Lecture 3: DAC

RK Shyamasundar

Authorization

- We will use the terms authorization and access control interchangeably
- *Authorization* answers the question "*who* is allowed to *do what*?"
- A first step in the development of an access control system is the identification of the *objects* to be protected, the *subjects* that execute activities and request access to objects, and the *actions* that can be executed on the objects, and that must be controlled

- *Principal* or *user* are used synonymously to subject
- Objects are also referred to as *resources*
- Actions are also called as *operations* or *transactions*
- Access control policies can be grouped into two main classes
  - *Discretionary (DAC)* (authorization-based) policies control access based on the identity of the requestor and on access rules stating what requestors are (or are not) allowed to do
  - *Mandatory (MAC)* policies control access based on mandated regulations determined by a central authority

Discretionary Access Control

- Discretionary policies enforce access control on the basis of the identity of the requestors and explicit access rules that establish who can, or cannot, execute which actions on which resources
- They are called discretionary as users can be given the ability of passing on their privileges to other users, where granting and revocation of privileges is regulated by an administrative policy

The Access Matrix model (AMM)

- The access matrix model provides a framework for describing discretionary access control
- First proposed by *Lampson* for the protection of resources within the context of operating systems, and later refined by *Graham* and *Denning*, the model was subsequently formalized by *Harrison*, *Ruzzo*, and *Ullmann* (HRU model), who developed the access control model proposed by Lampson to the goal of analyzing the complexity of determining an access control policy

The Access Matrix model

- The original model is called access matrix since the authorization state, meaning the authorizations holding at a given time in the system, is represented as a matrix

- The matrix therefore gives an abstract representation of protection systems

The Access Matrix model

- In the access matrix model, the state of the system is defined by a triple (S,O,A), where S is the set of subjects, who can exercise privileges; O is the set of objects, on which privileges can be exercised; and A is the access matrix, where rows correspond to subjects, columns correspond to objects, and entry $A[s,o]$ reports the privileges of s on o
- The type of the objects and the actions executable on them depend on the system

	File1	File2	Program1
Ann	own, read, write	read, write	execute
Bob	read		
Carl		read	execute, read

An example of Access matrix model

The Access Matrix model

- Changes to the state of a system is carried out through *commands* that can execute *primitive* operations on the authorization state, possibly depending on some conditions
- The HRU formalization identified six primitive operations that describe changes to the state of a system
  - adding and removing a subject
  - adding and removing a object
  - adding and removing a privilege

Primitive operations of the HRU model

OPERATION (op)	CONDITIONS	NEW STATE ($Q \vdash_{op} Q'$)
enter r into $A[s,o]$	$s \in S$ $o \in O$	$S' = S$ $O' = O$ $A'[s,o] = A[s,o] \cup \{r\}$ $A'[s_i,o_j] = A[s_i,o_j] \quad \forall (s_i,o_j) \neq (s,o)$
delete r from $A[s,o]$	$s \in S$ $o \in O$	$S' = S$ $O' = O$ $A'[s,o] = A[s,o] \setminus \{r\}$ $A'[s_i,o_j] = A[s_i,o_j] \quad \forall (s_i,o_j) \neq (s,o)$
create subject s'	$s' \notin S$	$S' = S \cup \{s'\}$ $O' = O \cup \{s'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S, o \in O$ $A'[s',o] = \emptyset \quad \forall o \in O'$ $A'[s,s'] = \emptyset \quad \forall s \in S'$
create object o'	$o' \notin O$	$S' = S$ $O' = O \cup \{o'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S, o \in O$ $A'[s,o'] = \emptyset \quad \forall s \in S'$
destroy subject s'	$s' \in S$	$S' = S \setminus \{s'\}$ $O' = O \setminus \{s'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S', o \in O'$
destroy object o'	$o' \in O$ $o' \notin S$	$S' = S$ $O' = O \setminus \{o'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S', o \in O'$

HRU model

- Each command has a conditional part and a body and has the form

command $c(x_1, \ldots, x_k)$
  if r_1 in $A[x_{s1}, x_{o1}]$ and
    r_2 in $A[x_{s2}, x_{o2}]$ and
    .
    .
    r_m in $A[x_{sm}, x_{om}]$
  then op_1
    op_2
    .
    .
    op_n
end.

with $n > 0, m \geq 0$. Here r_1, ..., r_m are actions, op_1, ..., op_n are primitive operations, while s_1, ..., s_m and o_1, ..., o_m are integers between 1 and k. If m=0, the command has no conditional part

For example, the following command creates a file and gives the creating subject ownership privilege on it

command CREATE(creator,file)
create object file
enter Own *into* *A*[creator,file] **end.**

The following commands allow an owner to grant to others, and revoke from others, a privilege to execute an action on her files

command CONFER$_a$(owner,friend,file)
if Own in *A*[owner,file]
then *enter a into* *A*[friend,file] **end.**

command REVOKE$_a$(owner,ex-friend,file)
if Own in *A*[owner,file]
then *delete a from* *A*[ex-friend,file] **end.**

- Let $Q \vdash_{op} Q'$ denote the execution of operation *op* on state *Q*, resulting in state *Q′*. The execution of command $c(a_1, ..., a_k)$ on a system state $Q = (S,O,A)$ causes the *transition* from state *Q* to state *Q′* such that $\exists\ Q_1, \ldots, Q_n$ for which $Q \vdash op^*_1\ Q_1 \vdash op^*_2 ... \vdash op^*_n\ Q_n = Q'$, where $op^*_1 \ldots Op^*_n$ are the primitive operations $op_1 \ldots op_n$ in the body (operational part) of command *c*, in which actual parameters a_i are substituted for each formal parameters x_i, $i := 1, \ldots, k$. If the conditional part of the command is not verified, then the command has no effect and $Q = Q'$

- Although the HRU model does not include any built-in administrative policies, the possibility of defining commands allows their formulation

- Administrative authorizations can be specified by attaching flags to access privileges

- For instance, a *copy flag*, denoted *, attached to a privilege may indicate that the privilege can be transferred to others

Granting of authorizations can then be accomplished by the execution of commands like the one below

command TRANSFER$_a$(subj,friend,file)
if a^* in A[subj,file]
then *enter a into* A[friend,file] **end.**

The ability of specifying commands of this type clearly provides flexibility as different administrative policies can be taken into account by defining appropriate commands

- An alternative administrative flag (called *transfer only* and denoted +) can be supported, which gives the subject the ability of passing on the privilege to others but for which, so doing, the subject looses the privilege
- Such a flexibility introduces an interesting problem referred to as safety, and concerned with the propagation of privileges to subjects in the system
- Intuitively, given a system with initial configuration Q, the safety problem is concerned with determining whether or not a given subject s can ever acquire a given access a on an object o, that is, if there exists a sequence of requests that executed on Q can produce a state Q′ where a appears in a cell A[s,o] that did not have it in Q

- Note that, not all leakages of privileges are bad and subjects may intentionally transfer their privileges to *trustworthy* subjects
- Trustworthy subjects are therefore ignored in the analysis
- It turns out that the safety problem is *undecidable* in general
- It remains instead *decidable* for cases where subjects and objects are finite, and in systems where the body of commands can have at most one operation (while the conditional part can still be arbitrarily complex)

- *Sandhu* proposed the Typed Access Matrix (TAM) model
- TAM extends HRU with strong typing: each subject and object has a type; the type is associated with the subjects/objects when they are created and thereafter does not change
- Safety results decidable in polynomial time for cases where the system is monotonic (privileges cannot be deleted), commands are limited to three parameters, and there are no cyclic creates
- Safety remains undecidable otherwise

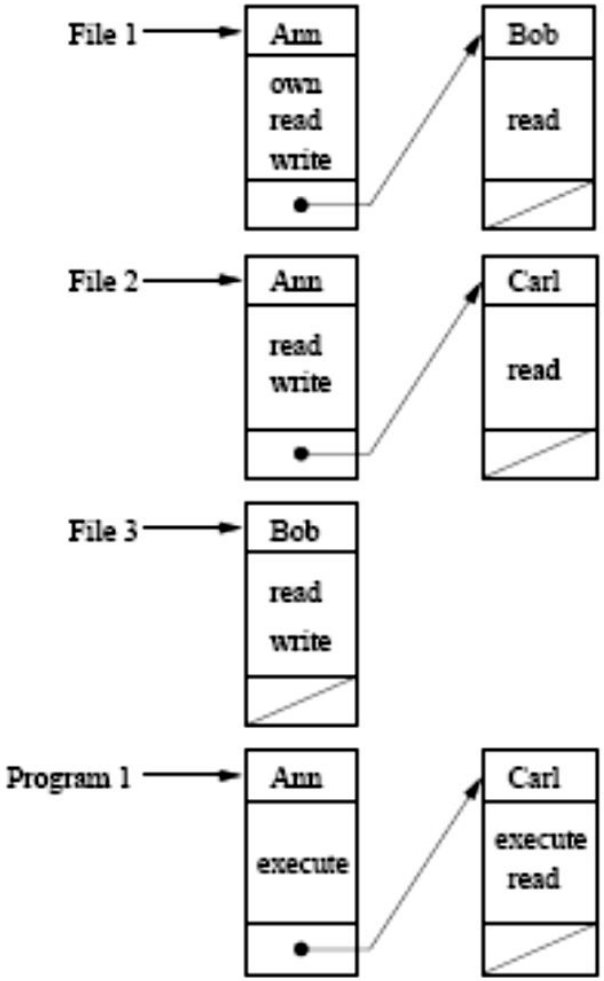
Implementation of the AMM

- Although the matrix represents a good conceptualization of authorizations, it is not appropriate for implementation
- In a general system, the access matrix will be usually enormous in size and sparse (most of its cells are likely to be empty)
- Storing the matrix as a two-dimensional array is therefore a waste of memory space
- There are three approaches to implementing the access matrix in a practical way

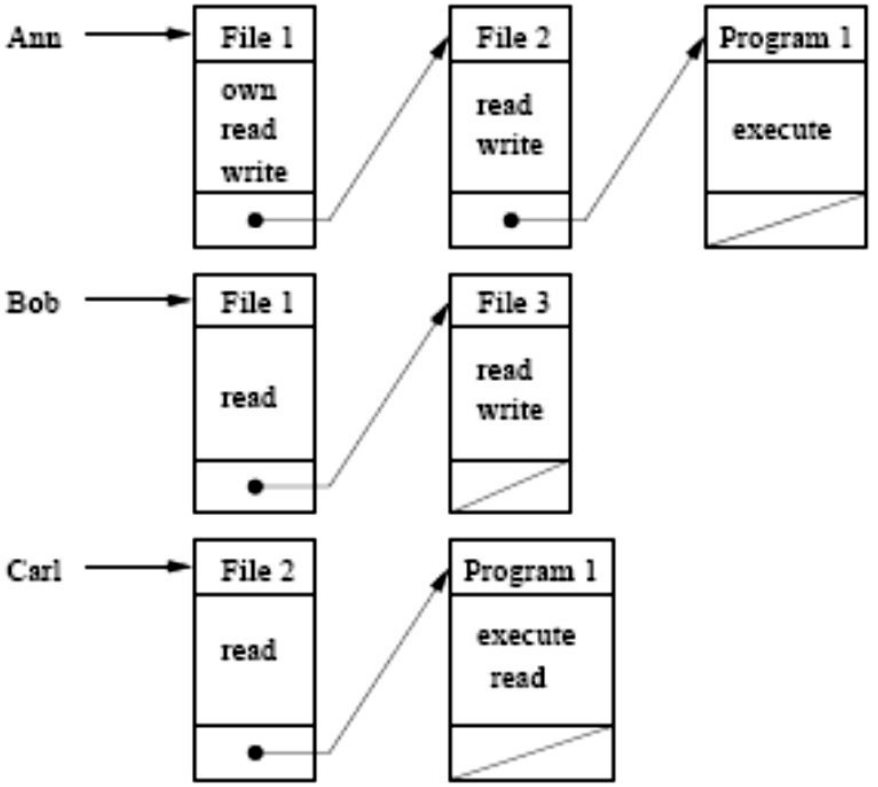
- *Authorization Table*: Non empty entries of the matrix are reported in a table with three columns, corresponding to subjects, actions, and objects, respectively. Each tuple in the table corresponds to an authorization
- *Access Control List (ACL)*: The matrix is stored by column. Each object is associated with a list indicating, for each subject, the actions that the subject can exercise on the object
- *Capability*: The matrix is stored by row. Each user has associated a list, called capability list, indicating, for each object, the accesses that the user is allowed to exercise on the object

USER	ACCESS MODE	OBJECT
Ann	own	File 1
Ann	read	File 1
Ann	write	File 1
Ann	read	File 2
Ann	write	File 2
Ann	execute	Program 1
Bob	read	File 1
Bob	read	File 3
Bob	write	File 3
Carl	read	File 2
Carl	execute	Program 1
Carl	read	Program 1

Example Authorization table



Example ACL



Example capabilities

- Capabilities and ACLs present advantages and disadvantages with respect to authorization control and management
  - In particular, with ACLs it is immediate to check the authorizations holding on an object, while retrieving all the authorizations of a subject requires the examination of the ACLs for all the objects
  - Analogously, with capabilities, it is immediate to determine the privileges of a subject, while retrieving all the accesses executable on an object requires the examination of all the different capabilities

- These aspects affect the efficiency of authorization revocation upon deletion of either subjects or objects

- In a system supporting capabilities, it is sufficient for a subject to present the appropriate capability to gain access to an object
- This represents an advantage in distributed systems since it permits to avoid repeated authentication of a subject: a user can be authenticated at a host, acquire the appropriate capabilities and present them to obtain accesses at the various servers of the system
- However, capabilities are vulnerable to *forgery* (they can be copied and reused by an unauthorized third party)
- Another problem in the use of capability is the enforcement of revocation, meaning invalidation of capabilities that have been released

- Modern operating systems typically take the ACL-based approach
- Some systems implement an abbreviated form of ACL by restricting the assignment of authorizations to a limited number (usually one or two) of named groups of users, while individual authorizations are not allowed
- The advantage of this is that ACLs can be efficiently represented as small bit-vectors
- For instance, in the popular Unix operating system, each user in the system belongs to exactly one group and each file has an owner (generally the user who created it), and is associated with a group (usually the group of its owner)

- Authorizations for each file can be specified for the file's owner, for the group to which the file belongs, and for "the rest of the world" (meaning all the remaining users)
- No explicit reference to users or groups is allowed
- Authorizations are represented by associating with each object an access control list of 9 bits: bits 1 through 3 reflect the privileges of the file's owner, bits 4 through 6 those of the user group to which the file belongs, and bits 7 through 9 those of all the other users
- The three bits correspond to the read (r), write (w), and execute (x) privilege, respectively

Vulnerabilities of DAC

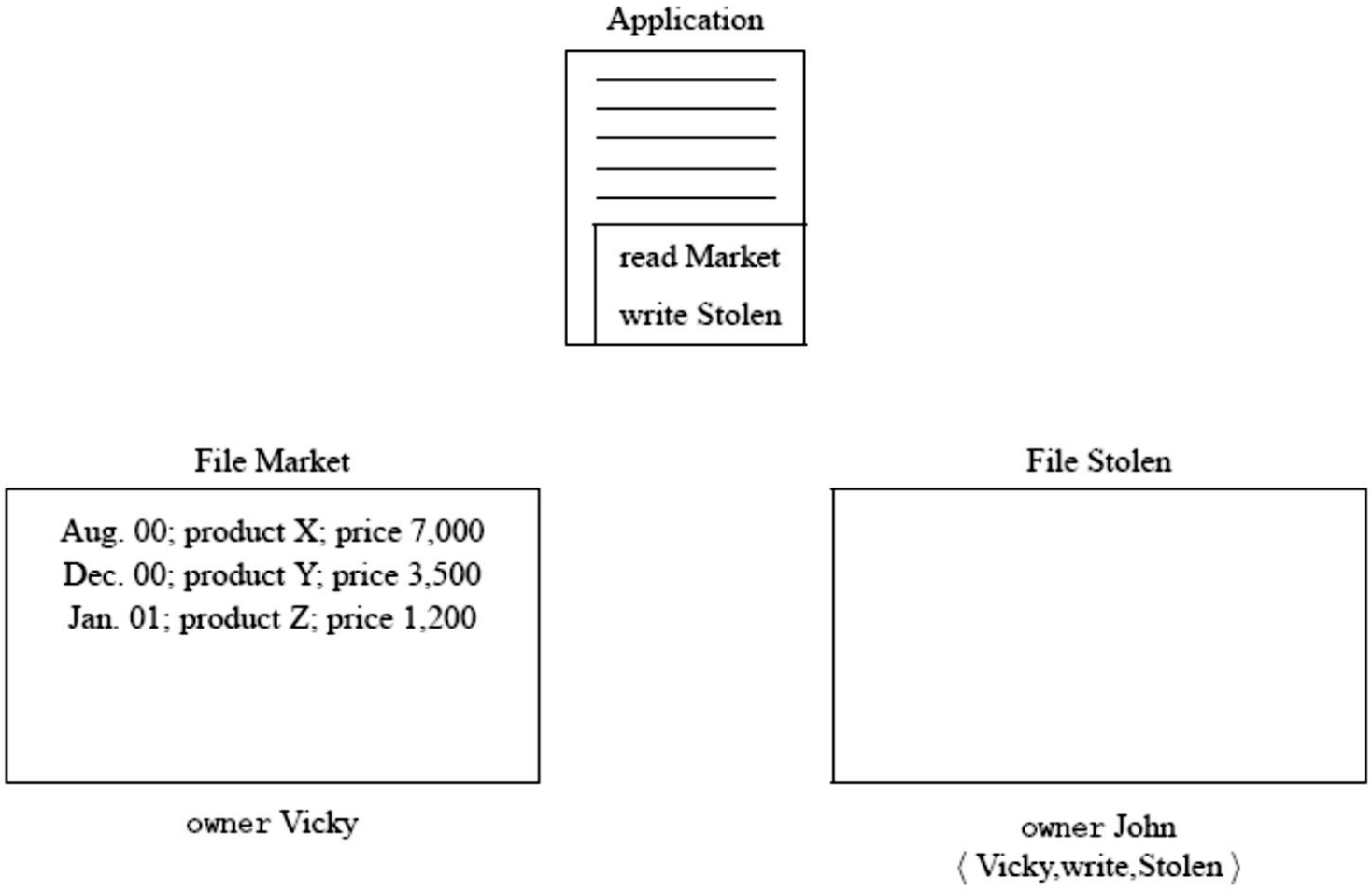
- In defining the basic concepts of discretionary policies, we have referred to access requests on objects submitted by users, which are then checked against the users' authorizations

- Although it is true that each request is originated because of some user's actions, a more precise examination of the access control problem shows the utility of separating *users* from *subjects*

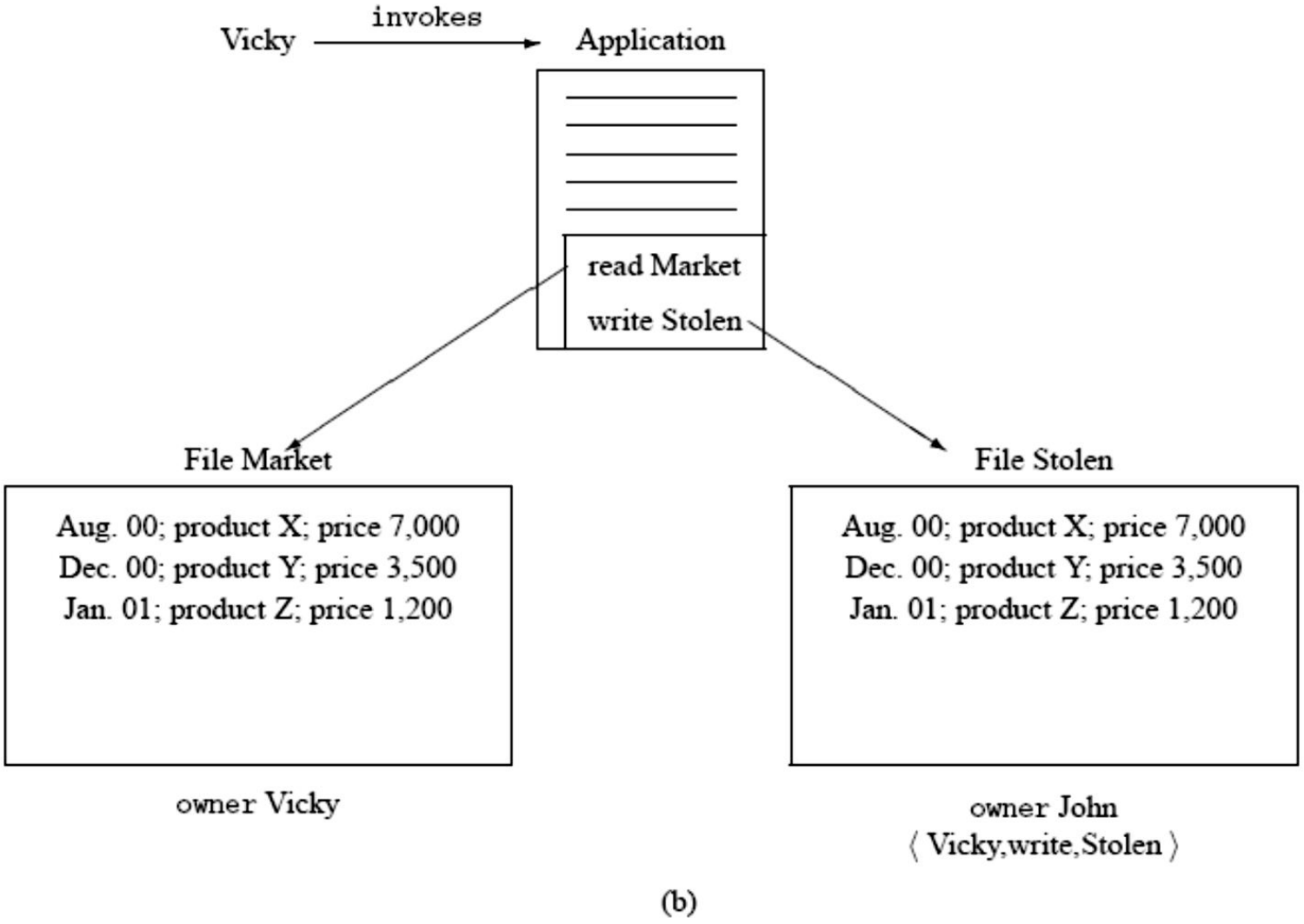
- Users are *passive entities* for whom authorizations can be specified and who can connect to the system
- Once connected to the system, users originate processes (*subjects*) that execute on their behalf and, accordingly, submit requests to the system
- Discretionary policies ignore this distinction and evaluate all requests submitted by a process running on behalf of some user against the authorizations of the user
- This aspect makes discretionary policies vulnerable from processes executing *malicious programs* exploiting the authorizations of the user on behalf of whom they are executing

- In particular, the access control system can be bypassed by Trojan Horses embedded in programs
- A Trojan Horse is a computer program with an apparently or actually useful function, which contains additional hidden functions that exploit the legitimate authorizations of the invoking process
- A Trojan Horse can improperly use any authorizations of the invoking user, for example, it could even delete all files of the user

- This vulnerability to Trojan Horses, together with the fact that discretionary policies do not enforce any control on the flow of information once this information is acquired by a process, makes it possible for processes to leak information to users not allowed to read it
- All this can happen without the cognizance of the data administrator/owner, and despite the fact that each single access request is controlled against the authorizations


Application
read Market
write Stolen
File Market
Aug. 00; product X; price 7,000
Dec. 00; product Y; price 3,500
Jan. 01; product Z; price 1,200
owner Vicky
File Stolen
owner John
⟨ Vicky,write,Stolen ⟩

(a)


Vicky
invokes
Application
read Market
write Stolen
File Market
Aug. 00; product X; price 7,000
Dec. 00; product Y; price 3,500
Jan. 01; product Z; price 1,200
owner Vicky
File Stolen
Aug. 00; product X; price 7,000
Dec. 00; product Y; price 3,500
Jan. 01; product Z; price 1,200
owner John
⟨ Vicky,write,Stolen ⟩

(b)

- We object that there is little point in defending against Trojan Horses leaking information flow: such an information flow could have happened anyway, by having Vicky explicitly tell this information to John, possibly even off-line, without the use of the computer system
- This is where the distinction between users and subjects operating on their behalf comes in
- While users are trusted to obey the access restrictions, subjects operating on their behalf are not

- Processes run programs which, unless properly certified, cannot be trusted for the operations they execute
- For this reason, restrictions should be enforced on the operations that processes themselves can execute
- In particular, protection against Trojan Horses leaking information to unauthorized users requires controlling the flows of information within processes execution and possibly restricting them
- Mandatory policies provide a way to enforce information flow control through the use of labels

Principles

Limits of firewalls

- Once a host on an intranet behind a firewall has been compromised, the attacker can communicate with this machine by tunnelling traffic over an open protocol (e.g., HTTPS) and launch further intrusions unhindered from there.
- Little protection is provided against insider attacks.
- Centrally administered rigid firewall policies severely disrupt the deployment of new services. The ability to “tunnel” new services through existing firewalls with fixed policies has become a major protocol design criterion. Many new protocols (e.g., SOAP) are for this reason designed to resemble HTTP, which typical firewall configurations will allow to pass.
- Firewalls can be seen as a compromise solution for environments, where the central administration of the network configuration of each host on an intranet is not feasible. Much of firewall protection can be obtained by simply deactivating the relevant network services on end machines directly.

Enforcement (1)

- Monitoring: Because attacks (by definition) involve execution, a second means of defense can be to monitor a set of interfaces and halt execution before any damage is done using operations those interfaces provide. Three elements comprise this defense:
- a security policy, which prescribes acceptable sequences of operations from some set of interfaces;
- a reference monitor, which is a program that is guaranteed to receive control whenever any operation named in the policy is requested, and
- a means by which the reference monitor can block further execution that does not comply with the policy.

Enforcement (2)

- Principle of Complete Mediation: The reference monitor intercepts every access to every object
- Principle of Least Privilege. A principal should be only accorded the minimum privileges it needs to accomplish its task
  - impossible to implement if the same privilege suffices for multiple different objects or operation
- **Principle of Separation of Privilege. Different accesses should require different privileges.**
- Principle of Failsafe Defaults. The presence of privileges rather than the absence of prohibitions should be the basis for determining whether an access is allowed to proceed

Recovery

- Attacks whose effects are reversible could be allowed to run their course, if a recovery mechanism were available afterwards to undo any damage
- Confidentiality Violations: no longer confidential once revealed
- Integrity Violations: Changes to internal system state are usually reversible, so recovery can be used to defend against attacks whose sole effect is to change that state.
- **Availability Violations:** For a system not involved in sensing or controlling the physical environment, recovery from availability violations could be feasible: *evict the attacker and resume normal processing*

Information Flows