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.