5.2. The Bell-LaPadula Model

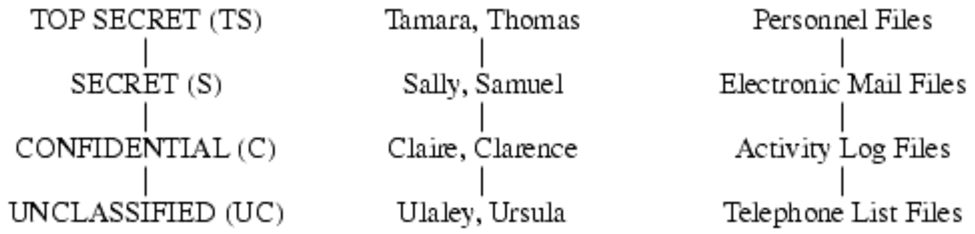
The Bell-LaPadula Model [67, 68] corresponds to military-style classifications. It has influenced the development of many other models and indeed much of the development of computer security technologies.[1]

[1] The terminology in this section follows that of the unified exposition of the Bell-LaPadula Model [68].

5.2.1. Informal Description

The simplest type of confidentiality classification is a set of **security clearances** arranged in a linear (total) ordering (see Figure 5-1). These clearances represent sensitivity levels. The higher the security clearance, the more sensitive the information (and the greater the need to keep it confidential). A subject has a **security clearance**. In the figure, Claire's security clearance is C (for CONFIDENTIAL), and Thomas' is TS (for TOP SECRET). An object has a **security classification**; the security classification of the electronic mail files is S (for SECRET), and that of the telephone list files is UC (for UNCLASSIFIED). (When we refer to both subject clearances and object classifications, we use the term "classification.") The goal of the Bell-LaPadula security model is to prevent read access to objects at a security classification higher than the subject's clearance.

Figure 5-1. At the left is the basic confidentiality classification system. The four security levels are arranged with the most sensitive at the top and the least sensitive at the bottom. In the middle are individuals grouped by their security clearances, and at the right is a set of documents grouped by their security levels.



The Bell-LaPadula security model combines mandatory and discretionary access controls. In what follows, "**S** has discretionary read (write) access to **O**" means that the access control matrix entry for **S** and **O** corresponding to the discretionary access control component contains a read (write) right. In other words, were the mandatory controls not present, **S** would be able to read (write) **O**.

Let **L**(**S**) = $\mathbf{l_s}$ be the security clearance of subject **S**, and let **L**(**O**) = $\mathbf{l_o}$ be the security classification of object **O**. For all security classifications $\mathbf{l_i}$, **i** = 0, ..., **k** – 1, $\mathbf{l_i} < \mathbf{l_{i+1}}$.

- **Simple Security Condition, Preliminary Version**: **S** can read **O** if and only if $\mathbf{l_o} \le \mathbf{l_s}$ and **S** has discretionary read access to **O**.

In Figure 5-1, for example, Claire and Clarence cannot read personnel files, but Tamara and Sally can read the activity log files (and, in fact, Tamara can read any of the files, given her clearance), assuming that the discretionary access controls allow it.

Should Tamara decide to copy the contents of the personnel files into the activity log files and set the discretionary access permissions appropriately, Claire could then read the personnel files. Thus, for all practical purposes, Claire could read the files at a higher level of security. A second property prevents this:

- ***-Property (Star Property), Preliminary Version**: **S** can write **O** if and only if $\mathbf{l_s} \le \mathbf{l_o}$ and **S** has discretionary write access to **O**.

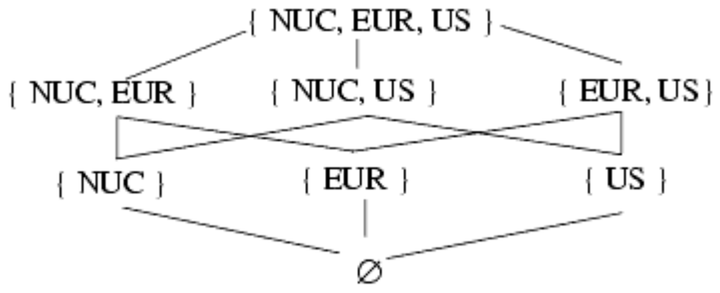
Because the activity log files are classified C and Tamara has a clearance of TS, she cannot write to the activity log files.

Define a **secure system** as one in which both the simple security condition, preliminary version, and the *-property, preliminary version, hold. A straightforward induction establishes the following theorem.

> **Theorem 5–1. Basic Security Theorem, Preliminary Version**: Let Σ be a system with a secure initial state σ_0, and let **T** be a set of state transformations. If every element of **T** preserves the simple security condition, preliminary version, and the *-property, preliminary version, then every state σ_i, **i** ≥ 0, is secure.

Expand the model by adding a set of **categories** to each security classification. Each category describes a kind of information. Objects placed in multiple categories have the kinds of information in all of those categories. These categories arise from the “need to know” principle, which states that no subject should be able to read objects unless reading them is necessary for that subject to perform its functions. The sets of categories to which a person may have access is simply the power set of the set of categories. For example, if the categories are NUC, EUR, and US, someone can have access to any of the following sets of categories: Ø (none), { NUC }, { EUR }, { US }, { NUC, EUR }, {NUC, US }, { EUR, US }, and { NUC, EUR, US }. These sets of categories form a lattice under the operation ⊆ (subset of); see Figure 5-2. (Chapter 30, “Lattices,” discusses the mathematical nature of lattices.)

Figure 5-2. Lattice generated by the categories NUC, EUR, and US. The lines represent the ordering relation induced by ⊆.



Each security level and category form a **security level**.[2] As before, we say that subjects **have clearance at** (or **are cleared into**, or **are in**) a security level and that objects **are at the level of** (or **are in**) a security level. For example, William may be cleared into the level (SECRET, { EUR }) and George into the level (TOP SECRET, { NUC, US }). A document may be classified as (CONFIDENTIAL, {EUR }).

[2] There is less than full agreement on this terminology. Some call security levels “compartments.” However, others use this term as a synonym for “categories.” We follow the terminology of the unified exposition [68].

Security levels change access. Because categories are based on a “need to know,” someone with access to the category set { NUC, US } presumably has no need to access items in the category EUR. Hence, read access should be denied, even if the security clearance of the subject is higher than the security classification of the object. But if the desired object is in any of the security levels Ø, { NUC }, { US }, or { NUC, US } and the subject's security clearance is no less than the document's security classification, access should be granted because the subject is cleared into the same category set as the object.

This suggests a new relation for capturing the combination of security classification and category set. Define the relation **dom** (dominates) as follows.

Definition 5–1. The security level (**L, C**) **dominates** the security level (**L', C'**) if and only if **L'** ≤ **L** and **C'** ⊆ **C**.

We write (**L, C**) **¬dom** (**L', C'**) when (**L, C**) **dom** (**L', C'**) is false. This relation also induces a lattice on the set of security levels [267].

EXAMPLE: George is cleared into security level (SECRET, { NUC, EUR}), DocA is classified as (CONFIDENTIAL, { NUC }), DocB is classified as (SECRET, { EUR, US}), and DocC is classified as (SECRET, { EUR }). Then:

George **dom** DocA as CONFIDENTIAL ≤ SECRET and { NUC } ⊆ { NUC, EUR }

George **¬dom** DocB as { EUR, US } ⊄ { NUC, EUR }

George **dom** DocC as SECRET ≤ SECRET and { EUR } ⊆ { NUC, EUR }

Let **C**(**S**) be the category set of subject **S**, and let **C**(**O**) be the category set of object **O**. The simple security condition, preliminary version, is modified in the obvious way:

- **Simple Security Condition**: **S** can read **O** if and only if **S dom O** and **S** has discretionary read access to **O**.

In the example above, George can read DocA and DocC but not DocB (again, assuming that the discretionary access controls allow such access).

Suppose Paul is cleared into security level (SECRET, { EUR, US, NUC }) and has discretionary read access to DocB. Paul can read DocB; were he to copy its contents to DocA and set its access permissions accordingly, George could then read DocB. The modified *-property prevents this:

- ***-Property**: **S** can write to **O** if and only if **O dom S** and **S** has discretionary write access to **O**.

Because DocA **dom** Paul is false (because **C**(Paul) $\not\subseteq$ **C**(DocA)), Paul cannot write to DocA.

The simple security condition is often described as "no reads up" and the *-property as "no writes down."

Redefine a **secure system** as one in which both the simple security property and the *-property hold. The analogue to the Basic Security Theorem, preliminary version, can also be established by induction.

Theorem 5–2. Basic Security Theorem: Let Σ be a system with a secure initial state σ_0, and let **T** be a set of state transformations. If every element of **T** preserves the simple security condition and the *-property, then every σ_i, $i \geq 0$, is secure.

At times, a subject must communicate with another subject at a lower level. This requires the higher-level subject to write into a lower-level object that the lower-level subject can read.

EXAMPLE: A colonel with (SECRET, { NUC, EUR }) clearance needs to send a message to a major with (SECRET, { EUR }) clearance. The colonel must write a document that has at most the (SECRET, { EUR }) classification. But this violates the *-property, because (SECRET, { NUC, EUR }) **dom** (SECRET, { EUR }).

The model provides a mechanism for allowing this type of communication. A subject has a **maximum security level** and a **current security level**. The maximum security level must dominate the current security level. A subject may (effectively) decrease its security level from the maximum in order to communicate with entities at lower security levels.

EXAMPLE: The colonel's maximum security level is (SECRET, { NUC, EUR }). She changes her current security level to (SECRET, { EUR }). This is valid, because the maximum security level dominates the current security level. She can then create the document at the major's clearance level and send it to him.

How this policy is instantiated in different environments depends on the requirements of each environment. The conventional use is to define "read" as "allowing information to flow from the object being read to the subject reading," and "write" as "allowing information to flow from the subject writing to the object being written." Thus, "read" usually includes "execute" (because by monitoring the instructions executed, one can determine the contents of portions of the file) and "write" includes "append" (as the information is placed in the file, it does not overwrite what is already in the file, however). Other actions may be included as appropriate; however, those who instantiate the model must understand exactly what those actions are. Chapter 8, "Noninterference and Policy Composition," and Chapter 17, "Confinement Problem," will discuss this subject in considerably more detail.

5.2.2. Example: The Data General B2 UNIX System

The Data General B2 UNIX (DG/UX) system provides mandatory access controls (MACs). The MAC label is a label identifying a particular compartment. This section describes only the default labels; the system enables other labels to be created.

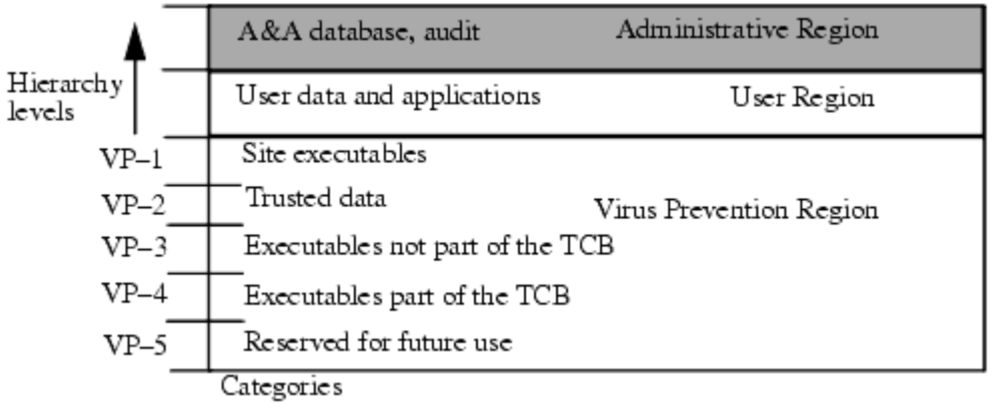
5.2.2.1. Assigning MAC Labels

When a process (subject) begins, it is assigned the MAC label of its parent. The initial label (assigned at login time) is the label assigned to the user in a database called the **Authorization and Authentication (A&A) Database**. Objects are assigned labels at creation, but the labels may be either **explicit** or **implicit**. The system stores explicit labels as parts of the object's attributes. It determines implicit labels from the parent directory of the object.

The least upper bound of all compartments in the DG/UX lattice has the label IMPL_HI (for "implementation high"); the greatest lower bound has the label IMPL_LO (for "implementation low"). The lattice is divided into three regions, which are summarized in Figure 5-3.[3]

[3] The terminology used here corresponds to that of the DG/UX system. Note that "hierarchy level" corresponds to "clearance" or "classification" in the preceding section.

Figure 5-3. The three MAC regions in the MAC lattice (modified from the DG/UX Security Manual [257], p. 4–7, Figure 4-4). TCB stands for "trusted computing base."



The highest region (administrative region) is reserved for data that users cannot access, such as logs, MAC label definitions, and so forth. Because reading up and writing up are disallowed (the latter is a DG/UX extension to the multilevel security model; see Section 5.2.2.2), users can neither read nor alter data in this region. Administrative processes such as servers execute with MAC labels in this region; however, they sanitize data sent to user processes with MAC labels in the user region.

System programs are in the lowest region (virus prevention region). No user process can write to them, so no user process can alter them. Because execution requires read access, users can execute the programs. The name of this region comes from the fact that viruses and other forms of malicious logic involve alterations of trusted executables.[4]

[4] The TCB, or trusted computing base, is that part of the system that enforces security (see Section 19.1.2.2).

Problems arise when programs of different levels access the same directory. If a program with MAC label **MAC_A** tries to create a file, and a file of that name but with MAC label **MAC_B** (**MAC_B dom MAC_A**) exists, the create will fail. To prevent this leakage of information, only programs with the same MAC label as the directory can create files in that directory. For the **/tmp** directory, and the mail spool directory **/var/mail**, this restriction will prevent standard operations such as compiling and delivering mail. DG/UX introduces a “multilevel directory” to solve this problem.

A **multilevel directory** is a directory with a set of subdirectories, one for each label. These “hidden directories” normally are not visible to the user, but if a process with MAC label **MAC_A** tries to create a file in **/tmp**, it actually creates a file in the hidden directory under **/tmp** with MAC label **MAC_A**. The file can have the same name as one in the hidden directory corresponding to label **MAC_A**. The parent directory of a file in **/tmp** is the hidden directory. Furthermore, a reference to the parent directory goes to the hidden directory.

EXAMPLE: A process with label **MAC_A** creates a directory **/tmp/a**. Another process with label **MAC_B** creates a directory **/tmp/a**. The processes then change the correct working directory to **/tmp/a** and then to .. (the parent directory). Both processes will appear to have **/tmp** as the current working directory. However, the system call

```
stat(".", &stat_buffer)
```

returns a different inode number for each process, because it returns the inode number of the current working directory—the hidden directory. The system call

```
dg_mstat(".", &stat_buffer)
```

translates the notion of "current working directory" to the multilevel directory when the current working directory is a hidden directory.

Mounting unlabeled file systems requires the files to be labeled. Symbolic links aggravate this problem. Does the MAC label the target of the link control, or does the MAC label the link itself? DG/UX uses a notion of inherited labels (called **implicit labels**) to solve this problem. The following rules control the way objects are labeled.

1. Roots of file systems have explicit MAC labels. If a file system without labels is mounted on a labeled file system, the root directory of the mounted file system receives an explicit label equal to that of the mount point. However, the label of the mount point, and of the underlying tree, is no longer visible, and so its label is unchanged (and will become visible again when the file system is unmounted).
2. An object with an implicit MAC label inherits the label of its parent.
3. When a hard link to an object is created, that object must have an explicit label; if it does not, the object's implicit label is converted to an explicit label. A corollary is that moving a file to a different directory makes its label explicit.
4. If the label of a directory changes, any immediate children with implicit labels have those labels converted to explicit labels before the parent directory's label is changed.
5. When the system resolves a symbolic link, the label of the object is the label of the target of the symbolic link. However, to resolve the link, the process needs access to the symbolic link itself.

Rules 1 and 2 ensure that every file system object has a MAC label, either implicit or explicit. But when a file object has an implicit label, and two hard links from different directories, it may have two labels. Let **/x/y/z** and **/x/a/b** be hard links to the same object. Suppose **y** has an explicit label IMPL_HI and **a** an explicit label **IMPL_B**. Then the file object can be accessed by a process at IMPL_HI as **/x/y/z** and by a process at **IMPL_B** as **/x/a/b**. Which label is correct? Two cases arise.

Suppose the hard link is created while the file system is on a DG/UX B2 system. Then the DG/UX system converts the target's implicit label to an explicit one (rule 3). Thus, regardless of the path used to refer to the object, the label of the object will be the same.

Suppose the hard link exists when the file system is mounted on the DG/UX B2 system. In this case, the target had no file label when it was created, and one must be added. If no objects on the paths to the target have explicit labels, the target will have the same (implicit) label regardless of the path being used. But if any object on any path to the target of the link acquires an explicit label, the target's label may depend on which path is taken. To avoid this, the implicit labels of a directory's children must be preserved when the directory's label is made explicit. Rule 4 does this.

Because symbolic links interpolate path names of files, rather than store inode numbers, computing the label of symbolic links is straightforward. If **/x/y/z** is a symbolic link to **/a/b/c**, then the MAC label of **c** is computed in the usual way. However, the symbolic link itself is a file, and so the process must also have access to the link file **z**.

5.2.2.2. Using MAC Labels

The DG/UX B2 system uses the Bell-LaPadula notion of dominance, with one change. The system obeys the simple security condition (reading down is permitted), but the implementation of the *-property requires that the process MAC label and the object MAC label be equal, so writing up is not permitted, but writing is permitted in the same compartment.

Because of this restriction on writing, the DG/UX system provides processes and objects with a range of labels called a **MAC tuple**. A **range** is a set of labels expressed by a **lower bound** and an **upper bound**. A MAC tuple consists of up to three ranges (one for each of the regions in Figure 5-3).

EXAMPLE: A system has two security levels, TS and S, the former dominating the latter. The categories are COMP, NUC, and ASIA. Examples of ranges are

[(S, { COMP }), (TS, { COMP })]

[(S, Ø), (TS, { COMP, NUC, ASIA })]

[(S, { ASIA }), (TS, { ASIA, NUC })]

The label (TS, {COMP}) is in the first two ranges. The label (S, {NUC, ASIA}) is in the last two ranges. However,

[(S, {ASIA}), (TS, { COMP, NUC})]

is not a valid range because (TS, { COMP, NUC }) **dom** (S, { ASIA }).

An object can have a MAC tuple as well as the required MAC label. If both are present, the tuple overrides the label. A process has read access when its MAC label grants read access to the upper bound of the range. A process has write access when its MAC label grants write access to any label in the MAC tuple range.

EXAMPLE: Suppose an object's MAC tuple is the single range

[(S, { ASIA }), (TS, { ASIA, COMP})]

A subject with MAC label (S, { ASIA }) cannot read the object, because

(TS, { ASIA, COMP}) **dom** (S, { ASIA })

It can write to the object, because (S, { ASIA }) dominates the lower bound and is dominated by the upper bound. A subject with MAC label (TS, { ASIA, COMP, NUC }) can read the object but cannot write the object. A subject with MAC label (TS, { ASIA, COMP }) can both read and write the object. A subject with MAC label (TS, {EUR}) can neither read nor write the object, because its label is incomparable to that of the object, and the **dom** relation does not hold.

A process has both a MAC label and a MAC tuple. The label always lies within the range for the region in which the process is executing. Initially, the subject's accesses are restricted by its MAC label. However, the process may extend its read and write capabilities to within the bounds of the MAC tuple.

5.2.3. Formal Model

no 5.2.3 and 5.2.4

Let **S** be the set of subjects of a system and let **O** be the set of objects. Let **P** be the set of rights r for read, q for write, w for read/write, and e for empty.[5] Let **M** be a set of possible access control matrices for the system. Let **C** be the set of classifications (or clearances), let **K** be the set of categories, and let $\mathbf{L} = \mathbf{C} \times \mathbf{K}$ be the set of security levels. Finally, let **F** be the set of 3-tuples $(\mathbf{f_s}, \mathbf{f_o}, \mathbf{f_c})$, where $\mathbf{f_s}$ and $\mathbf{f_c}$ associate with each subject maximum and current security levels, respectively, and $\mathbf{f_o}$ associates with each object a security level. The relation **dom** from Definition 5–1 is defined here in the obvious way.

[5] The right called "empty" here is called "execute" in Bell and LaPadula [68]. However, they define "execute" as "neither observation nor alteration" (and note that it differs from the notion of "execute" that most systems implement). For clarity, we changed the e right's name to the more descriptive "empty."

The system objects may be organized as a set of hierarchies (trees and single nodes). Let **H** represent the set of hierarchy functions $\mathbf{h}: \mathbf{O} \rightarrow \mathbf{P}(\mathbf{O})$.[6] These functions have two properties. Let $\mathbf{o_i}, \mathbf{o_j}, \mathbf{o_k} \in \mathbf{O}$. Then:

[6] **P**(**O**) is the power set of **O**—that is, the set of all possible subsets of **O**.

1. If $\mathbf{o_i} \neq \mathbf{o_j}$, then $\mathbf{h}(\mathbf{o_i}) \cap \mathbf{h}(\mathbf{o_j}) = \varnothing$.
2. There is no set $\{ \mathbf{o_1}, \mathbf{o_2}, \ldots, \mathbf{o_k} \} \subseteq \mathbf{O}$ such that for each $\mathbf{i} = 1, \ldots, \mathbf{k}$, $\mathbf{o_{i+1}} \in \mathbf{h}(\mathbf{o_i})$, and $\mathbf{o_{k+1}} = \mathbf{o_1}$.

(See Exercise 6.)

A state $\mathbf{v} \in V$ of a system is a 4-tuple $(\mathbf{b}, \mathbf{m}, \mathbf{f}, \mathbf{h})$, where $\mathbf{b} \in \mathbf{P}(\mathbf{S} \times \mathbf{O} \times \mathbf{P})$ indicates which subjects have access to which objects, and what those access rights are; $\mathbf{m} \in \mathbf{M}$ is the access control matrix for the current state; $\mathbf{f} \in \mathbf{F}$ is the 3-tuple indicating the current subject and object clearances and categories; and $\mathbf{h} \in \mathbf{H}$ is the hierarchy