Capabilities:

Capabilities can be viewed as rows in ACL. It is a ticket that a subject can present so a certain resource can be accessed.

1. It should not be forgeable
2. It should be transferable

Difference between ACL and capabilities:

1. ACLs are associated with objects and hence it is easy as objects reside inside the operating system. Capabilities are associated with users and hence have to be protected against tampering.
2. ACLs suit single machine scenario whereas capabilities suit distributed systems (e.g. Kerberos)

Capabilities in Kerberos

Kerberos uses capabilities. In Kerberos capabilities are applied across the systems (in a distributed manner which suits the very nature of capabilities) whereas within each system it uses a traditional access control mechanism.

Kerberos in Brief:

1. Central Server- A central server is a server where everyone gets authenticated to. It has 'n' keys for 'n' users. (Compare with the number of keys needed if pairwise authentication was attempted – that will require O(n^2) keys.)
2. Once a user U gets authenticated to the central server, she/he gets a ticket 't' for a specific service (say capability to "print" service.)
3. In this way the server providing the service (say 'print' service in this case) does not need to worry about authorization decisions (which printer can print on what printer at what time.) As we said before, "t" could be just a message "allow U access" that is encrypted using the key shared between (and only known to) the central server and the print server.

Implementing capabilities

It is obvious that unforgeable capabilities can be implemented using cryptography. For instance, in the case of Kerberos, suppose that a client C requests a capability to the print service P. The trusted central server can provide a capability to the client as follows. It can generate a message that says "allow C to print," encrypt it using the secret key that it shares with P, and send the result to C. Now, when C sends this capability to P, P knows that it must have been given out by the central server – there is no way for C to forge this message since C does not know P's secret key. Therefore the capabilities are unforgeable since it can only be created by the Kerberos server or the printer P. (Note: this is not exactly the way Kerberos works, but the essential ideas are similar.)

What may not be that obvious is that unforgeable capabilities can be implemented without using cryptography. For instance, a capability may be implemented using a pointer to a data structure that is held inside the kernel. The creation and maintenance of capabilities are performed inside the operating system instead of an user-level process; thus making it difficult for a capabilities to be manufactured outside the kernel. To make

it unforgeable, the kernel can keep track of the capabilities that a process has been given, so that any attempt to forge a pointer to a capability that it has not been given will be detected. The downside of this, though, is that this makes it difficult to transfer capabilities, which is supposed to be possible when using capabilities. (Note: unrestricted transfer abilities are not a good idea in most cases, so transferability should be limited.)

Using indices (instead of pointers) makes it unnecessary to keep track of valid capabilities. In particular, the OS keeps a table consisting of capabilities given to a process. Any attempt to forge will result in an invalid index. In this sense, the open file descriptors in UNIX can be thought of as capabilities. Note that this table is inherited across fork operations, and – this means that the child inherits the capabilities of the parent. It is also preserved across exec, which means that the capabilities are passed on to the new program executed from the old program.

Issues with capabilities:
Revocation is an issue. With a pure access control approach, revocation is as simple as removing the user from an ACL. With a pure capability based approach, a capability may have been handed to many users, so it may not be possible to prevent one user from accessing the resource while allowing others. In other words, with a pure capability based approach, revocation may prevent every one from using the resource.

Due to the revocation problem, capabilities must always be associated with an expiry date. This, of course, introduces additional management overhead.

Resource naming may be an issue --- if the resource name is reused (after the previous instance was deleted), you need to make sure that a capability to the old instance does not allow access to the new instance. You need unique resource ids to avoid this.

As an example, consider public key certificates. They must be uniquely named, must have an expiry date (How many times have you seen a “Server's certificate has expired” message?), and moreover, clients should check “revocations lists” distributed by the CA to verify if a certificate issued by the CA has been revoked. Clearly, there is quite a bit of administrative overhead as compared to managing access permissions on files.

Another issue with kernel-level capabilities is that the scope of transfer is only limited to the operating system. Capabilities that are implemented by the kernel can travel across other computers. For example, a file descriptor capability cannot leave the scope of the machine which the file resides. In addition, some capabilities (such as file descriptors) will be lost if the machine powers down.

5.

Mandatory access control (MAC):
Necessity for MAC- DAC has problems when the owner has not carefully and/or correctly configured the DAC mechanism. This may be the result of a lack of knowledge, time, or willingness to learn. As a result, the crucial sensitive files may be accessible to individuals who should not have access.

Unlike DAC, where the protection of a resource becomes the responsibility of its owner, in MAC, the permissions are set by a centralized process. This ensures that an organization's policies will be followed.

Other issues with DAC:

1. DAC can't differentiate between users and processes acting on their behalf. An example would be a malicious program that changes a file permission, thus making a file meant only for its owner's consumption to become readable by everyone in the world. (Such a program is called a Trojan Horse.)

With MAC in place, permissions are no longer controlled by users, so a malicious program executed by a user cannot change access control policies on an arbitrary file.

Note that DAC and MAC can co-exist on a system. In such combinations, an access will be granted if it is granted by both the DAC and MAC policies. Thus, in the Trojan Horse example, it can only change the DAC policies, but if the MAC policies are set correctly, they will continue to protect critical files from being accessed by those that should not have access.

There are many kinds of MAC policies --- we start our discussion with MLS.

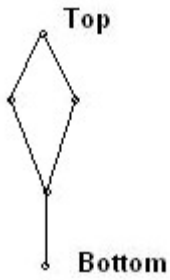
Multi Level Security (MLS):
Multi Level Security (MLS) is an example of MAC. The motivation for MLS is as follows. Access control mechanisms of the kind discussed before allows us to control which subject can access what information, but they provide no control over how this information can be used. MLS policies control information flow and hence provide some control over how information is used.

MLS confidentiality policies:
Confidentiality is an important requirement in military services and hence MLS confidentiality was mainly developed for them.

In MLS, objects are labeled with levels. Labels form a lattice. (A lattice is a partial order that has the following additional property: every two points in the lattice have a unique greatest lower bound and a least upper bound. This property implies that there will be a

unique top element that is greater than every element in the lattice, and a unique bottom element.)

In the simplest case, the partial order may be a total (aka linear) order, as illustrated below by the picture on the left. In other cases, it may not be a linear order, as illustrated by the picture on the right.



In military, the following (linear) lattice is commonly used.

- Top Secret
- Secret
- Classified
- Unclassified

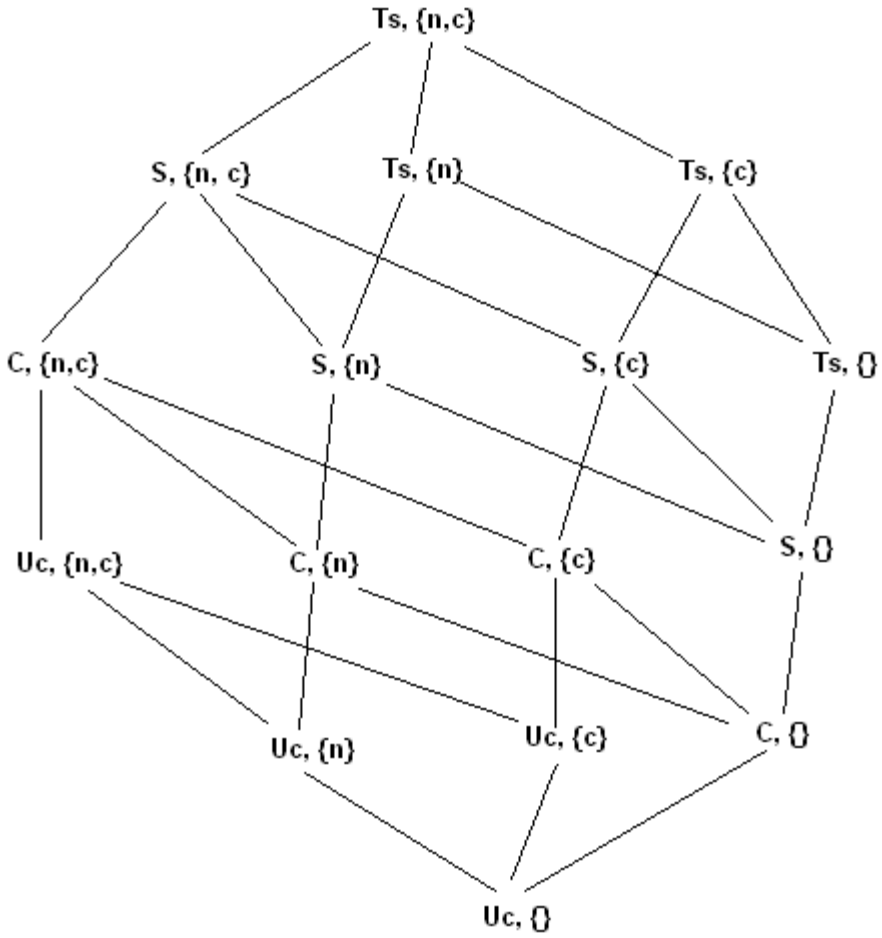
Information is also compartmentalized. E.g. - Nuclear, Crypto, etc. This way, a user does not have to be given access to all secret information, but to only that subset that also falls within specified compartments.

A user gets “clearance” to access information at certain levels. A subject can access the object if his/her clearance level is equal to or greater than that of the object. In the military, clearance requires some sort of background investigation of the user to ensure that he/she is trustworthy. Once granted, clearance remains in place for relatively long term (say, years). In addition, sensitive information is compartmentalized, and a user can access information in a compartment only if he/she has a need to know --- typically, this means that the user is working on some project related to that compartment.

When the two dimensions (level and compartment) are combined, the lattice becomes larger. The points in the lattice are a pair of the form (Level, CompartmentSet), with the partial order defined as follows: (L1, S1) >= (L2, S2) iff (L1 >= L2) and (S1 >= S2), where the ordering on sets is the subset ordering. A lattice with the above 4 levels and 2 compartments (“nuclear” and “crypto”) has the following structure.

In reality we have both levels and compartments and subjects refer to persons.

Ts- Top Secret
S- Secret
C- Classified
Uc- Unclassified
n- nuclear
c- crypto
{}- empty compartment set



Two Levels of MLS for Confidentiality:

a) MLS: Bell La-Padula model

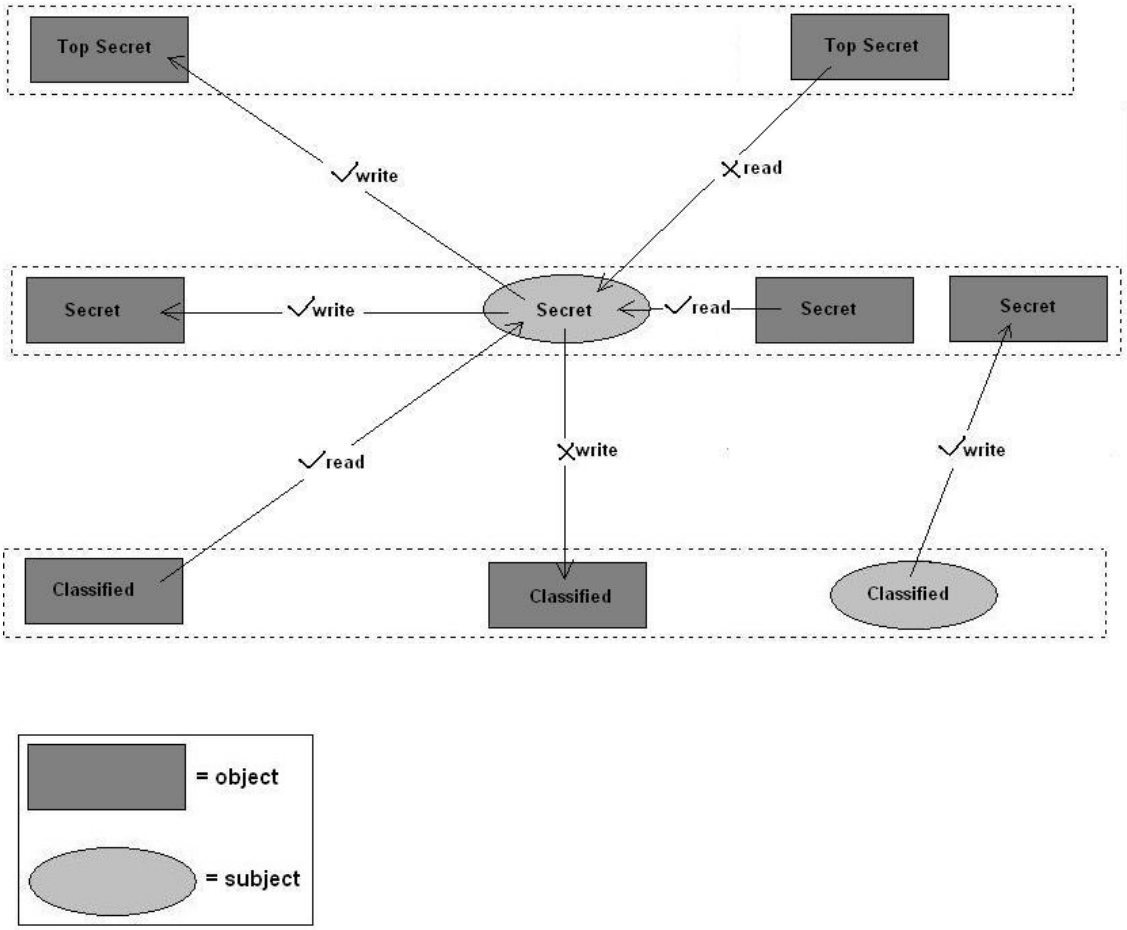
1. It is a model specifying the policy for information flow.
2. It enforces transitivity. E.g. - Process 'A' read File 'M' and outputs File 'N'. Process 'B' reads File 'N' and outputs File 'O'. Now another process 'D' can be constrained from accessing the contents of File 'O'.

Two Policies:

1. 'No read up'-
   A subject 'S' can read object 'O' only if Clearance[S] >= Level[O]
   (S must also be given access to the compartment in which O resides, but we will ignore this for simplicity here.)
2. 'No Write Down'-

A subject ‘S’ can write object ‘O’ only if Clearance[S] <= Level[O]

No “write-down” makes sure that no potential leaking of information exists. In general both the policies make sure that information does not flow from a higher level to a lower level. This is called as “*”-property.

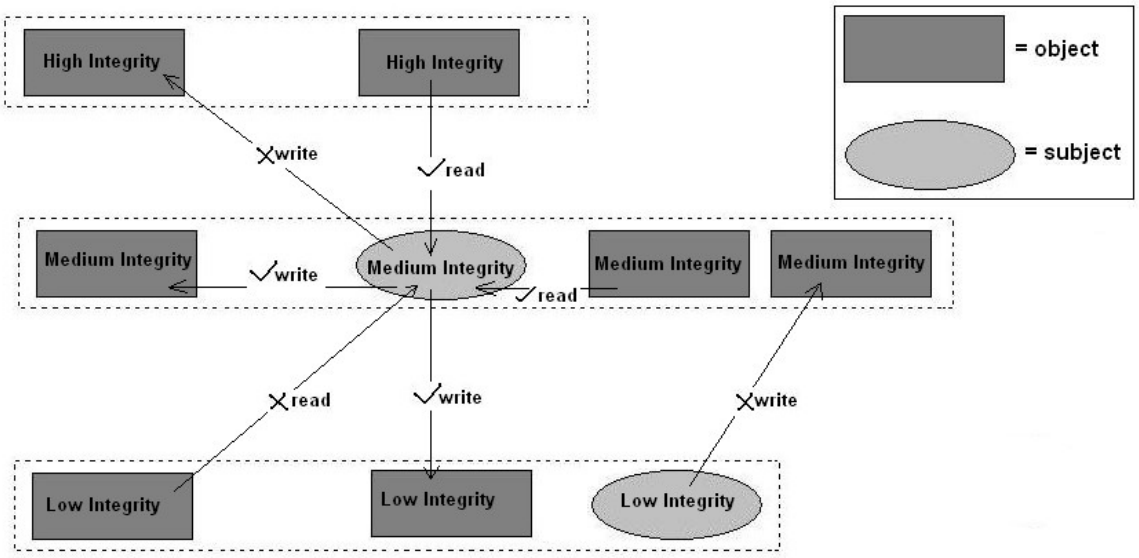


b) MLS Biba Model for Integrity

This model is mainly designed for integrity rather than confidentiality. It has the following two policies

1. ‘No read down’-
   A subject ‘S’ can read object ‘O’ only if C[S] <= O[S]
2. ‘No write up’-
   A subject ‘S’ can write an object only if C[S] >= O[S]

It states that a (benign) subject who relies on high integrity data should be able to produce high integrity data. ‘No read down’ policies prevent higher integrity processes from being exposed to lower integrity input data. “No write up” policies prevent lower integrityprocesses from over higher integrity data.



Labels in integrity and confidentiality:

An integrity label on an object reflects the trustworthiness of the data contained within that object. A high integrity label may be the result of either (a) some verification procedure that verifies that the data is trustworthy, or (b) the data was produced by highly trustworthy applications based on highly trustworthy inputs. In other words, it is either based on an independent verification of the data, or it reflects the fact that if we trust the process by which some data is produced, then we should trust the results as well.

In confidentiality the label reflects a policy on future use. In particular, just because an object is labeled "classified" does not necessarily mean that it is sensitive – it may be entirely public domain data. What it says though is that it should be treated as sensitive, and not be revealed to those with lower security clearance.

Low Water Mark Policy (LOMAC)

'read down' allowed but downgrade subject to the level of the object. E.g.- the copy binary. It can copy a high integrity file to a high integrity file. When copying a low integrity file, the copy process (subject) is downgraded to low integrity when it opens a low integrity file for reading. Subsequently, when it creates a copy, the copy has low integrity as it inherits its integrity from the process. As a result, copy program works as we would expect it to, when the LOMAC policy is used. In contrast, if the strict Biba model is used, the copy process would not be allowed to read a low integrity file. So, in order to copy low integrity files, you would have to create a second version of the copy program that has lo integrity, and can be used to copy low integrity files. Moreover, the user has to figure out which of the two copy programs needs to be used for a given copying operation.

Although LOMAC addresses some of the problems of strict Biba policy, it introduces new problems of its own. In typical OS, access permissions are given when a file is

opened. Consider a high integrity subject opens a high integrity file H for writing. If it subsequently opens a low integrity file L, its level is downgraded. Now, to ensure that high integrity files are not corrupted by this low integrity subject, the open file descriptor for H should be invalidated, i.e., any attempt to write using that descriptor should result in an error. This problem, where a program causes its existing rights to be revoked as a result of opening low integrity files, is called the "self-revocation problem."

Programs assume that once they have opened a file, they can always write into it; and hence they may not check error codes on writes, or they may not deal with them gracefully (ie the program may crash).

Label creation for objects

1. Objects inherit the integrity level of the subject. However a lower integrity level can be given explicitly, without violating the principles behind information flow based integrity.
2. For programs the subject will inherit the lowest among the integrities of its executable and libraries.

Information Flow Problems

1. *Label Creep:*
   More objects become sensitive making it difficult for the system to be used by lower level subjects. For instance, a subject may open top-secret, unclassified files, but one of its outputs may depend only on the unclassified file. However, MLS policies will classify this output as top-secret, preventing lower level processes from accessing this file.
   More generally, the result of processing a file can result in new files that are at the same or higher level. In other words, as new files are created, the number of files at higher levels of classification cannot decrease, and it actually tends to go up.
2. There are a large number of objects and subjects on a typical system, e.g., ~100K files on Linux. Labeling all objects and subjects correctly (so that system integrity will be preserved without breaking functionality) is difficult.
3. Exceptions need to be made- E.g. – Encryption program should be able to take a top secret data and produce a unclassified file – the point is that even though the output is at a lower level, since it is encrypted, no one can read it, and hence its low level does not compromise confidentiality goals. In MLS (for confidentiality) this cannot be supported directly, since it would, by default, set the level of output file as top-secret. So, we need to make exceptions. With MLS, this is done based on the notion of "trusted programs" -- such programs are exempted from the "*"-property. The problem, though, is that once there is an exception mechanism, it tends to be used heavily, as people want to get their work done, and when security stands in the way, they tend to use whatever exception mechanism is available to work around their problem. When the exception is unconditional and total, the danger posed by the exception is very large. This motivates our discussion of

DTE policies that allow us to capture “least privilege policies” for trusted programs, so that trust can be conditional and limited.

Alternative Approach: Dividing the Root Privilege

Most problems can be related to the fact that there is an unrestricted privilege called the root privilege. The root privilege has unrestricted access to the system thus if exploited can cause an attacker to gain unlimited control of the system. One way to mitigate this problem is by breaking down the root privilege into separate components. For instance, we can create a single privilege which only binds to low level ports and another privilege to read higher-level files. Therefore if a server had been compromised, its privilege allows it to bind to low level ports but it can’t read arbitrary files on the system. Therefore by separating the root privilege we can prevent the compromised applications from further compromising our system.