Multifactor authentication:

Authenticating people can be based on 2 factors:

- Something the user **KNOWS** : e.g. – a password or PIN
- Something the user **HAS**: e.g. – An ATM card, smartcard or hardware token, or a biometric (fingerprint, retina, ...).

Most systems use a single factor - passwords for authentication. Good passwords must have 2 properties:

- They must not be easily guessable.
- They must be easy to remember.

Since these two properties are conflicting, it is desirable to use shorter easy to remember passwords, coupled with more than one factor for authentication.
Multifactor authentication uses a combination of methods for authentication. A typical scenario could be usage of two factor authentication for authenticating a user to a ATM machine based on ATM card and a secret user PIN. One possible protocol that could be used by the ATM machine is as follows. The ATM card uses magnetic strip to store the information of the users account encrypted with a secret key known the ATM machines. The PIN number, along with other account information, could be stored either on the card in this encrypted form. . With this approach, the ATM machine can validate it locally, without having to contact the bank. Note that online attacks are made difficult in this scenario since the ATM machine would typically capture the card after a few unsuccessful attermpt (usually, 3 attempts). Offline attacks, where that attacker tries to read the contents of the magnetic strip offline and try to crack the PIN number stored on it, is rendered difficult by the fact that the attacker does not know the key used to encrypt the information stored on the magnetic strip. However, a malicious ATM can of course steal the PIN since the user has to enter it in plaintext.

There can also protocols where the PIN verification is done by the bank. Still, if the user has to enter her PIN in plaintext on the ATM machine, attacks using a malicious ATM are possible. To avoid such attacks, the bank can rely on a challenge-response protocol. For instance, the bank can send a challenge to the user, and expect the user to send a decrypted version of this challenge, decrypted using the PIN. Such a protocol, naturally, will require a smart ATM card that can respond to this challenge, rather than expecting the user to do this.

Access Control :

There are 3 kinds of entities related to access control: Users, Subjects and Objects.

A user is the actual end user who needs access to a resource in the system.
A subject is typically some process acting on behalf of user. (However, many access control mechanisms don't distinguish between users and the processes acting on their behalf.)
An object is a system resource, e.g., a file, a device etc. The goal is to control the operations performed by subjects on objects.

General principles and concepts:

- **Principle of least privilege:** By the principle of least privilege, a subject is assigned only those privileges which are absolutely necessary for a subject to complete its task. This makes the overall system less vulnerable to damage in case of a compromise, since the set of rights gained by the attacker as a result of the compromise will be minimized.
  To support realization of this principle, we need a mechanism for increasing or reducing rights (rights amplification and attenuation) as necessary. For example, a user wants to view a pdf file. The system can start off the pdf viewer with a small set of rights, say, the ability to read user files and GUI access, but not allow it to write files or access other types of resources on the system.

Note that the PDF viewer may be launched from a command shell that may be used for other purposes, and hence its privileges should not be restricted in the same way. In other words, the set of privileges must be reduced at the point when the command shell launches the PDF viewer. Instead of allowing permission change only at exec boundary (as in the PDF example above), we can imagine doing this at a finer granularity, e.g., an application may dynamically increase or lower its privilege set in order to practice the principle of least privilege. For example, a server application may utilize high privilege to bind itself to a low-numbered port (binding to ports < 1024 in UNIX requires root privileges), and then drop these privileges. This is commonly practiced by web servers to minimize the impact of a subsequent compromise (e.g., such a compromise will not allow arbitrary root-owned files to be overwritten.)

- **Reference Monitor:** Reference Monitor is the access mediator that intercepts access to all system resources to monitor and check for user privileges. e.g : The component of the OS that checks permissions before allowing access to system resources. There may be more than one reference monitor in a system due to following needs:
    - Layered system has many reference monitors: each layer may need to incorporate a reference monitor for mediating accesses from the higher layer.
    - Different security checks may be implemented by different reference monitors
- **Security kernel:** This is the hardware and software that implements reference monitor.
- **Trusted Computing Base (TCB):** The totality of access control mechanisms for the system is referred to as Trusted Computing Base. It is desirable to keep the TCB footprint as small as possible, in order to give us confidence that the TCB will work correctly, and hence security mechanisms will function as intended.

The operations could be among the following (and more):

1) Read
2) Write
3) Append
4) Execute
5) Delete
6) Change permission
7) Change ownership

Finer distinctions between operations helps in minimizing the privileges that are granted. For instance, append can be thought of as an instance of write, but for certain files, such as event or error logs, it is relatively safe to allow most subjects to append to it, but you don't want to give write access --- a write access introduces much higher risk damage (say, due to truncation or corruption of a log file) as a result of a bug (in the application that writes to the file) or an attack. However, for simplicity/ease of implementation, systems do tend to combine some of these operations --- for instance, x86 processors generally can't express permission on memory pages that permit a read and a write but not an execute. (Some newer 64-bit x86 processors do have this capability, which is sometimes called the NX-bit.)

Discretionary access control mechanisms (DAC) :
Discretionary access control (DAC) is an access policy determined by the owner of an object. There is a notion that objects are owned by users and so owners can set permissions. By setting appropriate permissions, the owner decides who is allowed to access the object and what privileges they have. DAC can be represented using a matrix where **Rowsrepresent Subjects** and **Columns represent Objects**, with each **intersection of row and column specifying the access rights** for that subject on the object. For example:

Users/Objects	/etc/password	/bin/login
Alice	Read	Read, Execute
Bob	Read, Write, Execute	

Admin	Read, Write, Execute	Read, Write, Execute

Fig 1: Access Control Matrix

The DAC can be decomposed into 2 access control mechanisms:

1. Access Control List (ACL)
2. Capabilities

Access Control List (ACL), can be thought of as a **column-wise decomposition** of the above matrix. With each resource, we associate the column corresponding to that resource. This column lists the users that have access to the resource, and for each user, identifies the operations they can perform. The ACL can then used by a Reference Monitor to mediate/intercept access to all resources and grant permission to only valid requests.

Capabilities, on the other hand, can be thought of as **row-wise implementation** of the above matrix. Each user is given a capability that identifies the objects that they can access, and for each object, the set of operation permitted. Capabilities cannot be forgeable because the operating system checks a user's capabilities before it permits an operation. Therefore if capabilities were forgeable, it would defeat the purpose of capabilities. In addition, capabilities must be transferable so that users can allow another set of users to access its object. For manageability, the capabilities of a user may be broken up into smaller pieces. Note that since the capabilities are handed to a user, and since a user isn't typically trusted by the system to always tell the truth, implementation of capabilities is a bit more involved than ACLs. A prime example of a common capability is a password since it is transferable (a user can share his/her password) and unforgeable (in other words, they are not (supposed to be) guessable).

Improving Manageability of permissions by Indirection:
The management of permissions can be simplified by using levels of indirection.
For instance, one can identify a group of users that must be given access to a certain file. This can be done by creating a group with these users, and allowing the group to read/write the file. If group members change, then it is easy to support this change by simply changing the definition of the group – this can be done on UNIX by modifying the /etc/group file. If the concept of user groups is not supported, it is painful to implement changes to group membership – we would have to hunt down all the files to which the original group members had access, and change their permissions individually so that the new members have the access, while the old members that are no longer members won't have the access.
(An obvious application of groups arises in the context of a source-code control for software projects. A group can be created for the projects, and all members of that group given access to the source code repository for that project. Thus, these users will be able to access (which may be read-only or read-write) the source code for the project.)

Other mechanisms to improve manageability include: inheritance/default permissions (how the permissions of newly created objects get decided), and negative permissions (you can say all users except Bob can access a file, which is simpler than having to list all the users explicitly).

Role-based access control (RBAC): In role based access control, we create roles corresponding to a particular position and provide access rights to the role. Like groups, roles provide a level of indirection that simplifies the management of access control policies.

Negative Permissions: Some times, it is more convenient to specify policies in the form "all users in these groups except the following." To state and enforce such policies, the system needs to support specification of negative policies. This is fairly common in many access control systems (but not in operating systems): you often hear about "allow" and "deny" rules, with the deny-rules serving as negative permissions. Negative permissions are also useful for implementing conflict-of-interest policies, where users that have certain access permissions should be denied access to others that might cause a conflict of interest, e.g., consultants for company A should be denied access to documents relating to another company B that is a competitor of A.

Rights propagation:
A system evolves over time – new users are created, permissions get changed over time. A system contains certain permissions on certain set of objects at one point in time.

Question: "Is it possible that a right *r* on object *o* be granted to a subject *s* in future?

For instance, can some piece of code change the permission of a file to allow write access by all users or specific user that is not supposed to have it.

Take a current state of a system. It contains a certain set of files and users, who have rights on these files. So the question is that if the system is continuously used, can there be a point of time when a file such as a password file has world write permission on it? Here, there are certain assumptions, e.g., operating system will permit permission changes on an object only if the object is owned by the subject performing this operation.

[Harrison, Rizzo, Ullman 1976] showed that the above problem is undecidable. While the construction behind the proof is interesting (i.e., to show that access control settings, together with OS enforced constraints on these settings, can model a Turing Machine), this result does not address the problem that is more interesting from a practical perspective. In practice, we typically want the ability to change the permissions on arbitrary files if that is desired --- this is the reason for the use of "administrator privilege" in commercial OSes, which essentially suspends permission checking for administrators. The more interesting practical question these days is whether a malfunction of some application on the system can lead to an unsafe system state. Given that administrators are unconstrained in their actions, this amounts to answering the question of whether there are vulnerabilities in this application – clearly, this is an undecidable problem as well. (But it is once of the problems that is studied in software vulnerability analysis --- these techniques identify possible vulnerabilities, although they cannot necessarily rule out the absence of all vulnerabilities.)

Implementation of DAC on UNIX:
In UNIX, every system resource is a file (except resources like TCP connections etc). Permissions are associated with persistent objects. No explicit permissions exist on transient objects like sockets – they are implicitly derived from permissions on persistent objects when they get created.

File System Permissions :

- Each file has an owner, group owner.

- Permissions are divided into 3 parts. Each of which has a read/write/execute access. Therefore we have 9 bit string to represent access information of a file, plus 3 additional bits.
- Only owner can change the file permission.(POSIX)
- Only root can change the owner of the file. (POSIX)
- Some operating systems also have a special permission bit that allows users to assign privileges to other users to modify the permission bits, but this kind of mechanism is more complex to understand and use correctly.
- Directories have special meaning of these permission bits. (Detailed below)

Owner	Group	Other	Special
Read| write|	Read| write|	Read| write|	Suid | Guid | Sticky bit

execute	**execute**	**execute**	

The way these checks are carried out is as follows:

- First, note that each file has a owner and a group owner. When a user with specific user ID tries to access a file, the UID of user is checked with owner ID of the file. If there is a match, the permission associated with the owner bits is used. If not, it is checked if the user belongs to the group identified as the group owner of the file, and if so, the permission bits associated for the group are used. Finally, the permission bits associated with the world are checked.
- Subjects typically inherit the userid of parent. Authentication programs like the login program or ssh server has to explicitly set this information. To support this, UNIX allows root-owned processes to change their userid. (Obviously, non-root processes cannot change their userid.)
- UNIX uses username only for the purpose of logging in. After this point, it is the userid (a 16-bit value specified in the /etc/passwd file) that is used in permission checking.
- A super user has ID = 0. No permissions are checked for super user.
- Setuid() can be used for delegation and amplification of rights, as described below.

Additional permissions: suid, sgid and sticky bits

SUID bit :

When a user logs in the login process verifies the credentials of a user and loads the user previleges as assigned to the user and invokes the shell program. (In unix the fork system call is used to spawn new child processes, and the execve system call is used b a running process to load and run another executable. In this context, the login process forks, and its child execve's a shell.) Although the login process had root privileges, the shell no longer has those providences. But there are times when the users need to perform operations that require root privileges.

Example:

Consider ls command :

Shell → ls → fork → exec ls → ouputs a listing of current directory

The above is the sequence that is followed when a user executes an ls command on the shell promt.
Sometimes we need to increase the privilege of the command being executed temporarily.
For instance, consider a password change. It requires a write operation on the /etc/passwd (and/or /etc/shadow) file. However, if this permission is granted to all users, then the security of entire system is defeated, since a user can now change anyone's password, and hence be able to login as that user. An alternative is to allow the user to execute a trusted program (/sbin/passwd) that operates with the privileges necessary to edit the password files, but since the program won't be under the control of the user, it can ensure that it is used by a user only for the purpose of changing her password, but not that of others. The suid permission mechanism supports such programs.

The way suid bit operates is as follows: If the bit is set on a file, when the file is executed, the resulting process assumes the userid of the owner of the file, rather than the default of assuming the userid of the process performing the execve operation.

In the case of the trusted program /sbin/password which is owned by the root and has the suid bit set on itself, and hence when executed the program gets root privileges and it can edit the password for the given user.

Note that setuid is a powerful primitive and may be misused. Its design needs care --- for instance, in the normal case, a user has complete control over his processes --- they can debug the process, modify its memory, send signals, and so on. But if this is allowed for setuid processes, then the user can compromise the setuid process. Thus, the user rights as it relates to setuid processes are limited --- they can send signals, but not write to its memory. In addition, they can't change its behavior indirectly, e.g., by setting LD_LIBRARY_PATH which controls the locations where dynamically loaded libraries (called shared libraries) are searched for.

Setuid bit is used for **delegation:** Owner of executable is willing to perform certain actions that require owner privileges; the owner is willing to delegate these rights to any one that runs the program that has its setuid bit set. When the owner delegating privileges is root, this is also called amplification.

Setgid bit:
Likewise, setgid bit operates in a similar way, but sets the groupid of the process rather than userid. (Note: in UNIX, a file has a single group owner, but a process will have multiple groupids --- these correspond to the list of groups to which a user belongs. This list is normally computed and set at login time, and inherited across fork/exec.).

Example:
In gaming programs, there needs to be an update of files that maintain highest score. To ensure that this file can be updated by anyone that runs the game, without allowing users to directly edit this file (the latter right is highly likely to be abused), the following is done. A new group is created for use by the game. The highscores file as well as the game executable is set to have this group ownership. The game program is also setgid. With this configuration, the game program can update the highscores files regardless of who runs it, but the user cannot directly modify the highscores file.

umask : umask (user creation mode mask) affects the default file creation mode in Unix environments. User can set the umask using the umask command. Umasks are specified as a 3 bit octal and actual default permissions are obtained by aking away the rights mentioned in the umask from full access mode. The full access mode is 666 in the case of files, and 777 in the case of directories.

Eg : If umask is set to 022, the default permissions on creating a new directory are : 777 ^ 022 = 755.
(ie : rwxr-xr-x)

Generally, users add umask command to startup file like .bashrc so that the default is set for all future login sessions.

Meaning of permission bits for directories:

- read allows users to list contents of directory.
- write allows users to create or delete files in a directory. Users cannot overwrite a file unless they have write permission on it. However, users can achieve the same effect by deleting the file file first and recreating it.
- execute permission allows user to change into that directory (using cd command) and read files from that directory. If the user knows a filename, he/she can still read it despite no read permissions on directory if user has execute permission on the same. Reasonable scenarios for this situation exist! e.g. In Web servers, it is desirable to allow users to read files (eg: clicking on known links) but it is not a good idea to allow users to see a list of files served by the web server.

Sticky bit: The sticky bit provides the ability to control overwrites in a directory with write permissions. A example scenario: We want to allow someone to create new files but should not be able to delete files without having write permission on file. In such cases, we can set directory write permission and sticky bit. This ensures that a file can get deleted only by someone that has write permission on the file. Generally

used while handling shared folders among a group of users to avoid unintentional or intentional deletion of files by other users who do not have permissions.

ACLs : Some modern UNIX versions go beyond the 12-bit (9 for user/group/other access, and 3 special bits: setuid, setgid and sticky bit) model for allowing explicit access-control lists that can grant permissions to specific users and groups. This provides additional flexibility and granularity in permissions.

Changing permissions and ownership
In Unix, the ability to change permission is not modeled as a permission – instead, there is a hard-coded policy that says that only the owner of a resource can change its permission. Similarly, there is another hard-coded policy that says that only the root can change ownership of a file.

While one may fault this hard-coding as the lack of a general design, it does have benefits. It is very simple to use. If the ability to change permissions is granted to someone, that could indirectly allow the user to take complete control of that resource – a user may not realize this ramification and hence may end up using the "more general" capabilities incorrectly, in effect achieving less security and/or functionality.

DAC on Windows v/s UNIX:
The principles are roughly the same in Windows as UNIX, but the details may be a bit different. So we basically discuss the difference here.

- Windows uses an Object Oriented Design. Therefore operations can differ depending on the object. As against everything in UNIX is a file and has 12 bits to define permissions enforced and operations allowed on it. File objects in NTFS have additional (deletion, modification of ACL, take ownership) operations in Windows. By giving the permission to modify ACL associated with a file, another person is allowed to control how the file is being used.

- Files inherent permissions from the directory to which they belong.

- Uses registry for configuration of data. This information is saved in various files in /etc in UNIX. Use of registry leads to a better organization to the data stored in Windows. Many more operations are allowed via registry entry than the ones on NTFS files, like creating sub keys, being notified via emails on changes in the registry, etc.

- Mandatory file system locks are used in Windows. As a result, you tend to notice a behavior on Windows that prevents users from deleting files which are in use. However, it is not a standard practice to have mandatory file locks in UNIX. Mandatory locks are prone to misuse that can lead to deadlocks in the system, which is why it is not supported by default on UNIX, although it can be enabled if the administrator so chooses.

- There is no equivalent of suid mechanism in windows. So there cannot be amplification of privileges at any point of time in Windows.

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.

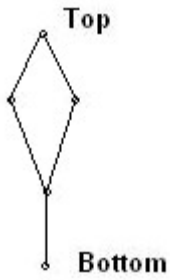
Multi Level Security (MLS):
Multi Level Security (MLS) is an example of MAC. The motivation for MLS is as follows. Access control mechanisms of the kind discussed before allows us to control which subject can access what information, but they provide no control over how this information can be used.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

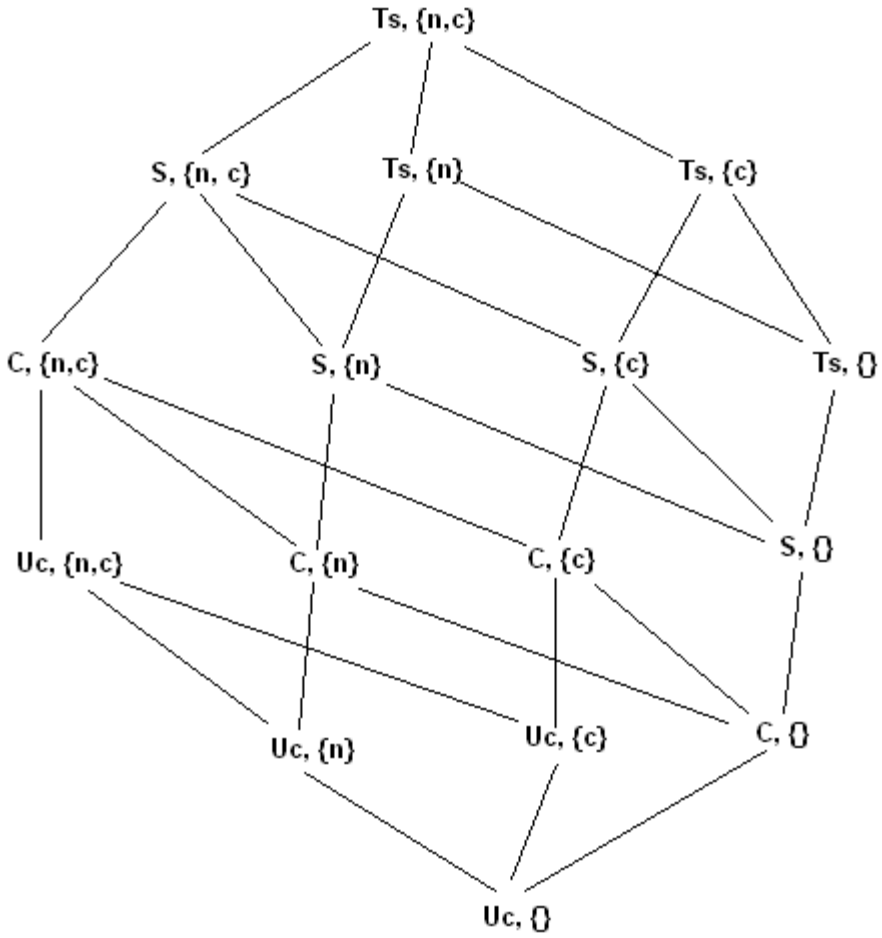
Information is also compartmentalized. E.g. - Nuclear, Crypto, etc. This way, a user does not have to be given access to all secret information, but to only that subset that also falls within specified compartments.

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

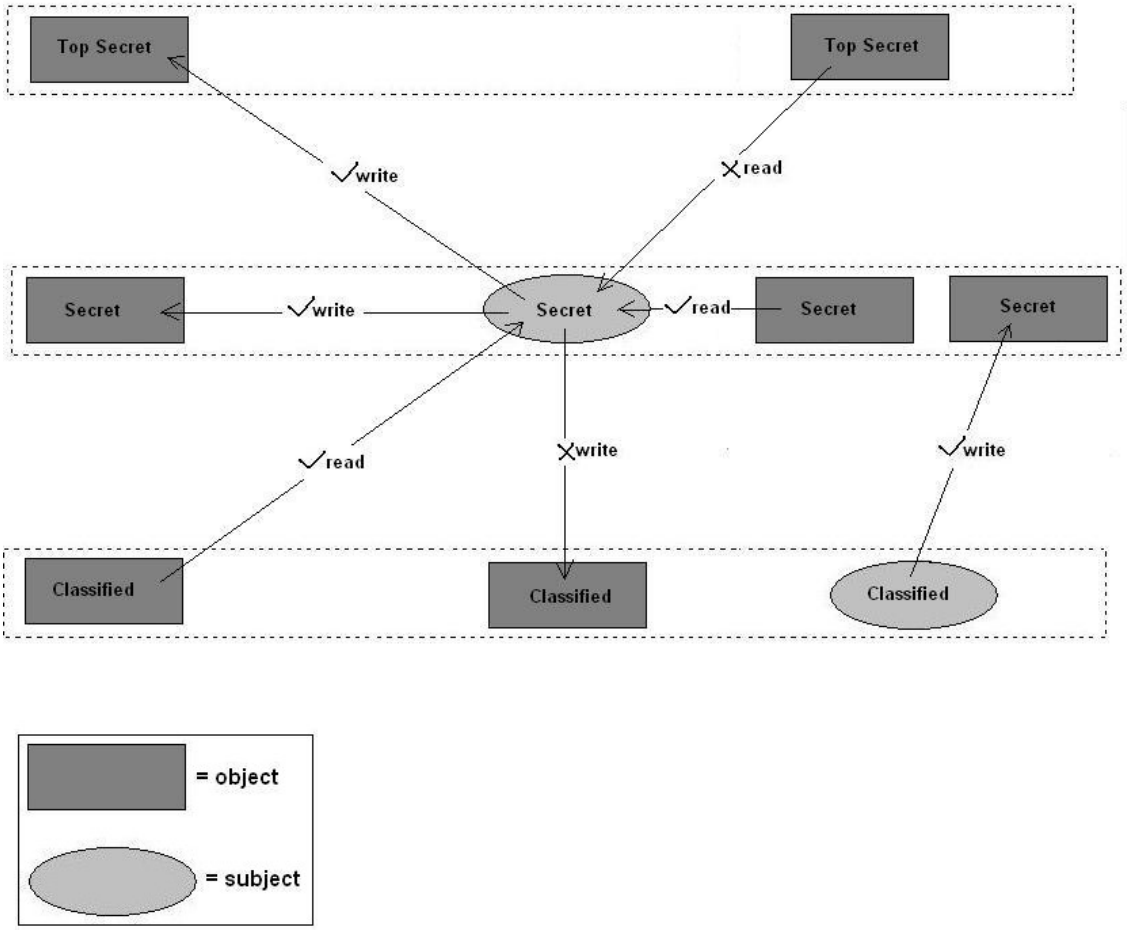
1. It is a model specifying the policy for information flow.
2. It enforces transitivity. E.g. - Process 'A' read File 'M' and outputs File 'N'. Process 'B' reads File 'N' and outputs File 'O'. Now another process 'D' can be constrained from accessing the contents of File 'O'.

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.

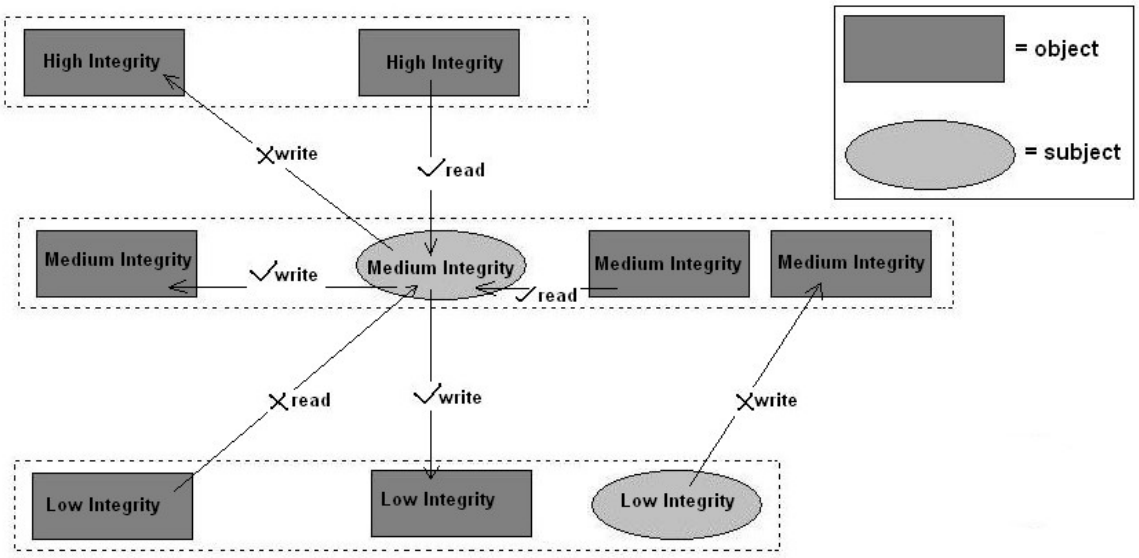


b) MLS Biba Model for Integrity

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.

Problems with Information flow :

The increasing usage of information flow based policies may lead to many objects becoming sensitive, making it difficult for the system to be used by lower clearance objects. This is called “Label Creep”.

We need to enforce legitimate exceptions in the policies. If trusted programs are allowed to be exempted from many of the policies, then there always exist chances of misuse. It is also difficult to configure the whole system and be confident about the strength of the security.

So, we require alternative and hybrid approaches that can overcome the main concerns like:

- Many programs being tagged as trusted
- Trusted programs themselves having hidden vulnerabilities

1.Domain and type enforcement:

1.1 Introduction

Domain and type enforcement (DTE) is an access control mechanism which helps in controlling the accesses granted to specific programs.

DTE defines “domains” as a set of subjects and “types” as a set of objects. You may recall that subjects are entities like processes that act on behalf of a user and objects are entities like files, sockets etc.. on which subjects operate on.

The enforcement of policies in DTE is such that a system administrator knows exactly what domains access what types. The policies basically need to handle:

- Restrictions of accesses from domains to types
- Restrictions of transitions from one domain to other domains

DTE can help enforce the principle of least privilege to processes, typically based on the functionality and usage of a particular process. For instance, consider the encryption example discussed before in the context of MLS. We had to make that program fully trusted. In this context of DTE, we can grant it more rights than other programs, but still don't have to give complete access. For instance, if the encryption program is used mainly to transmit sensitive data over a public network, then the program can be given access to write to the network, but it need not be given the ability to overwrite local files.

Another example: you can have a “viewer” program that can be used by multiple applications (programs) which may want to provide some interface for the users to view output data. When the viewer program is invoked, DTE policies can ensure that the viewer can read various files, but is very limited in terms of the files it can write, or other types of resources that it can access. (For instance, it may be permitted to write its own preferences files.) We can arrange for a domain transition at the point of invocation of a viewer program so as to ensure that these restrictions can be selectively applied to viewers without having to be applied to the programs that invoke them.

1.2 : Domain Transitions:

As was mentioned in the introduction to DTE, policies should enforce restrictions not only on domains accessing types, but also on domain transitions.

Example DTE configuration

Note that after booting, a number of processes are started up initially. These processes, called daemons, are responsible for all subsequent operations of the system, including the startup of network services, allowing user logins, and so on. As such, many of these daemon processes need to run with root privileges. In the absence of mechanisms like DTE, this need for root-privilege can be exploited by attackers. In particular, suppose that a network server runs with root privilege. If an attacker can exploit a vulnerability on this server to inject code, then this code can do a lot of damage. For instance, the attacker can install a “rootkit,” which can be thought of as a set of Trojan programs that allow attackers future access to the system (e.g., they can remotely login into the system) but hide their presence as well as the presence of attacker-launched processes. Installing a rootkit requires the attacker to overwrite important system binaries, which is possible because the exploited processes were running with root privileges.

The figure below shows how we could configure DTE to eliminate the possibility of rootkits being installed as a result of exploiting daemons. The figure shows 5 domains. The daemon domain is not permitted to install new executables/libraries or overwrite existing ones. This is permitted only in the “admin” domain --- when a system administrator wants to install new programs, she needs to go into that domain. Guarding access to this domain is a new login program that performs user authentication. It can perform stronger authentication before transitioning to the admin domain, as compared to the transition to the user domain.

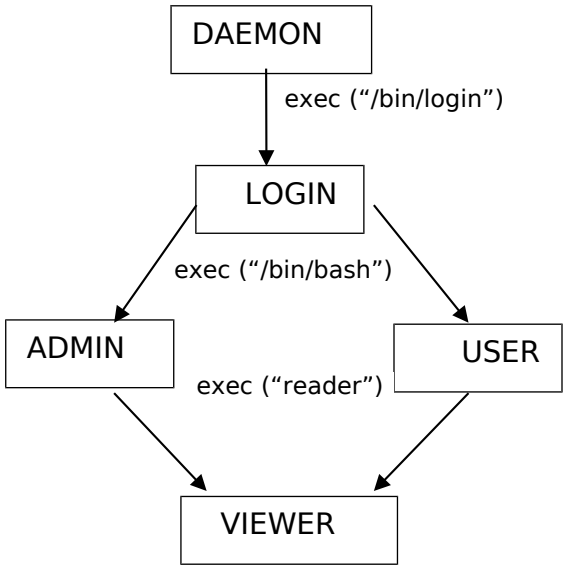


Figure: 1.1: A highly simplified example of DTE domains

A transition from the daemon domain to login domain (a domain that contains a single application, namely, the new login program) takes place when the login program is executed. The login program can transition to admin domain or user domain --- it has to specify which --- by executing the shell program. In the former case, a shell with full access, including the ability to install or upgrade software, is launched. In the latter case, the rights can be restricted so that normal user activities can be carried out but not tasks that require overwriting system programs. We can also limit access to the login program (ie prevent its execution) from user domain. In addition, we can ensure that the only way to transition into admin domain is using the login program. As a result, no matter what happens, it

becomes impossible to install or upgrade binaries without first going through strong authentication incorporated into the login program for access to the admin domain.

The figure also shows a “viewer” domain, where the accesses are further restricted so that file writes are not possible. As a result, even if a viewer program is attacked and compromised, it will not be possible for an attacker to create or modify any files on the system.

It is clear from this discussion that DTE can help in practicing the principle of least privilege. It allows for privilege amplification as well as attenuation. While all this flexibility is good when DTE is viewed as a mechanism, it does complicate policy development. For instance, DTE policies in SELinux are quite large and complex. They are difficult to develop, understand, or administer.

Another problem with DTE is that in reality, we end up creating one domain for each application. Part of the problem is that the basic DTE does not provide much in terms of parameterization, which makes it difficult to reuse policies across different applications. In SELinux, they attempt to mitigate this by using macros, but macros dont provide particularly clean abstractions, so in the end, the policies become difficult to understand or reuse.

Finally, while DTE allows each domain to be confined in a fine-grained fashion, it provides no help in understanding the overall effect on a system. Compare this with MLS, where we can assert overall properties such as the absence of information leaks from highly sensitive data to low-sensitivity data.

2. DTE and SELinux:

SELinux stands for “Security-Enhanced Linux”.

As the name suggests, they are talking about a Linux system with inbuilt enhanced security features.

SELinux combines the standard UNIX DAC and DTE. (It can also support MLS and a whole slew of other policies, but in practice, its essence is DAC and DTE.)

3. DTE and MLS:

In MLS, you can either not trust a program or completely trust it. Whereas, in DTE, you can limit the access rights such that the rights granted are just enough for the program to run as it is intended to.

DAC and DTE are mechanisms whereas MLS can be thought of as a high-level policy.

In DTE, the security is dependent on the configuration of the system and classifying domains, whereas in MLS, policy mostly involves object labeling (attributing access rights).

4. Linux (POSIX) capabilities:

The main idea here is to decompose the root privilege into several components, so that instead of giving root privilege, only subset of these privileges are available. This enables better adherence to the least privilege principle and hence lead to more secure systems. can be provided to the legitimate user and process to reduce the damage due to compromise. Following are some of the capabilities to which root privilege can be decomposed:

CAP_CHOWN: Capability to change the owner of any file.

CAP_DAC_OVERRIDE: Capability to override the DAC permissions.

CAP_NET_BIND_SERVICE: Capability to bind to low numbered ports (typically below 1024).

CAP_SETUID: Capability to change the uid of the process.

CAP_SYS_MODULE: Capability for the process to load kernel.

You can learn more by doing “man capabilities” on a POSIX-compatible OS like Linux or by searching on the same string on the Internet.

4.1 Effective, Permitted and Inheritable Capabilities for processes:

There are 3 types of capabilities associated with a process:

Effective capability is a set of capabilities that signify whether the current process has a particular privilege. This is similar to effective uid in DAC.

Permitted capability is the set of capabilities that the process can have. Effective capability can be changed only if the new set of capabilities is a subset of permitted capability for the process. This is similar to the real uid.

Inheritable capability is the set of capabilities that can be retained by child process of current process. When a child process is created using execve, capabilities of the child is determined by calculating permitted capabilities intersected with the effective capabilities and then masked with inheritable capabilities.

4.2 Attaching capabilities to executable files:

There are 3 types of capabilities associated with executable files:

Allowed capability is a set of maximum capabilities that an executable file can have. This is the superset of all the allowed capabilities for the file.

Forced capability is the set of capabilities that the executable file is given that did not belong to the parent process executing it. This is similar to setuid and is a mechanism for privilege amplification.

Effective capability is the set of capabilities that can be retained by the child process executing this file from the effective capabilities of the parent process.

5. Commercial policies:

High-level policies in commercial environment (like banks) are different from those enforced in military environments. There is a difference of degree of confidentiality expected in military setting and degree of data integrity expected in commercial settings.

Example policies:

- Clark-Wilson policy

- Chinese wall policy

5.1 Clark-Wilson policy:

In this policy, emphasis is on data integrity. Data integrity is a major concern in commercial deployments: in these setting, errors and fraud correspond to attacks on integrity, while confidentiality threats can largely be addressed using processes, procedures, regulations, and laws.

To come up with integrity policies in a domain-independent way, C/W is phrased in terms of transactions. C/W does not constrain the semantics of these “Well-formed transactions” (WFTs) but requires that they maintain system integrity, i.e., each WFT takes the system from one consistent state to another. An integrity verification procedure (IVP) may be used periodically to verify consistency. (Note: consistency and integrity are being used synonymously here.)

These transactions are assumed to be aware of the data integrity requirements and hence preserve them. However, there is the issue of integrity of the WFTs themselves --- i.e., we need to be sure that a fraudulent WFT was not performed. For instance, consider a transaction that transfers money from one customer account to another. This WFT maintains overall data consistency, since the total amount of monies across all accounts in unchanged in the absence of money coming into or going out of the bank. However, it is still possible for a fraudulent bank clerk to initiate such a transaction to transfer money from customer accounts to her own account. To prevent such fraud:

1. WFTs could be launched only by authorized users. Proper authentication should be used. Moreover, mechanisms must be in place to restrict access to WFTs, e.g., we may want to limit a person to performing travel voucher approvals, and another to process equipment purchase vouchers.

2. Critical functions need to be performed by multiple users. This is called “separation of duty.” For instance, in a transaction involving the processing of travel vouchers, the person approving the voucher should be different from the one making the claim. Similarly, in the money transfer example, the person performing the transfer should be different from the ones that own the accounts in question.

3. Auditing to verify integrity of transactions. Auditing is used to check if various processes and policies set up to protect the integrity of WFTs were in fact followed. The purpose of auditing is to ensure actions can be traced and attributed. (Such attribution serves as a powerful deterrent against misuse and fraud.)

4. Maintain adequate logs so that if errors/fraud were discoverd, the WFTs in question can be undone.

5.2 Chinese Wall policy:

In this policy, emphasis is on conflict of interest when dealing with confidential information.

This policy is defined in terms of:

- CD (Company Data) – objects related to a single company.

- COI (Conflict Of Interest) – sets of competing companies

The policy specifies that no person can have access to two or more CDs which come under the same COI class. I.e. a consultant cannot provide service to companies X and Y if X and Y are competitors. And this applies to past, present or future access, i.e., if a user has had past access to the data of company X, they can never work for company Y. (In the real-world, there is often a time limit, or finer granularity separation among the data belonging to the same company.)

6. Delegation:

Delegation refers to the ability to transfer certain rights to an entity to operate on behalf of the initiator. For instance, if a principal A requests a service from another principal B, then A must be willing to transfer any rights that B may need in order to service A's request.

Example: Requesting a print server to print a file, where (a) the file is present on a file server, and (b) the file server will only allow the user to access his files. In this case, the user has to grant access rights to the print server to read files from the file server on his behalf. As you may have observed this can be accomplished by public key cryptography.

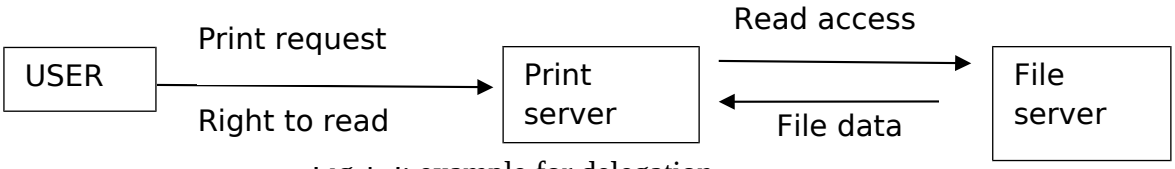


Fig 1.2: example for delegation.

It can also be done using symmetric key cryptography, in a manner similar to the printer example we used with Kerberos in one of the previous lectures.

Trust management is concerned with specification and enforcement of security policies in a distributed setting that is characterized by explicit statements regarding trust. A principal can say which principals it trusts for what information. It is convenient to think of these in terms of attributes, where policies specify which principals can vouch for which attributes. For instance, an instructor can state that a course page is can be accessed by a user provided the registrar determines that the user is registered for the course. In this example, the instructor's policy delegates the responsibility of determining course registration information to the registrar.

Operating System security:

The system is layered and each layer is supposed to cater to the requests of the upper layers.

This means that the user, and user applications need to be at the topmost layer followed by the kernel and then at the bottom most layer by the hardware.

In ensuring security of such a system, we do not assume that the higher layers may not attempt to bypass all or some of the lower layers. An attacker can try to attack at any layer whichever exhibits vulnerability and is easier to attack.

Memory Protection:

One of the important aspects of Operating system security is Memory Protection. Without memory protection, other protection mechanisms may be bypassed. For instance, suppose that process A is permitted access to a file F, while process B is not. Process B can bypass this policy by attempting to read F from A's memory. Alternatively, B may attempt to modify the policy that is stored in the OS memory so that B is allowed access to F.