

Mechanisms (Access control)

Access control

Access control is prominent at many levels:

- memory management hardware,
 - operating systems, file systems, and related services,
 - middleware,
 - applications,
 - in particular, sandboxing for mobile code,
- and (with separate bodies of techniques):
- firewalls,
 - physical protection.

It is a mechanism and a model.

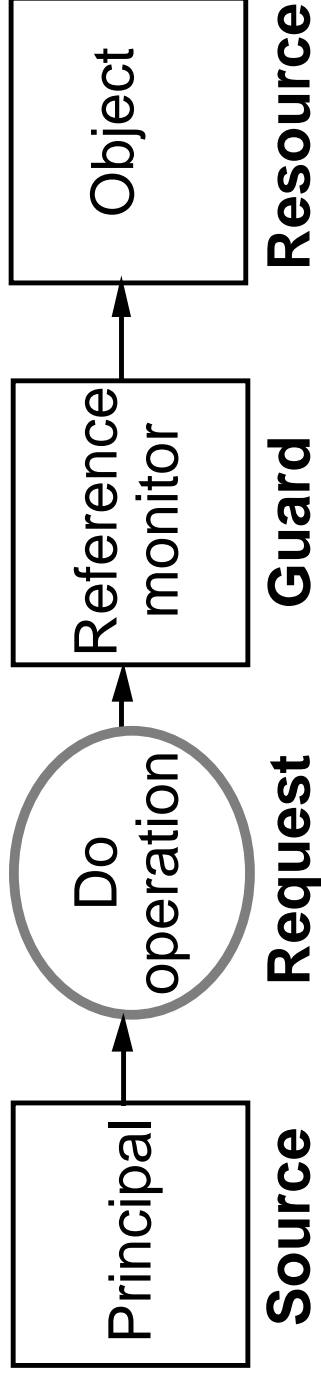
We will aim to cover:

- fundamentals of access control,
- how they get complicated, and why,
- bits of concrete examples.

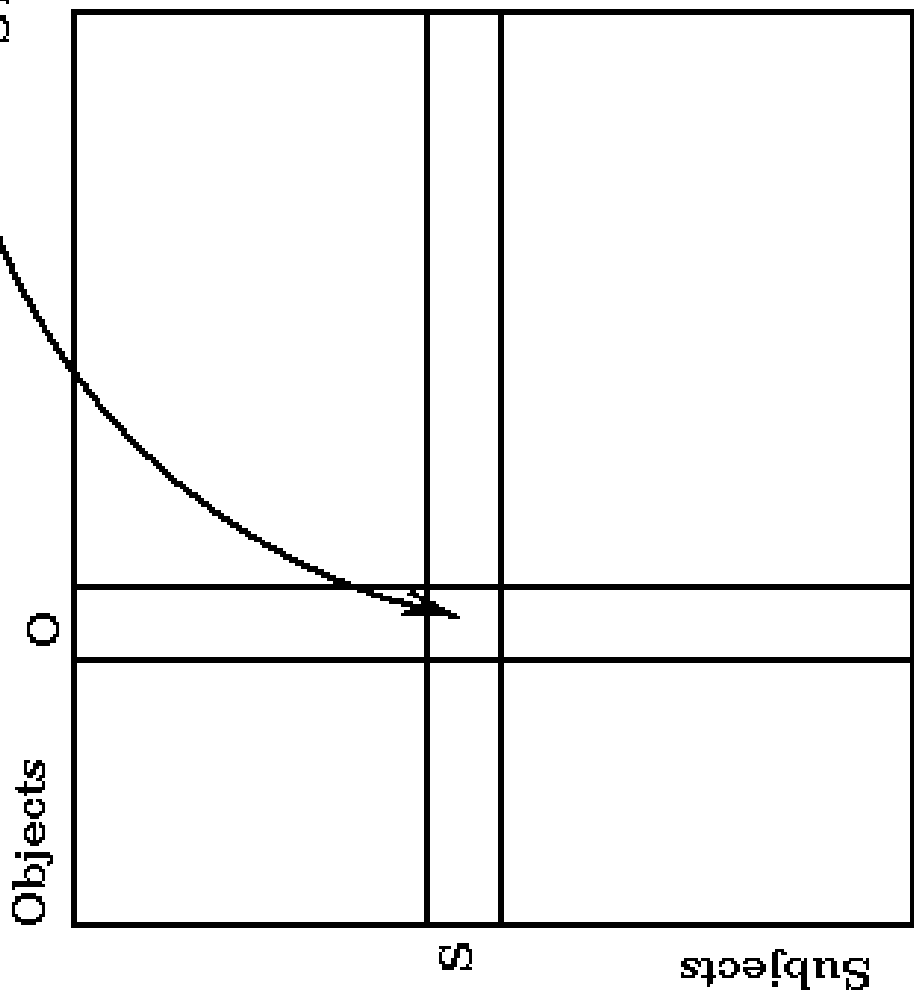
The access control model

Elements:

- **objects**, resources
- **requests** for **operations** on objects
- sources for requests, called **principals**
- a **reference monitor** to decide on requests



access rights that
S has to O: [S, O]



Authentication

Access control:

Is A trusted on s ?

If A requests s , should s be granted?

Access control is based on **authentication**:

Who says s ?

Other mechanisms

“Before” and “after” :

- Set-up.
- Auditing.
- Recovery.

And more:

- Cryptography, particularly across networks.
- Inference control in databases.
- ⋮

Lampson's Golden Rule

Authentication.

Authorization.

Audit.

The principle of complete mediation

Every access to every object is checked.

This principle can be enforced in several ways:

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

More principles

Least privilege

Separation of privilege

Failsafe defaults

Diversity of mechanism, multiple lines of defense

Psychological acceptability

These occasionally conflict with one another.

Implementing access control

Two strategies (often combined): ACLs and capabilities.

ACL: a column of an access control matrix.

Capability: (basically) a pair of

- an object,
- an operation.

The holder may perform the operation on the object.

Implementing access control: ACLs

An **access control list** says which subjects 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.
- ACLs can be easy to review.
- Revoking a subject can be painful.

Implementing access control: capabilities

An alternative is to associate **capabilities** with subjects.

These capabilities form a row of an access control matrix for the subject.

- Capabilities are easy to pass around, so they enable delegation.
- They can be hard to revoke.

Implementing capabilities

A capability should convey an object and an operation. It means that the holder can perform the operation on the object.

Subjects should not be allowed to forge capabilities.

This leads to implementations of capabilities:

- stored in a protected address space,
- with special tags with hardware support,
- :

Implementing capabilities (cont.)

Capabilities may also be typed references in a suitable programming language:

- Objects are objects in a typed language like Java.
- Each capability is or contains an object reference.
- It cannot be forged, e.g., by casting an integer.

Implementing capabilities (cont.)

Or each capability may contain a random number:

- The random number serves as a password for an object.
- The random number should be hard to guess.
- The random number may be the object's address.

Implementing capabilities (cont.)

Or we may rely on cryptography:

- The capability may include a certificate for its meaning, i.e., the holder's right to access.
- The certificate should be digitally signed by the issuer of the capability.
- The certificate may include additional constraints, e.g., a time of validity.

Implementing capabilities (summary)

When each subject keeps its capabilities, it should not be allowed to forge them.

⇒ Sophisticated implementations of capabilities:

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

Comparisons

ACLs and capabilities are dual.

Both yield practical implementations of access matrices.

In actual systems, they are often combined.

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 .

Groups and roles

Suppose that any member of a group G may access a file f owned by A .

- G may be maintained by someone other A .
- The group may change over time, without immediate knowledge of A .
- The ACL for f should be short and clear.
- Proofs of memberships resemble (are?) capabilities.
- Access to f might be partly anonymous.
- Still, A may require a proof of identity at each f access, for auditing.

Disjunctions

A or *B* is a little group, with members explicitly listed.

A may weaken its authority to *A* or FlakyProgram.

A request from *A* or FlakyProgram will be granted only if it would be granted both to *A* and to FlakyProgram.

Conjunctions

An operation may be granted only if it is requested by a conjunction of two principals A and B .

This is an instance of dual control.

Negations

Some systems (e.g., VMS) include also negations.

Then ACLs sometimes become order-sensitive!

Combinations of principals (summary)

- Groups.
- Principals in roles (much like principals in groups?).
- Disjunctions of principals (A or B), as small groups.
- Conjunctions of principals (A and B), for dual control.
- Negations in ACLs.
- Delegations (not yet discussed in class).

Some of these are supported in standard platforms.

Others appear in applications and in some research operating systems.

More on objects and operations

Objects and operations may also be put in groups, e.g.,

- all company files,
- all read operations on an object.

Sometimes operations should be bundled, e.g.,

- read a patient's record,
- write a log record.

Design issues

- 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.

Programs

Programs may be principals too.

But then:

- we need to deal with call chains,
e.g., applet on browser on OS,
- we still need to connect programs to other principals
 - who write them or edit them,
 - who provide them,
 - who install them,
 - who call them.

Running programs: rights

What are the rights of a program?

- those of the caller,
- those of the program owner, or
- some combination, or
- something else, e.g, because of intrinsic properties.

E.g., the program that moves incoming mail to a user's inbox may need to combine system rights and user rights.

Installing programs

Programs should be set up so that they get appropriate rights when they run.

One should install and run only trusted programs, or use compartments by trust level.

Programs should be adequately protected from editing.

Running programs: protection and confinement

At run-time, programs should be protected and confined, that is, limited to communicate over proper interfaces.

This is often the job of the 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 finer control over communication.

(Remember the implementations of capabilities?)

Common dangers

Access control can be insufficient or irrelevant

- when it is implemented incorrectly,
- when the underlying operations are implemented incorrectly,
- when dangerous code is privileged,
- when it is circumvented.

Common dangers: bugs in access control

Hopefully not a big issue.

But sometimes even simple security machinery can be the source of insecurity.

And the underlying authentication can be broken.

WATCH BBC NEWS IN VIDEO

BBC NEWS UK EDITION

Last Updated: Tuesday, 20 April, 2004, 01:44 GMT 02:44 UK

E-mail this to a friend Printable version

Passwords revealed by sweet deal

More than 70% of people would reveal their computer password in exchange for a bar of chocolate, a survey has found.

It also showed that 34% of respondents volunteered their password when asked without even needing to be bribed.

A second survey found that 79% of people unwittingly gave away information that could be used to steal their identity when questioned.

Security firms predict that the lax security practices will fuel a British boom in online identity theft.

Security shock

The survey on passwords was carried out for the Infosecurity Europe trade show due to take place at Olympia in London from 27-29 April.

The survey data was gathered by questioning commuters



Security crumbles in the face of sweet bribes

WATCH AND LISTEN

The BBC's Quentin Sommerville
"Strangers asking for your mother's maiden name or date of birth should ring alarm bells"

▶ VIDEO

SEE ALSO:

- ▶ Britain sees surge in 'phishing' 25 Mar 04 | Business
- ▶ 'I'm calling about breast cancer insurance' 10 Mar 04 | Magazine
- ▶ Mice sign on the dotted line 01 Sep 03 | Technology
- ▶ How secure is your password? 10 May 02 | Sci/Tech
- ▶ PCs 'infested' with spy programs 16 Apr 04 | Technology
- ▶ Criminals cash in on passports 27 Feb 04 | UK

RELATED INTERNET LINKS:

- ▶ Infosec Europe
- ▶ RSA Security
- ▶ National Hi-Tech Crime Unit

- Front Page
- World
- UK
- England
- Northern Ireland
- Scotland
- Wales
- Business
- Politics
- Health
- Education
- Nature
- Technology
- Entertainment
- Have Your Say
- Magazine
- In Pictures
- at a Glance
- Country Profiles
- In Depth
- Programmes

- SPORT
- WEATHER
- 10 NEWS
- ON THIS DAY

Common dangers: bugs in underlying operations

There are lots.

When security-sensitive code is properly isolated, most bugs should not be the subject of this course...

Common dangers: privileged dangerous code (a)

Why did it run in the first place?

The user runs foreign code with privileges by default, on whims, or for lack of better options.

For example:

- all Visual Basic scripts in incoming mail,
- because of a nice subject line (I Love You),
- because it claims to be just the right device driver.

Pieces of a solution:

- Educated users.
- Safer languages for mobile code.
- Additional in-line reference monitors.
- Finer delegations of privileges.
- Signed code.
- Virus scanners.

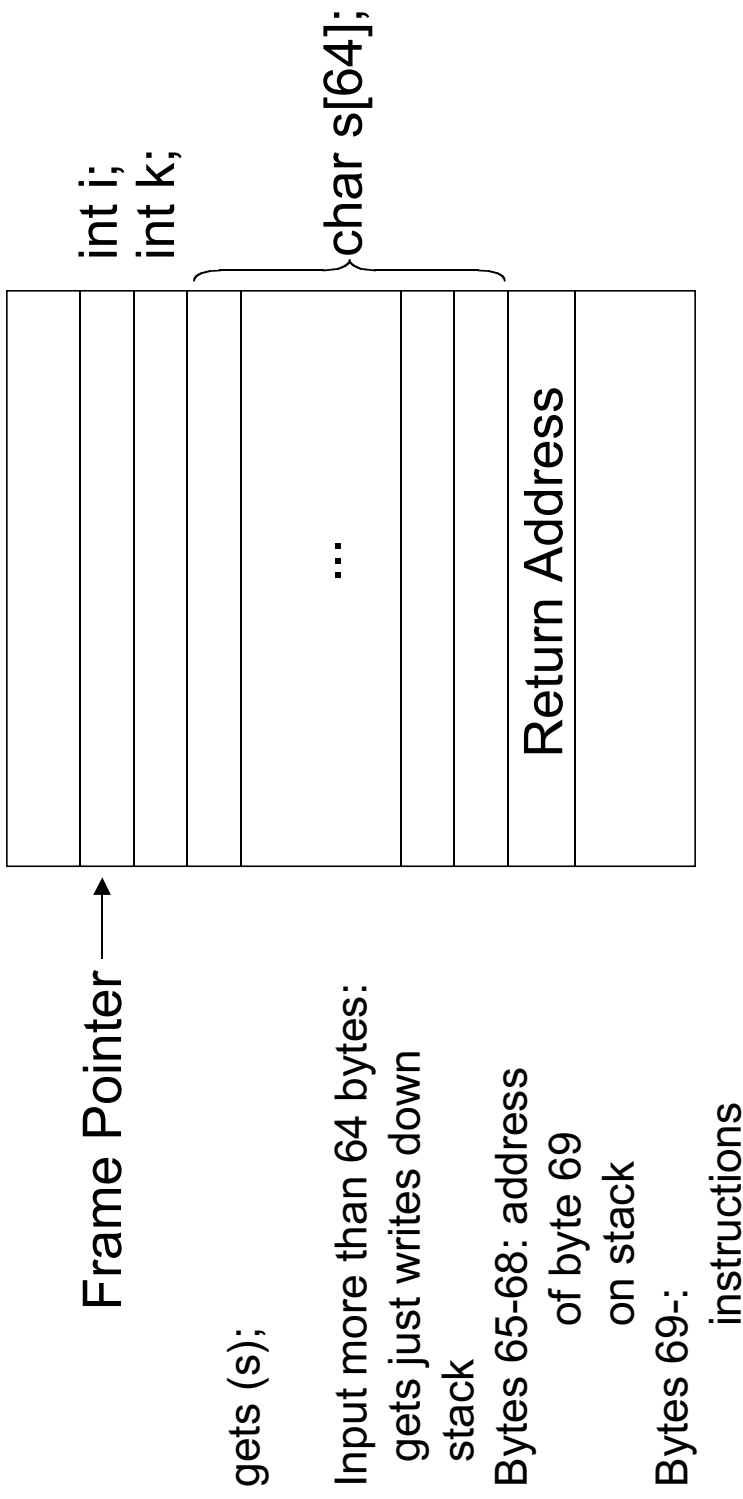
Common dangers: privileged dangerous code (b)

Why did it run in the first place?

Some other was meant to run, but the attack exploits a bug and subverts execution!

(Úlfar Erlingsson's guest lecture will contain more on this.
What follows is largely a preview.)

Example: classic buffer overflows



- The tail of a long argument smashes the return address on the stack.
- Upon a “return”, control jumps to malicious code.

Pre-reading on buffer overflows and more

A tutorial by Aleph One:

- “Smashing the stack for fun and profit”
at www.insecure.org/stf/smashstack.txt.

A paper by Pincus and Baker:

- “Beyond stack smashing: Recent advances in exploiting
buffer overruns”

reachable from research.microsoft.com/users/jpincus/.

Software security

Some approaches to avoiding buffer overflows and other software flaws:

- Static analysis.
- Safe libraries.
- Better programming languages.

Some approaches to mitigating their impact:

- Separate code and data segments,
 - don't allow code modifications,
 - don't allow jumps into data.

∴

Additional references (in case you are interested)

A paper by Cowan et al.:

- “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks”

at

www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf.

A paper by Wagner et al.:

- “A first step towards automated detection of buffer overrun vulnerabilities”

reachable from www.cs.berkeley.edu/~daw/papers/.

Common dangers: access control circumvented

Sometimes the reference monitor does not protect all important objects and operations, or does not protect them all the time.

- Race conditions.
 - ⇒ Use better locking, maybe static analysis.
- Data recovery from disks.
 - ⇒ Make sure sensitive data stays in RAM, or learn to erase data from disks.
(See www.usenix.org/events/lisa04/tech/talks/garfinkel.pdf.)
- Hostile platforms (e.g., for DRM systems).

Common dangers (cont.)

There is a lot more to say on how to break security (e.g., on famous viruses).

Much of it has to do (legitimately) with user psychology.

Discretionary access control

- leaves security policy to each subject,
- is intrinsically limited in saving subjects from themselves.

Access control in Unix

Subjects are users (plus root).

Objects are files.

Each file has an owner and a group.

Operations are:

- read (**r**),
- write (**w**),
- execute (**x**).

This makes some sense for directories too.

- **r** means listing.
- **x** means using in constructing paths.

Access control in Unix (cont.)

ACLs are typically limited to 9 bits: `rwX` for each of

- user (**u**),
- group (**g**),
- others (**o**).

So for example `rw-r--r--` allows the owner to write the file and everyone to read it.

Access control in Unix (cont.)

Only the owner of a file (and root) can change its ACL (chmod it).

If a program file is marked as `suid` by its owner, then the program executes with the privilege of the owner (not that of the caller).

And there is also `sgid`.

Access control in Windows

Fundamentally, Windows NT is much like Unix.

It is richer:

- It is easier to obey the principle of least privilege.
- Users are organized into domains.

Windows 2000 is even richer.

Questions on access control

Can we dismiss most of access control as a leftover from timesharing? (Probably not.)

Has it gotten better or worse?

What should platforms provide?

Will applications use it?

How much should we leave to applications?

Will it be secure

- in theory?

- in practice?

Access control is pervasive in:

- applications,
- virtual machines,
- operating systems,
- firewalls,
- doors,
- ;

Access control seems difficult to get right.

Distributed systems make it harder.

General theories and systems

Over the years, there have been many theories and systems for access control.

- Logics
- Languages
- Infrastructures (e.g., PKIs)
- Architectures

They often aim to explain, organize, and unify access control.

They may be intellectually pleasing.

They may actually help.

An approach

A notation for representing principals and their statements, and perhaps more:

- objects and operations,
- trust,
- channels,
- :

Derivation rules

Reading

Two papers by Lampson et al.:

- “Authentication in distributed systems: Theory and practice”,
- “Computer security in the real world”

reachable from

www.research.microsoft.com/users/blampson/,

and DeTreville’s “Binder, a logic-based security language”
at

<ftp://ftp.research.microsoft.com/pub/tr/tr-2002-21.pdf>.

“Authentication ...”

A **says** relation represents communication across contexts and abstracts from the details of authentication.

A **speaks-for** relation explains the relations between principals, uniformly.

Delegations, roles, conjunctions, etc., come into authorization decisions.

We may represent them in principals in order to exploit the access-control model.

A logic is not a proof of security, but it helps

- as a foundation,
- in guiding algorithms,
- for auditing.

Access-control logic

Basic access-control logic

$A \text{ says } s$

$A \wedge B$ “A and B”

$B \mid A$ “B quoting A”

$A = B$

Defined notions

$A \text{ controls } s = (A \text{ says } s \Rightarrow s)$

$A \text{ as } R = A \mid R$

$B \text{ for } A = (B \wedge D) \mid A$

D is a fictional delegation server

$A \Rightarrow B = (A = A \wedge B)$

A is stronger (more trusted) than B

Access-control logic (cont.)

Axioms

$(B \mid A) \text{ says } s \equiv B \text{ says } A \text{ says } s$

$A \text{ says } s \wedge A \text{ says } (s \Rightarrow t) \Rightarrow A \text{ says } t$

...

Deciding access

Problem:

given a principal A that makes a request,
an ACL B_0, \dots, B_i, \dots
and assumptions (e.g., memberships),
decide whether A should have access.

Other languages and systems

PolicyMaker and KeyNote [Blaze et al.]

SDSI [Lampson and Rivest]

SPKI [Ellison et al.]

D1LP and RT [Li et al.]

SD3 [Jim]

Binder [DeTreville]

XrML 2.0

:

Several of the most recent are based on ideas and techniques from logic programming.

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 programs can define and use new, application-specific predicates.

Queries in Binder are decidable (in PTime).

An example in Binder

Facts:

owns(Alice, foo.txt).

Alice says good(Bob).

Rules:

may-access(p,o) :- owns(q,o), blesses(q,p).

blesses(Alice,p) :- Alice says good(p).

Conclusions:

may-access(Bob,foo.txt).

Binder's proof rules

Suppose F has the rules

$\text{may-access}(p,o) :- \text{owns}(q,o), \text{blesses}(q,p).$

$\text{blesses}(\text{Alice},p) :- \text{Alice says good}(p).$

Some context D may import them as:

$\text{F says may-access}(p,o) :- \text{F says owns}(q,o), \text{F says blesses}(q,p).$

$\text{F says blesses}(\text{Alice},p) :- \text{Alice says good}(p).$

D and F should agree on Alice's identity.

The meaning of predicates may vary.

$\text{good}(p) :- \text{Alice says good}(p).$

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

Homework 2 (for April 20)

This homework consists of two exercises.

Exercise 1:

Consider a game 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.)

Exercise 2:

The following questions are about the definition of security by Joshi and Leino. If in doubt, you may use the relational version of the definition. For questions (a–g), you may simply answer “yes”, “no”, or “it depends” .

- a) Is HH secure?
- b) Is **true** secure?
- c) Is $k := k * h$ secure? (* represents multiplication)
- d) Is $k := n * h$ secure, where n is some integer constant?
- e) Is $h := 0; k := h$ secure?

Exercise 2 (cont.):

- f) If S and T are secure, is **if** $k > 0$ **then** S **else** T **end** also secure?
- g) If S and T are secure, is their composition $S;T$ also secure?
- h) Give a rigorous proof of your answer to (e), using the relational semantics.
- i) Briefly explain the meaning (or significance) of your answers to (f) and (g). (You may want to look again at McLean's discussion of composability.)

Homework 3 (for April 27)

Consider a system with three users A, B, and C, two objects O and P, and one operation on the objects.

Consider the following policy (written in Binder notation):

```
trusted(A).  
A says trusted(B).  
A says may-access(C,O).  
may-access(x,O) :- trusted(y), y says may-access(x,O).  
may-access(x,P) :- trusted(x).  
trusted(x) :- trusted(y), y says trusted(x).
```

(x and y are variables that range over A, B, and C.)

Write the resulting access control matrix.