Discretionary Access Control (DAC)

- No precise definition
- Widely used in modern operating systems
- In most implementations it has the notion of owner of an object
- The owner controls other users' accesses to the object
- Allows access rights to be propagated to other subjects

Problems with DAC in OS

- DAC cannot protect against
  - Trojan horse
  - Malware
  - Software bugs
  - Malicious local users
- It cannot control information flow

The Trojan Horse

Diagram:
- Process P
  - read O1
  - write O2
- O1
  - (alice,r,O1)
- O2
  - (alice,r,O2), (alice,w,O2), (mallory,r,O2)
Mandatory Access Control

- Mandatory access control (MAC) restricts the access of subjects to objects based on a system-wide policy.
- The system security policy (as set by the administrator) entirely determines the access rights granted.
  - denying users full control over the access to resources that they create.

The Need for MAC

- Host compromise by network-based attacks is the root cause of many serious security problems.
  - Worm, Botnet, DDoS, Phishing, Spamming
- Why hosts can be easily compromised:
  - Programs contain exploitable bugs
  - The discretionary access control mechanism in the operating systems was not designed by taking into account buggy software.

MAC

- MAC specifies the access that subjects have to objects based on subjects and objects classification.
- This type of security has also been referred to as multilevel security.
- Database systems that satisfy multilevel security properties are called multilevel secure database management systems (MLS/DBMSs).
- Many of the MLS/DBMSs have been designed based on the Bell and LaPadula (BLP) model.

A Characterization of the Difference between DAC and MAC

- Discretionary Access Control Models (DAC)
  - Definition [Bishop p.53] If an individual user can set an access control mechanism to allow or deny access to an object, that mechanism is a discretionary access control (DAC), also called an identity-based access control (IBAC).
- Mandatory Access Control Models (MAC)
  - Definition [Bishop p.53] When a system mechanism controls access to an object and an individual user cannot alter that access, the control is a mandatory access control (MAC), occasionally called a rule-based access control.]
Bell and LaPadula (BLP) Model

Elements of the model:
- objects - passive entities containing information to be protected
- subjects: active entities requiring accesses to objects (users, processes)
- access modes: types of operations performed by subjects on objects
  - read: reading operation
  - append: modification operation
  - write: both reading and modification

BLP Model

- Subjects are assigned clearance levels and they can operate at a level up to and including their clearance levels
- Objects are assigned sensitivity levels
- The clearance levels as well as the sensitivity levels are called access classes

BLP Model - access classes

- An access class consists of two components: a security level and a category set
- The security level is an element from a totally ordered set - example
  
  \{Top Secret (TS), Secret (S), Confidential (C), Unclassified (U)\}

  where \(TS > S > C > U\)
- The category set is a set of elements, dependent from the application area in which data are to be used - example
  
  \{Army, Navy, Air Force, Nuclear\}

Access class \(c_i = (L_i, SC_i)\) dominates access class \(c_k = (L_k, SC_k)\), denoted as \(c_i \geq c_k\), if both the following conditions hold:
- \(L_i \geq L_k\) The security level of \(c_i\) is greater or equal to the security level of \(c_k\)
- \(SC_i \supseteq SC_k\) The category set of \(c_i\) includes the category set of \(c_k\)
• If $L_i > L_k$ and $SC_i \supset SC_k$, we say that $c_i$ strictly dominates $c_k$.
• $c_i$ and $c_k$ are said to be incomparable (denoted as $c_i < > c_k$) if neither $c_i \geq c_k$ nor $c_k \geq c_i$ holds.

Access classes:
- $c_1 = (TS, \{Nuclear, Army\})$
- $c_2 = (TS, \{Nuclear\})$
- $c_3 = (C, \{Army\})$
  - $c_1 > c_2$ (TS > C and $\{Army\} \subset \{Nuclear, Army\}$)
  - $c_1 > c_3$
  - $c_2 < > c_3$

The state of the system is described by the pair $(A, L)$, where:
- $A$ is the set of current accesses: triples of the form $(s, o, m)$ denoting that subject $s$ is exercising access $m$ on object $o$ - example (Bob, o1, read).
- $L$ is the level function: it associates with each element in the system its access class.

Let $O$ be the set of objects, $S$ the set of subjects, and $C$ the set of access classes.

$L: O \cup S \rightarrow C$.

Simple security property (no-read-up): a given state $(A, L)$ satisfies the simple security property if for each element $a = (s, o, m) \in A$ one of the following condition holds:
1. $m = append$
2. $(m = read$ or $m = write)$ and $L(s) \geq L(o)$

Example: a subject with access class $(C, \{Army\})$ is not allowed to read objects with access classes $(C, \{Navy, Air Force\})$ or $(U, \{Air Force\})$.
BLP Model - Axioms

• The simple security property prevents subjects from reading data with access classes dominating or incomparable with respect with the subject access class.

• It therefore ensures that subjects have access only to information for which they have the necessary access class.

BLP Model - Axioms

• Star (*) property (no-write-down)

  a given state \((A, L)\) satisfies the *-property if for each element \(a = (s, o, m) \in A\) one of the following condition holds:

  1. \(m = \text{read}\)
  2. \(m = \text{append} \) and \(L(o) > L(s)\)
  3. \(m = \text{write} \) and \(L(o) = L(s)\)

• Example: a subject with access class \((C, \{\text{Army, Nuclear}\})\) is not allowed to append data into objects with access class \((U, \{\text{Army, Nuclear}\})\).

BLP Model - Axioms

• The *-property has been defined to prevent information flow into objects with lower-level access classes or incomparable classes.

• For a system to be secure both properties must be verified by any system state.

BLP Model

• Summary of access rules:

  – Simple security property: A subject has read access to an object if its access class dominates the access class of the object.

  – *-Property: A subject has append access to an object if the subject's access class is dominated by that of the object.
An Example of Application
The DG/Unix B2 System

- B2 is an evaluation class for secure systems defined as part of the Trusted Computer System Evaluation Criteria (TCSEC), known also as the Orange Book.
- DG/Unix provides mandatory access controls
  - MAC label identifies security level
  - Default labels, but can define others
- Initially
  - Processes (users) assigned MAC label of parent
    - Initial label assigned to user, kept in Authorization and Authentication database
  - Objects assigned MAC labels at creation
    - Explicit labels stored as part of attributes
    - Implicit labels determined from parent directory

Directory Problem

- Process \( p \) at access class MAC_A tries to create file /tmp/x
- /tmp/x exists but has access class MAC_B
  - Assume MAC_B \( \geq \) MAC_A (MAC_B dominates MAC_A)
- Create fails
  - Now \( p \) knows a file named x with a higher label exists
- Fix: only programs with same MAC label as the directory can create files in the directory
  - This solution is too restrictive

Multilevel Directory

- Directory with a set of subdirectories, one per label – referred to as polyinstation of the directory
  - Not normally visible to user
  - \( p \) creating /tmp/x actually creates /tmp/d/x where d is directory corresponding to MAC_A
  - All \( p \)'s references to /tmp go to /tmp/d
- The directory problem illustrates an important point:
  Sometimes it is not sufficient to hide the contents of objects. Also their existence must be hidden.

Other Examples of MAC for Linux/Unix

- MAC: a system-wide security policy restricts the access rights of subjects
- Existing MACs for Linux / Unix:
  - SELinux, from NSA
  - AppArmor (SubDomain), from Novell Inc.
  - Systrace, from University of Michigan
  - LOMAC, from NAI Labs
  - …
**SELinux**

- Developed by National Security Agency (NSA) and Secure Computing Corporation (SCC) to promote MAC technologies
- MAC functionality is provided through the FLASK architecture
- Can be applied to Unix-like operating systems, such as Linux, Solaris, and FreeBSD
- Available as a patch for 2.4 kernels
- Integrated into 2.6 kernels
- Available also with SUSE Linux

**FLASK**

- **Flux Advanced Security Kernel**
- General MAC architecture
- It defines what should be available and not how it should be implemented
- It supports flexible security policies
  - Policies implemented as part of the prototype
  - Multi-level security
  - Type enforcement
  - Identity-based access control
  - Role-based access control
- It separates policies from enforcement

---

**Flask in a Diagram**

![Diagram of FLASK architecture](image-url)
**Flask Overview**

- **Security server** – it provides functions to the object managers for retrieving:
  - Access – permission between two entities
  - Label – specifies security attributes of an object
  - Polyr instantiation – which member of a set of resources should be accessed for a particular request
- **Object manager** – It must define:
  - a mechanism to assign labels to their objects
  - a control policy, which specifies how security decisions are actually used and enforced
  - handling routines that are called when policy changes
- **Access control cache** – it caches decisions to minimize performance overhead

**Object Labeling**

- All objects controlled by the security policy are labeled with a set of security attributes, referred to as the *security context*
- Flask provides two data types for labeling objects
  - Security contexts – variable length strings that can be interpreted by any application or user that understands the security policy, can contain whatever is needed by the security policy and is therefore flexible
  - SID – fixed size values used as references to security contexts, created for efficiency reasons (cheaper to pass around), security server maintains SID mappings

**Domain-type Enforcement in SELinux**

- Each object (file) is labeled by a **type**
- Each subject (process) is associated with a **domain**
- The domain determines what accesses to types the process has
- The type of a new file is based on the domain of the creating process and the parent directory
- Configuration files specify how domains are allowed to access types and to interact with other domains.
- Linux and SELinux access controls are orthogonal
  - each mechanism uses its own access control attributes
  - two separate access checks; both must pass

**Access Control**

- Security decisions first go through DAC and then MAC
What is a Type?

- A type is an unambiguous identifier
  - created by the policy writer
  - applied to all subjects and objects and for access decisions
- Types group subjects and objects
  - signifies security equivalence
  - everything with the same type has the same access
  - policies have as few or as many types as needed

RBAC in SELinux

- Why Roles?: system processes can be separated from ordinary users processes
- Users are authorised for roles and roles are authorised for domains and types
- Each process has an associated role
- Configuration files specify the set of domains that may be entered by each role
- Each user role has an initial domain that is associated with the user’s login shell.
- As users execute programs, transitions to other domains may, according to the policy configuration, automatically occur to support changes in privilege.

Context

- SELinux assigns subjects and objects a security context:
  - user identifier
  - role identifier
  - type identifier
  - [mls identifier]
- Standard rwx permissions for user:group
  -rw------- root root anaconda-ks.cfg
- In SELinux
  -rw------- root root system_u:object_r:admin_home_t:s0 anaconda-ks.cfg
- Command chcon allows one to change the context of a file or directory

An Example of a Policy Statement

allow passwd_t shadow_t : file
- It allows processes with passwd_t domain type read, write, and create access to files with shadow_t type
- Purpose: passwd program runs with passwd_t type, allowing it to change shadow password file (/etc/shadow)
- Shadow password file attributes:
  -r-------- root root system_u:object_r:shadow_t /etc/shadow
SELinux in Practice

- **Strict policy**
  - A system where everything is denied by default
  - Minimal privilege's for every daemon
  - Separate user domains for programs like GPG, X, ssh, etc
  - Difficult to enforce in general purpose operating systems
  - Default in Fedora Core 2
- **Targeted policy**
  - System where everything is allowed, use deny rules.
  - Only restrict certain daemon programs
  - Default in Fedora Core 3
  - No protection for client programs

SubDomain

- The subdomain mechanism provides a sufficiently fine-grained mechanism
- It tries to achieve least privilege for programs
- Administrators specify the domain of activities the program can perform
  - Files, Operations

Examples

```bash
for (str = /
    /etc/htc/hst w,
    /usr/lib/htcread
    /var/lib/htcwrite w,
    /etc/htc/hst w,
    /usr/lib/htcread
    /var/lib/htcwrite w,
    /etc/htc/hst w,
    /usr/lib/htcread
    /var/lib/htcwrite w;
    )
```

Sub-process confinement

- Scriptable servers, Loadable modules, Plug-ins
- Provide a system call: change_hat()
- Like sandboxing
- The developer should make appropriate calls
Compatibility

- Who write the profile?
  - Vendors
  - Administrators
- Which programs need to be confined?
  - Policy
  - All programs
  - All listed user-ids
  - All root programs
  - Only specified programs
  - All network programs
- How to generate the profile?
  - Run, log, grant
  - Tool: dep, strace

AppArmor

- Implemented the SubDomain model
- Ships with SUSE Linux
- Usability
  - Interactive / Automatic tools
  - Path pattern matching
  - Good policy syntax

Usability of MAC Systems

- Only low-level mechanisms are provided
  - The security is achieved through proper policy configuration, which is extremely difficult to do even for security experts
- Policy specification is complicated and difficult to understand
  - SELinux: 29 classes of objects, hundreds of operations, thousands of policy rules
  - AppArmor: need a profile for each individual program

Covert Channels

- A covert channel allows a transfer of information that violates the MLS policy
- It is an information flow which is not controlled by a security mechanism
- Covert channels can be classified into two broad categories: timing and storage channels
- The difference between the two is that in timing channels the information is conveyed by the timing of events or processes, whereas storage channels do not require any temporal synchronization; it is conveyed by accessing system information
Covert Channels - example

- A well-known covert channel is based on the exploitation of the concurrency control
- Consider two processes \( P_l \) and \( P_h \) of access class low and high respectively; consider a data item \( d_1 \) classified at class low; assume that those all the only processes running
- Suppose that \( P_h \) requires a read lock on \( d_1 \); the lock is granted because no other process is running

Covert Channels - example

- Suppose now that process \( P_l \) wishes to write the same data item; it thus requires a write lock on \( d_1 \)
- Since process \( P_h \) holds a read lock on \( d_1 \), process \( P_l \) is forced to wait until \( P_h \) releases the lock on \( d_1 \)
- By selectively issuing requests to read low data, process \( P_h \) can modulate the delay experienced by process \( P_l \)

Covert Channels - example

- Since \( P_h \) has full access to high data, this delay can be used by \( P_h \) to transfer high information to process \( P_l \)
- Thus a timing channel is established between the two processes