms without a “says”.
 - It applies only to clauses where the head atom does not have “says”.

Importing by example

- Suppose F has the rules
 - $\text{may-access}(p, o, Rd) \text{ :- owns}(q, o), \text{blesses}(q, p)$.
 - $\text{blesses}(\text{Alice}, p) \text{ :- Alice says good}(p)$.
 - $\text{Alice says good}(\text{Bob})$.
- D may import the first two as:
 - F says $\text{may-access}(p, o, Rd) \text{ :-}$
 F says $\text{owns}(q, o), \text{F says blesses}(q, p)$.
 - F says $\text{blesses}(\text{Alice}, p) \text{ :- Alice says good}(p)$.
- D may import $\text{good}(\text{Bob})$ directly from Alice.

Importing by example (cont.)

- Suppose F has the rule
 - $\text{blesses}(\text{Alice}, p) \text{ :- Alice says good}(p)$.
- D may import it as:
 - F says $\text{blesses}(\text{Alice}, p) \text{ :- Alice says good}(p)$.
- D and F should agree on Alice's identity.
- But the meaning of predicates may vary, and it typically will.

For example, F may also have:

- $\text{blesses}(\text{Bob}, p) \text{ :- Bob says excellent}(p)$.

Important properties

- Policies can use application-specific predicates.
- Statements can be read declaratively.
- Queries are decidable (typically in PTime).

Issues and research directions

- What about algorithmic problems?
- What about more proof systems? Semantics?
- Can all reasonable policies be expressed?
Can the simple ones be expressed simply?
- Is this a way of explaining other approaches?
or something for direct use (e.g., as XACML)?



Reading

- Chen et al.'s "Setuid Demystified"
<http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf>
- Fournet & Gordon's "Stack Inspection: Theory and Variants"
<http://dl.acm.org/citation.cfm?id=641912>
- Enck et al.'s "Understanding Android Security"
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4768655>

Homework 2 (due October 11)

Exercise 1:

Consider a system with users U_1, \dots, U_m . The system has functionality for letting the users play a fixed, one-player game, for charging \$1 to a user account, for resetting a user account to \$0, and for showing the balance on a user account. Informally, its policy is:

- a) Users can play with a charge of \$1 each time.
- b) Only user U_1 can reset user accounts.
- c) The balance on a user account can be seen only by the user and by U_1 .

Express this policy as an access control matrix. Be explicit on the definitions of subjects, objects, and operations. (E.g., if the subjects are U_1, \dots, U_m , say so.)

Homework 2

Exercise 2:

Comment on how one of the principles of Saltzer and Schroeder is followed or disregarded in some aspect of a contemporary system that you know.

A paragraph should suffice.

Homework 2

Exercise 3:

Lampson's paper "Protection" (pp. 2–3) describes a simple scheme for identifying processes in systems. Comment on how it applies, or not, when the system in question is the Web.

A paragraph should suffice.