Access Control Mechanisms

Sources:

- Chapter 15 of *Computer Security: Art and Science* by Matt Bishop
- Chapter 2 of *Security in Computing*, 5th ed., by Pfleeger et al.
- *Role-Based Access Control Models* by R. S. Sanhu and E. J. Coyne, IEEE Computer, February 1996
- *Proposed NIST Standard for Role-Based Access Control* by D. F. Ferraiolo, R. S. Sandhu, and S. Gavrila, ACM Transactions on Information and System Security, August 2001
- *Attribute-Based Access Control* by B. Fisher et al., NIST Cybersecurity Practice Guide, NIST SP 1800-3b, 2015
- *Usage Control in Computer Security: A Survey* by A. Lazouski, F. Martinelli, and P. Mori, Computer Science Review 4, 2010

- *Attribute-Based Access Control* by V. C. Hu et al., IEEE Computer, February 2015

An *access control mechanism* is used to permit a subject too access an object in a particular mode.

Unauthorized subjects are *not* permitted to access the object.

Terminology:

- *Subject*: human user or program running on their behalf

- *Object*: thing on which an action can be performed, e.g.,
  - Files, tables, memory objects, devices, strings, data fields, network connections, processors

- *Access modes*: any controllable actions of subjects on objects, e.g.,
  - Read, write, modify, delete, execute, create, destroy, copy, export, import

Access Policies

Access control decisions are based on a formal or informal *access policy*, which an organization should develop carefully.

See LSE access control policy (Perkins; www.lse.ac.uk/intranet/LSEServices/policies/pdfs/school/accConPol.pdf).

Goals of policy implementation:

- *Check every access* – access may be revoked
- *Enforce the principle of least-privilege* – access to the fewest resources necessary to accomplish task
- *Verify acceptable usage* – check not only what is accessed but how it is accessed

Implementing Access Control

Access control is often performed by the operating system.

For finer grained access control, the OS may defer to a database manager or network appliance.

Access control may be implemented by a construct called a *reference monitor*.

It should correctly validate every access attempt and be immune from tampering.

Access Control Lists

Definition. Let S be the set of subjects and let R be the set of rights of a system. An *access control list* (*ACL*) is a set of pairs $l = \{ (s, r) : s \in S, r \subseteq R \}$.

Let *acl* be a function that returns the access control list associated with an object.

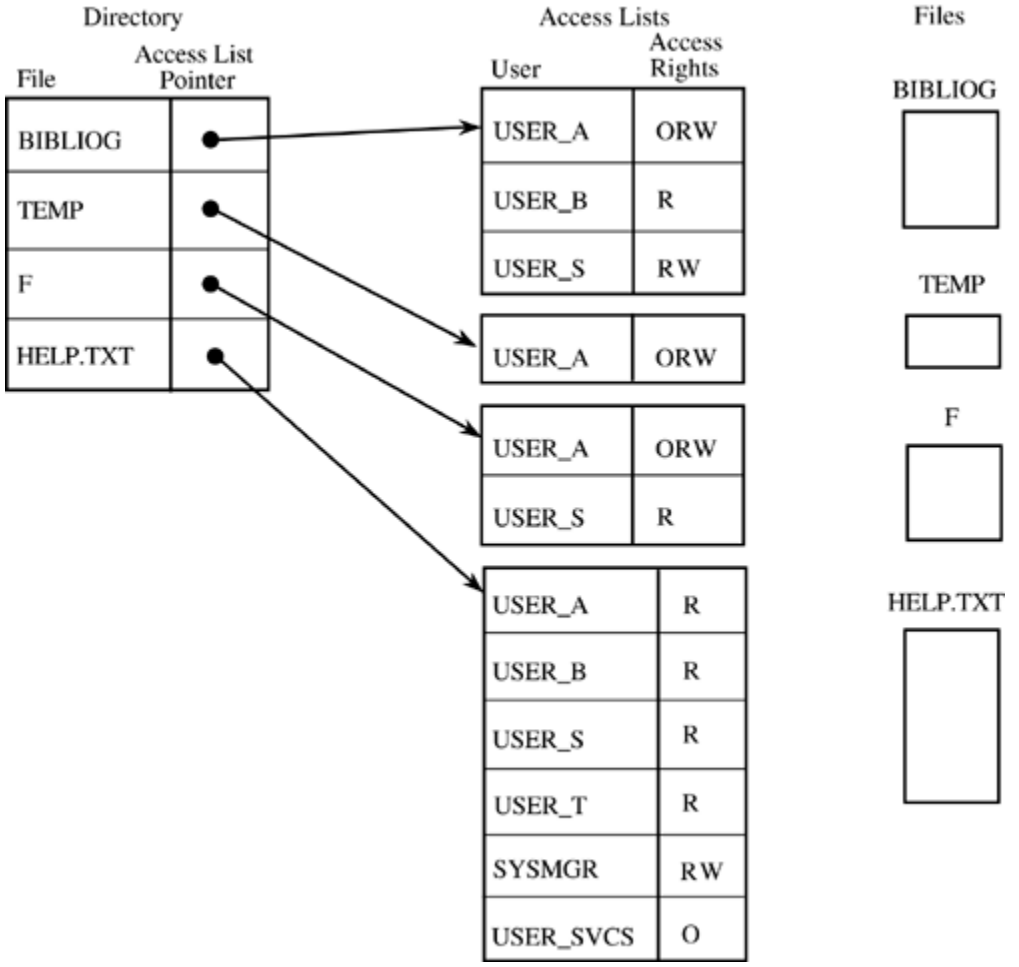
The interpretation of the access control list $acl(o) = \{ (s_i, r_i) : 1 \leq i \leq n \}$ is that subject s_i may access o using any right in r_i.

If a subject is not named in the ACL, it has no rights over the associated object.

Example:

acl(file 1) = { (process 1, {read, write, own}),
(process 2, {append}) }

acl(process 1) = { (process 1, {read, execute,
own}), (process 2, {read}) }



Access Control List [Pfleeger et al.]

Some systems abbreviate access control lists.

Example: UNIX systems recognize three classes of users with distinct sets of rights: the *owner* of a file, the *group owner* of the file, and *all other users*.

UNIX systems provide *read* (*r*), *write* (*w*), and *execute* (*x*) rights.

UNIX permissions are represented as *three triplets*, corresponding to owner rights, group rights, and other rights, respectively.

If the permissions for a file are *rw-r-----*, this means that the *owner* has *read/write* permission, the *group* has *read* permission, and *no one else has access* to the file.

Such abbreviated access control lists have limited flexibility.

Example: IBM's version of UNIX, called AIX, uses an ACL ("extended permissions") to augment traditional UNIX ACL abbreviations.

```
attributes:
base permissions
    owner(bishop):    rw-
    group(sys):       r—
    others:           ---
extended permissions enabled
    specify           rw-      u:holly
    permit            -w-      u:heidi, g=sys
    permit            rw-      u:matt
    deny              -w-      u:holly, g=fac
```

"**specify**" means to override the base permissions.

"**permit**" means to add rights.

"**deny**" means to remove rights.

"**u:**" means user; "**g=**" means group.

When an ACL is created, rights are instantiated.

Possessors of the *own* right can *modify* the ACL.

The *creator* of an object is typically given *all* rights, including *own*, initially.

Typically, ACLs are applied in a limited way to administrative users (e.g., *root*).

Example: *Solaris* has both abbreviated and full ACLs. Only the latter apply to *root*.

Some systems allow *wildcards* to be used in specifying permissions.

Example: In UNICOS, the permission string

```
holly : * : r
```

gives a process with UID **`holly`** *read* access to an object, regardless of the process's group (GID).

A subject's access rights to an object can be *revoked* simply by removing them from the object's ACL.

Revocation of one subject’s rights may affect those of another if they were granted by the former subject.

Access Control Matrix

This describes the rights of users over objects in the form of a matrix.

Example:

Access Control Matrix

Subject	File1	File2	File3	File4
Larry	Read	Read,Write	Read	Read,Write
Curly	Full Control	No Access	Full Control	Read
Mo	Read,Write	Full Control	Read	Full Control
Bob	Full Control	Full Control	No Access	No Access

[cdn.aiotestking.com/wp-content/uploads/cissp/10.png]

Capabilities

Definition. Let O be the set of objects and let R be the set of rights of a system. A *capability list* is a set of pairs $c = \{ (o, r) : o \in O, r \subseteq R \}$.

Let *cap* be a function that returns the capability list associated with a particular subject.

The interpretation of the capability list $cap(s) = \{ (o_i, r_i) : 1 \leq i \leq n \}$ is that subject s may access o_i using any right in r_i.

Example:

> *cap*(process 1) = { (file 1, {read, write, own}), (file 2, {read}), (process 1, {read, write, execute, own}), (process 2, {write}) }

Capabilities *encapsulate* object identity.

When a process *presents* a capability on behalf of a user, the operating system examines it to determine the object and the access to which the process is entitled.

Note that the *location* of the object in storage is *encoded* in the capability.

It is more *efficient* to check capabilities than to check ACLs at runtime.

Delegation of capabilities is also simple.

Changing a file's status can be more difficult with capabilities, because it is harder to determine which users have access.

A process must *not* be able to *forge* or *modify* a capability.

A *crpytographic checksum* can be used to prevent modification of a capability.

To prevent forgery, a server that creates an object can append a *random number* to a capability and record the number.

When the capability is presented to the server, it *verifies* the random number.

The ability to *copy* capabilities implies the ability to *give rights*.

A process cannot copy a capability unless the *copy flag* is set.

A capability may need to be temporarily *amplified*, e.g., during execution of a module implementing an abstract data type.

Role-Based Access Control (RBAC)

With RBAC, system administrators:

- Create *roles* according to the *job functions* in an organization
- Grant *access permissions* to those roles
- *Assign users* to roles based on their job responsibilities and qualifications

A role can represent:

- A specific *task competency* (e.g, physician or pharmacist)
- The *authority and responsibility* of, say, a project supervisor
- Specific *duty assignments* rotated through users (e.g., duty physician or shift manager)

Role-based access control terms and concepts

Access—A specific type of interaction between a subject and an object that results in the flow of information from one to the other.[1]

Access control—The process of limiting access to the resources of a system only to authorized programs, processes, or other systems (in a network).[2]

Administrative role—A role that includes permission to modify the set of users, roles, or permissions, or to modify the user assignment or permission assignment relations.

Constraint—A relationship between or among roles.

Group—A set of users.

Object—A passive entity that contains or receives information.[1]

Permissions—A description of the type of authorized interactions a subject can have with an object.[2]

Resource—Anything used or consumed while performing a function. The categories of resources are time, information, objects, or processors.[1]

Role—A job function within the organization that describes the authority and responsibility conferred on a user assigned to the role.

Role hierarchy—A partial order relationship established among roles.

Session—A mapping between a user and an activated subset of the set of roles the user is assigned to.

Subject—An active entity, generally in the form of a person, process, or device, that causes information to flow among objects or changes the system state.[1]

System administrator—The individual who establishes the system security policies, performs the administrative roles, and reviews the system audit trail.

User—Any person who interacts directly with a computer system.[1]

References

1. US Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, Washington, D.C., US Department of Defense, Dec. 1985.
2. US Department of Defense, National Computer Security Center, *Glossary of Computer Security Terms*, NCSC-TG-004-88, Ft. Meade, Md., National Computer Security Center, Oct. 21, 1988.

(From Sanhu and Coyne)

In RBAC, it should be:

- Roughly *as easy* to determine role *memberships* as role *permissions*

- Control of role memberships and permissions should be relatively *centralized* in a few users

RBAC *simplifies security administration*.

Example: When a user changes jobs within an organization, they can simply be assigned a new role and removed from their old one.

It is unnecessary to add and revoke individual permissions.

RBAC directly supports three security principles:

- *Least privilege:* Only those permissions required to carry out a role are assigned to it.

- *Separation of duties:* Invocation of *mutually exclusive* roles can be required to complete a sensitive task.

  Example: A ship's commander and a weapons officer must participate in launching a missile.

- *Data abstraction:* Application-specific, abstract permissions can be established.

  Example: Credit and debit permissions for an account object.

Static RBAC does *not* deal with the control of *operation sequences*.

Reference Model

The NIST RBAC reference model is defined in terms of four model components:

- *Core RBAC*
- *Hierarchical RBAC*
- *Static Separation of Duty Relations*
- *Dynamic Separation of Duty Relations*

Core RBAC

Core RBAC embodies the essential aspects of RBAC.

The basic concepts of RBAC are that:

- Users are assigned to roles.
- Permissions are assigned to roles.
- Users acquire permissions by being members of roles.

The same user can be assigned to multiple roles, and a single role can have multiple users.

A single permission can be assigned to many roles, and a single role can be assigned to many permissions.

Users may establish *sessions* during which they activate a subset of the roles they are assigned to.

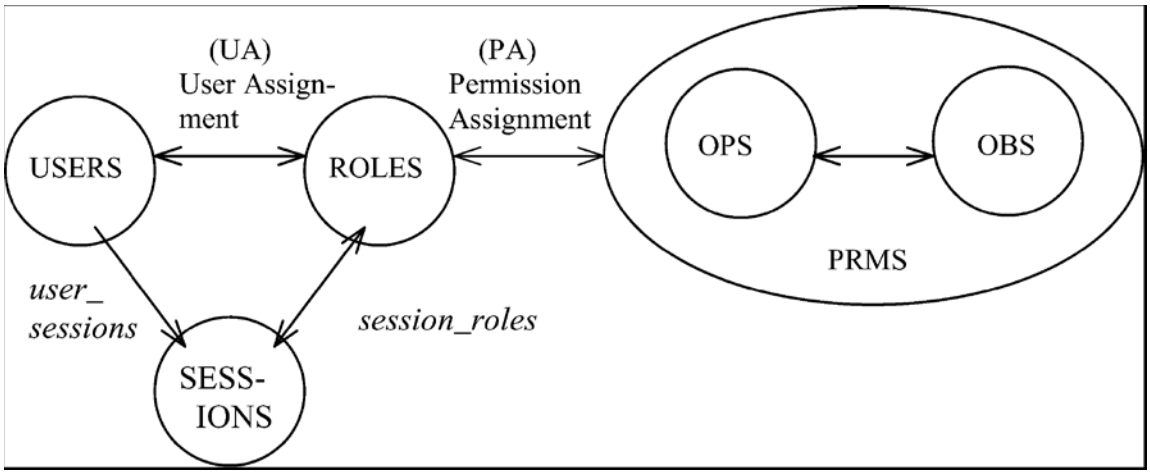
Permissions available to the users are the union the permissions associated with the roles activated in that session.

Each session is associated with a single user.

A user might have multiple sessions open simultaneously.

Each session might combine different active roles.

Core RBAC requires that it be possible to review such assignments.



Core RBAC [Ferraiolo *et al*]

OPS denotes *operations*.

OBS denotes *objects*.

PRMS denotes *permissions*.

Hierarchical RBAC

Hierarchies are a means of *structuring roles* to reflect an organization's lines of authority and responsibility.

By convention, more powerful roles are shown toward the top of role hierarchies.

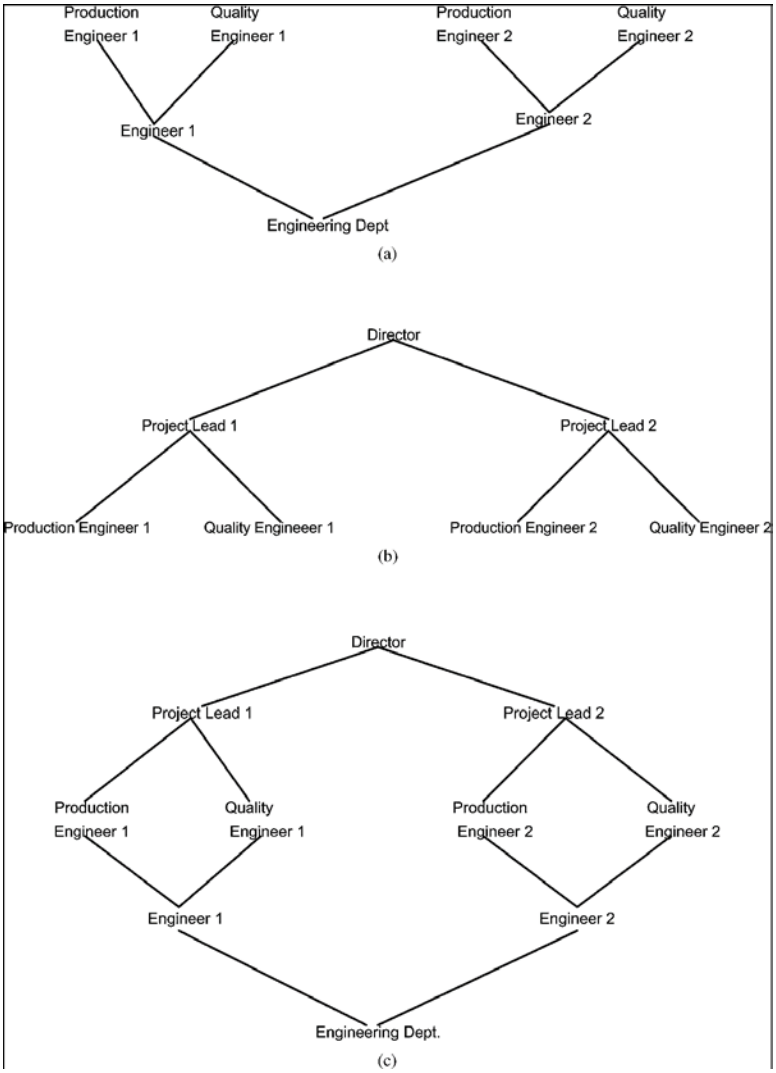
They may take the form of a general partial order or they may be limited to tree structures.

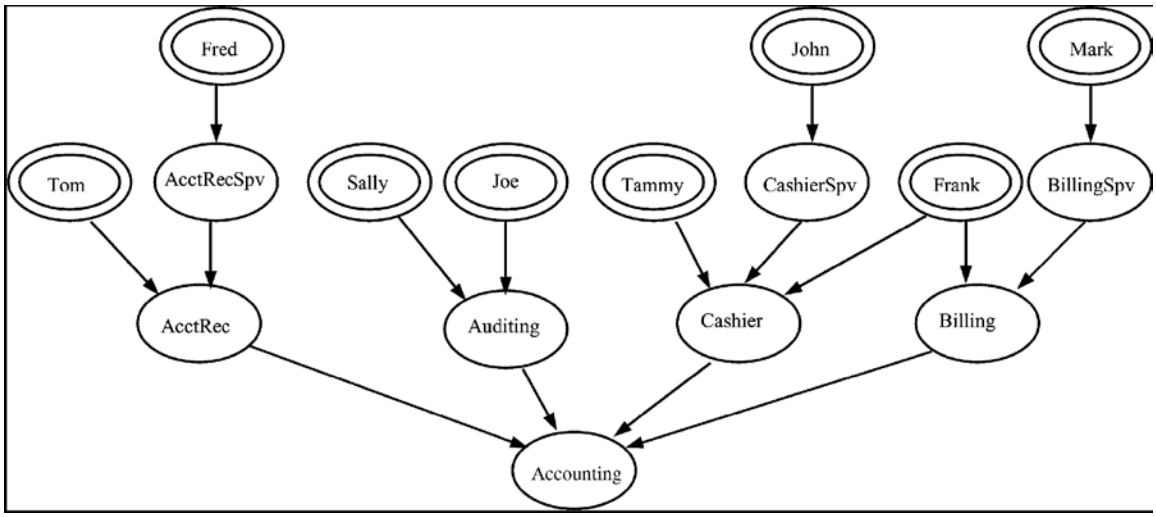
Role hierarchies define an *inheritance* relation among roles.

Role r_1 *inherits* role r_2 if all the privileges of r_2 are also privileges of r_1.

For example, a project lead might inherit all of the privileges of an engineer.

Example role hierarchies:





Accounting Roles [Ferraiolo *et al*]

Static Separation of Duty Constraints

Conflict of interest in a role-based system may arise when a user acquires permissions associated with conflicting roles.

One means of preventing this is through *static separation of duty* (*SOD*), that is, to enforce constraints on the assignment of users to roles.

A common example is that of *mutually disjoint* roles, such as those of purchasing manager and accounts payable manager.

The same person is not permitted to belong to both roles, because this creates a possibility for committing fraud.

Static constraints generally place restrictions on administrative operations that can undermine organizational SOD policies.

Dynamic Separation of Duty Constraints

With *dynamic separation of duties* (*DSD*) each user has different levels of permission at different times, depending on the role being performed.

This ensures that permissions do not persist beyond the time that they are required.

This aspect of least privilege is called *timely revocation of trust*.

DSD allows a user to be authorized for two or more roles that do not create a conflict of interest when acted on independently, but do so when activated simultaneously.

For example, a user may be authorized for both the roles of Cashier and Cashier Supervisor.

The supervisor is allowed to acknowledge corrections to a Cashier's open cash drawer.

If the user acting in the role Cashier attempted to switch to the role Cashier Supervisor, DSD would require the user to drop the Cashier role.

This would force the closure of the cash drawer before assuming the role Cashier Supervisor.

Attribute-Based Access Control (ABAC)

Organizations face *growing diversity* in both types of users and their access needs.

RBAC roles are often *insufficient* for the expression of real-world access control policies.

They also cannot handle *real-time environmental considerations* that may be relevant to access control decisions

> e.g., location of access, time of day, threat level, and client patch level

ABAC can use *arbitrary attributes*, rather than just a person's role, to authorize an individual's access

Example: Doctor responding to mass casualty event in neighboring state can gain access to patient records based on authentication and on attributes such as employee status, specialization, and certifications.

The administrator or owner of an object creates an *access control rule* using attributes of subjects and objects to govern the set of allowable capabilities

Example: *All nurse practitioners in the cardiology department can view the medical records of heart patients*.

ABAC allows definition of *attribute-based policies* that govern access decisions.

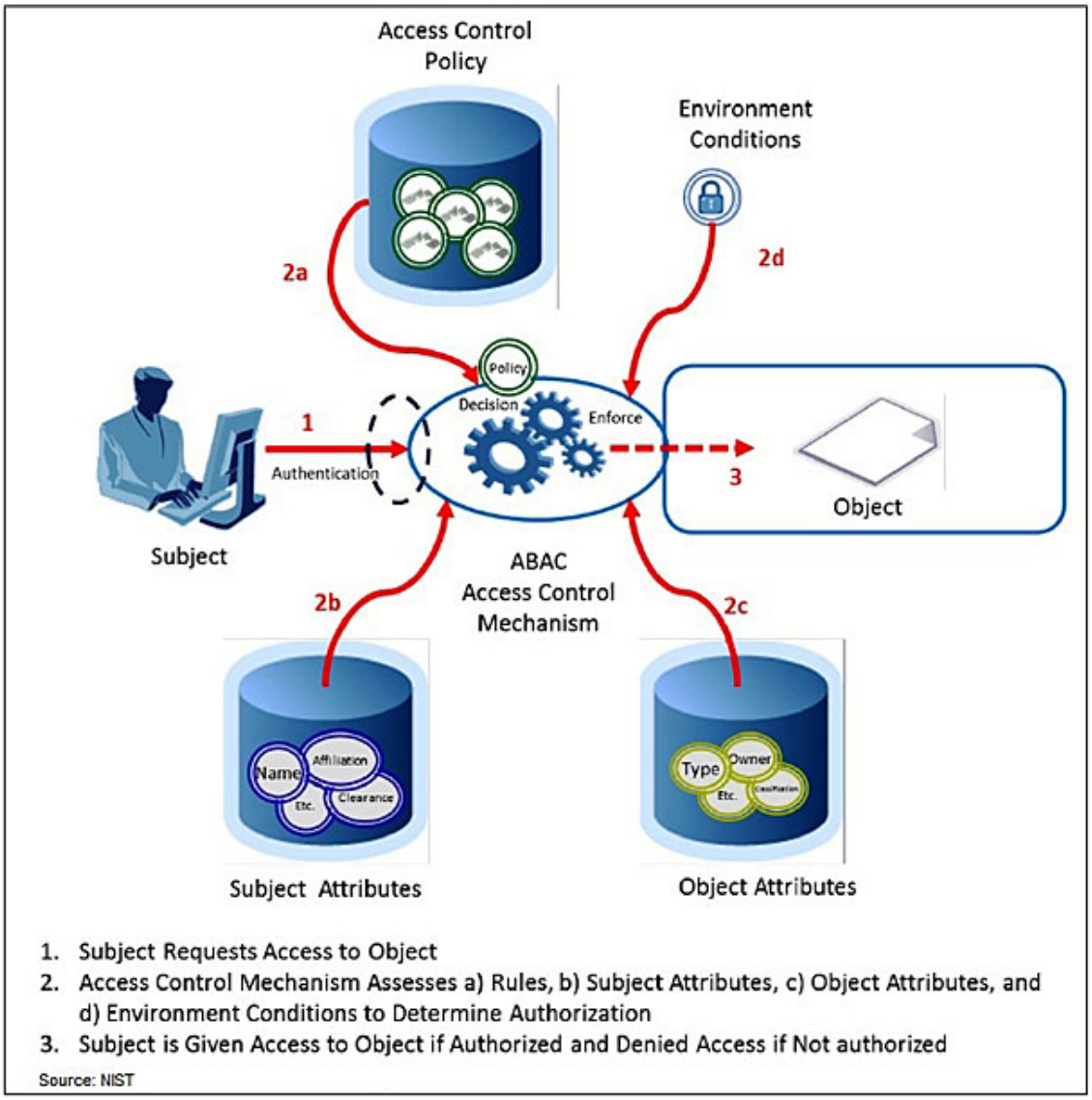
This allows for a *common, centralized* way of expressing policy and computing and enforcing decisions, over the access requests for *diverse systems*.

Under ABAC, access decisions can change between requests simply by *altering attribute values*.

This does *not* require changes to the subject/object relationships defining the underlying rule sets.

ABAC can provide *separation of duties* safeguards.

Example: A car insurance claims adjuster can be permitted to enter data about damage and generate a check, but only his/her supervisor can sign the check.



1. Subject Requests Access to Object
2. Access Control Mechanism Assesses a) Rules, b) Subject Attributes, c) Object Attributes, and d) Environment Conditions to Determine Authorization
3. Subject is Given Access to Object if Authorized and Denied Access if Not authorized

Source: NIST

Usage Control

Traditional DAC, MAC, and RBAC are focused on managing access within a *closed* and *trusted* security domain.

They don't apply well to *open* systems.

Traditional AC models don't assume control during access execution.

Sanhu and Park proposed a new approach, called *Usage Control* (*UCON*).

In UCON access permissions are based on *attributes* and are decided according to three factors: *authorizations*, *obligations*, and *conditions*.

UCON enhances traditional AC models to two respects:

1. *Mutability* of attributes
2. *Continuity* of an access decision

Attribute updates are *side effects* of accesses.

UCON enforce the security policy before, during, and after access execution.

UCON can *dynamically revoke rights* and *terminate resource usage*.

UCON is intended to provide richer, finer, and persistent controls on resources.

The UCON system can be placed on either the *service provider* or the *client side*.

Authorization predicates put *constraints* on a subject's or object's attributes

e.g., an object's type must be ".doc"

Obligations are actions that are required to be performed by the subject before, during, or after use of an object

e.g., a subject must sign a license

Conditions are environment restrictions that must be valid before or during usage

> e.g., an object is available only during working hours

UCON identifies the following components:

- Subjects S
- Subject attributes $ATT(S)$
- Objects O
- Object attributes $ATT(O)$
- Obligations B
- Conditions C
- System states
- State transitions actions
- Attribute update actions

A *subject* (requester) holds rights on the target object, initiates the usage request, and executes permitted actions.

A *subject's attributes* are properties or capabilities that can be used for a usage decision

> e.g., id, name, affiliation, prepaid credits, etc.

Objects are entities that subjects hold rights on.

An *object's attributes* can be used for a usage decision

> e.g., number of previous uses, resource type, security label, etc.

In UCON, *attribute-update actions* are defined by a security policy and may be done before, during, or after the usage.

Mutable attributes change as the result of the usage control process

> e.g, the number of subjects concurrently using an object

Immutable attributes can be modified only by administrative action.

Attributes may be *predefined* or *dynamic*, *local* or *multi-domain*.

Example usage control policy [Lazouski]:

- *Creating policy*:
  $createDoc\_policy(s, doc)$:
  $(s.role = scientist) \rightarrow permit(s, doc, create)$
  $createObj(doc); updateAtt : doc.readTimes = 10$
- *Reading policy*:
  $readDoc\_policy(s, doc)$:
  $(s.role = anonymous) \wedge (doc.readTimes > 0) \rightarrow permit(s, doc, read)$
  $updateAtt : doc.readTimes = doc.readTimes - 1.$