File Protection Mechanisms

Until now, we have examined approaches to protecting a general object, no matter the object's nature or type. But some protection schemes are particular to the type. To see how they work, we focus in this section on file protection. The examples we present are only representative; they do not cover all possible means of file protection on the market.

Basic Forms of Protection

We noted earlier that all multiuser operating systems must provide some minimal protection to keep one user from maliciously or inadvertently accessing or modifying the files of another. As the number of users has grown, so also has the complexity of these protection schemes.

All "None Protection

In the original IBM OS operating systems, files were by default public. Any user could read, modify, or delete a file belonging to any other user. Instead of software- or hardware-based protection, the principal protection involved trust combined with ignorance. System designers supposed that users could be trusted not to read or modify others' files, because the users would expect the same respect from others. Ignorance helped this situation, because a user could access a file only by name ; presumably users knew the names only of those files to which they had legitimate access.

However, it was acknowledged that certain system files were sensitive and that the system administrator could protect them with a password. A normal user could exercise this feature, but passwords were viewed as most valuable for protecting operating system files. Two philosophies guided password use. Sometimes, passwords were used to control all accesses (read, write, or delete), giving the system administrator complete control over all files. But at other times passwords would control only write and delete accesses , because only these two actions affected other users. In either case, the password mechanism required a system operator's intervention each time access to the file began .

However, this all-or-none protection is unacceptable for several reasons.

- Lack of trust . The assumption of trustworthy users is not necessarily justified. For systems with few users who all know each other, mutual respect might suffice; but in large systems where not every user knows every other user, there is no basis for trust.
- All or nothing . Even if a user identifies a set of trustworthy users, there is no convenient way to allow access only to them.
- Rise of timesharing . This protection scheme is more appropriate for a batch environment, in which users have little chance to interact with other users and in which users do their thinking and exploring when not interacting with the system. However, on timesharing systems, users interact with other users. Because users choose when to execute programs, they are more likely in a timesharing environment to arrange computing tasks to be able to pass results from one program or one user to another.
- Complexity . Because (human) operator intervention is required for this file protection,

operating system performance is degraded. For this reason, this type of file protection is discouraged by computing centers for all but the most sensitive data sets.

- File listings . For accounting purposes and to help users remember for what files they are responsible, various system utilities can produce a list of all files. Thus, users are not necessarily ignorant of what files reside on the system. Interactive users may try to browse through any unprotected files.

Group Protection

Because the all-or-nothing approach has so many drawbacks, researchers sought an improved way to protect files. They focused on identifying groups of users who had some common relationship. In a typical implementation, the world is divided into three classes: the user, a trusted working group associated with the user, and the rest of the users. For simplicity we can call these classes user, group, and world . This form of protection is used on some network systems and the Unix system.

All authorized users are separated into groups. A group may consist of several members working on a common project, a department, a class, or a single user. The basis for group membership is need to share . The group members have some common interest and therefore are assumed to have files to share with the other group members. In this approach, no user belongs to more than one group. (Otherwise, a member belonging to groups A and B could pass along an A file to another B group member.)

When creating a file, a user defines access rights to the file for the user, for other members of the same group, and for all other users in general. Typically, the choices for access rights are a limited set, such as {read, write, execute, delete}. For a particular file, a user might declare read-only access to the general world, read and write access to the group, and all rights to the user. This approach would be suitable for a paper being developed by a group, whereby the different members of the group might modify sections being written within the group. The paper itself should be available for people outside the group to review but not change.

A key advantage of the group protection approach is its ease of implementation. A user is recognized by two identifiers (usually numbers): a user ID and a group ID. These identifiers are stored in the file directory entry for each file and are obtained by the operating system when a user logs in. Therefore, the operating system can easily check whether a proposed access to a file is requested from someone whose group ID matches the group ID for the file to be accessed.

Although this protection scheme overcomes some of the shortcomings of the all-or-nothing scheme, it introduces some new difficulties of its own.

- Group affiliation . A single user cannot belong to two groups. Suppose Tom belongs to one group with Ann and to a second group with Bill. If Tom indicates that a file is to be readable by the group, to which group(s) does this permission refer? Suppose a file of Ann's is readable by the group; does Bill have access to it? These ambiguities are most simply resolved by declaring that every user belongs to exactly one group. (This restriction does not mean that all users belong to the same group.)
- Multiple personalities . To overcome the one-person one-group restriction, certain people

might obtain multiple accounts, permitting them, in effect, to be multiple users. This hole in the protection approach leads to new problems, because a single person can be only one user at a time. To see how problems arise, suppose Tom obtains two accounts, thereby becoming Tom1 in a group with Ann and Tom2 in a group with Bill. Tom1 is not in the same group as Tom2, so any files, programs, or aids developed under the Tom1 account can be available to Tom2 only if they are available to the entire world. Multiple personalities lead to a proliferation of accounts, redundant files, limited protection for files of general interest, and inconvenience to users.

- All groups . To avoid multiple personalities, the system administrator may decide that Tom should have access to all his files any time he is active. This solution puts the responsibility on Tom to control with whom he shares what things. For example, he may be in Group1 with Ann and Group2 with Bill. He creates a Group1 file to share with Ann. But if he is active in Group2 the next time he is logged in, he still sees the Group1-file and may not realize that it is not accessible to Bill, too.
- Limited sharing . Files can be shared only within groups or with the world. Users want to be able to identify sharing partners for a file on a per-file basis, for example, sharing one file with ten people and another file with twenty others.

Single Permissions

In spite of their drawbacks, the file protection schemes we have described are relatively simple and straightforward. The simplicity of implementing them suggests other easy-to-manage methods that provide finer degrees of security while associating permission with a single file.

Password or Other Token

We can apply a simplified form of password protection to file protection by allowing a user to assign a password to a file. User accesses are limited to those who can supply the correct password at the time the file is opened. The password can be required for any access or only for modifications (write access).

Password access creates for a user the effect of having a different "group" for every file. However, file passwords suffer from difficulties similar to those of authentication passwords:

- Loss . Depending on how the passwords are implemented, it is possible that no one will be able to replace a lost or forgotten password. The operators or system administrators can certainly intervene and unprotect or assign a particular password, but often they cannot determine what password a user has assigned; if the user loses the password, a new one must be assigned.
- Use . Supplying a password for each access to a file can be inconvenient and time consuming.
- Disclosure . If a password is disclosed to an unauthorized individual, the file becomes immediately accessible. If the user then changes the password to reprotect the file, all the other legitimate users must be informed of the new password because their old password will fail.
- Revocation . To revoke one user's access right to a file, someone must change the

password, thereby causing the same problems as disclosure.

Temporary Acquired Permission

The Unix operating system provides an interesting permission scheme based on a three-level user "group "world hierarchy. The Unix designers added a permission called set userid (suid) . If this protection is set for a file to be executed, the protection level is that of the file's owner , not the executor . To see how it works, suppose Tom owns a file and allows Ann to execute it with suid . When Ann executes the file, she has the protection rights of Tom, not of herself.

This peculiar-sounding permission has a useful application. It permits a user to establish data files to which access is allowed only through specified procedures.

For example, suppose you want to establish a computerized dating service that manipulates a database of people available on particular nights. Sue might be interested in a date for Saturday, but she might have already refused a request from Jeff, saying she had other plans. Sue instructs the service not to reveal to Jeff that she is available. To use the service, Sue, Jeff, and others must be able to read and write (at least indirectly) the file to determine who is available or to post their availability. But if Jeff can read the file directly, he would find that Sue has lied. Therefore, your dating service must force Sue and Jeff (and all others) to access this file only through an access program that would screen the data Jeff obtains. But if the file access is limited to read and write by you as its owner, Sue and Jeff will never be able to enter data into it.

The solution is the Unix SUID protection. You create the database file, giving only you access permission. You also write the program that is to access the database, and save it with the SUID protection. Then, when Jeff executes your program, he temporarily acquires your access permission, but only during execution of the program. Jeff never has direct access to the file because your program will do the actual file access. When Jeff exits from your program, he regains his own access rights and loses yours. Thus, your program can access the file, but the program must display to Jeff only the data Jeff is allowed to see.

This mechanism is convenient for system functions that general users should be able to perform only in a prescribed way. For example, only the system should be able to modify the file of users' passwords, but individual users should be able to change their own passwords any time they wish. With the SUID feature, a password change program can be owned by the system, which will therefore have full access to the system password table. The program to change passwords also has SUID protection, so that when a normal user executes it, the program can modify the password file in a carefully constrained way on behalf of the user.

Per-Object and Per-User Protection

The primary limitation of these file protection schemes is the ability to create meaningful groups of related users who should have similar access to one or more data sets. The access control lists or access control matrices described earlier provide very flexible protection. Their disadvantage is for the user who wants to allow access to many users and to many different data sets; such a user must still specify each data set to be accessed by each user. As a new user is added, that user's special access rights must be specified by all appropriate users.

Authentication (identification).

Identification is the way you tell the system who you are. Authentication is the way you prove to the system that you are who you say you are. In just about any multi-user system, you must identify yourself, and the system must authenticate your identity, before you can use the system. There are three classic ways in which you can prove yourself:

1. Something you know. The most familiar example is a password. The theory is that if you know the secret password for an account, you must be the owner of that account. There is a problem with this theory: You might give your password away or have it stolen from you. If you write it down, someone might read it. If you tell someone, that person might tell someone else. If you have a simple, easy-to-guess password, someone might guess it or systematically crack it.
2. Something you have. Examples are keys, tokens, badges, smart cards, cell phones, etc you must have to "unlock" your terminal or your account. The theory is that if you have the key or equivalent, you must be the owner of it. The problem with this theory is that you might lose the key, it might be stolen from you, or someone might borrow it and duplicate it. Electronic keys, badges, and smart cards are gaining acceptance as authentication devices and as access devices for buildings and computer rooms. With the proliferation of automatic teller machines (ATMs), people are becoming increasingly familiar with this type of authentication.
3. Something you are. Examples are physiological or behavioral traits, such as your fingerprint, handprint, retina pattern, voice, signature, or keystroke pattern. Biometric systems compare your particular trait against the one stored for you and determine whether you are who you claim to be. Although biometric systems occasionally reject valid users and accept invalid ones, they are generally quite accurate. *The problem with these authentication systems is that, on the whole, people aren't comfortable using them.*

Passwords are still the authentication tool of choice. Even when authentication devices like tokens and biometric devices are used, they're usually supplements no replace conventional login IDs and passwords. Biometrics and smartcards typically act only as a first line of defense against intruders, not as the only defense.

The process of establishing that you are the person you are trying to represent is called authentication. In other words, the authentication is a means of access control that ensures that user are who he claim to be. Every person who uses a Unix machine has own *account* identified by a user ID number (UID) that is associated with one or more *usernames* (or *account names*). Each account also has a secret *password* associated with it to prevent unauthorized use. You need to know both your username and your password to log in

Something u know,
Something u have
Something u are.

- Username in most organization are usually generated from user first and last name. For example first six characters of the last name, plus the first character of the first name and the first character of the middle name or digit in case of conflicts create eight character user name. For users that have identical first several characters of user id, the eight character can be selected to be unique (for example 1-9). Traditionally the length of the Unix user name is limited to eight characters although some systems now may allow longer names without implicit truncation to first eight meaningful characters.
- Passwords are the simplest form of authentication; and it is important not to kill simplicity with too much zeal. They are a secret string of characters that allow you to login. Older flavors of Unix used to limit passwords to 8 characters. In this case it difficult to use very effective and easily memorizable AOL-style double word scheme for password (e.g. nj0777-florham, Dogs&&Cats, Geo-Prism-1.6). When you log in, you type your password to prove to the computer that you are who you claim to be. The computer encode some string of characters (blanks in case of Unix) with your password and compare the result with the stored value. It never stores the password on the server.

Authentication can be performed not only via passwords, but also using certificates, security tokens (SecurID is one example) or any combination of those (two factor authentication). SMS messages are widely used in Eastern Europe and Russia. Two factor authentication is now standard for any financial data (ETrade and Paypal provide users with free security tokens) to minimize the possibility for unauthorized persons to gain access to the account. Using security token makes "phishing attacks" useless as one time password is unique for each logging attempt.

Web sites, especially financial web sites are even more difficult to secure as here *there are clear incentives to break-in.* Actually financial web site like Vanguard are in forefront of implementing simple but effective control measures mentioned above. To avoid "fake sites" companies like Vanguard implemented *a random, user-selectable "mascot"* for each user, which make each login page unique and substantially complicated creation of fake sites.

Passwords remain the simplest and the most common form of authentication, which now should be abandoned in favor or two factor authentication with the sell phone providing security code or, better, a hardware token. With regular passwords, the most important thing is not strength of the password per ser (although it should satisfy some minimal requirements), but three additional measures that help to block dictionary style attacks:

1. Assume that there is hunting season for large commercial sites accounts from various intelligence agencies and hackers.
2. Each server or site should have a unique password. Even simple scheme like the first word reflecting the current changeable password and the second word reflecting some information about the server or site. For example DL380g7up for a server. for Web site you can use location or some word associated with the state

(team, famous company, capital, etc) as the second part of password (for example "Msoft" for Californian sites or "Devils" for NJ , or even zip07929 -- zip code of location). But now in no way you should ever use the same password for two servers or site.

3. Length of password is much more important then its strength. Enforcing length of, say 10 or even 12 (two 6 symbols words) you essentially force the users (and sysadmins) to use two words combinations (AOL-style) for password. Which are the only way to diminish chances that you password will be cracked if a file that contains hashes is stolen. See Password Policy in 'After Snowden" World
4. Limiting number of failed login attempts to, say, 5, 7 or, even, 11 (but not 3 -- unless you want to create a stream of useless helpdesk tickets for your organization). Brute force attack with just seven attempts is meaningless.
5. Increasing waiting period after each failed login attempt or series of attempts. This is a must in "After Snowden" world.
6. Controlling IPs from which login is performed and using such an IP as an additional authentication parameter or even as a hidden part of password. If the IP address from which authentication is performed is "new" --- has no history of previous successful authentication attempts, then additional questions should be asked before allowing access. This is especially important in case of Web access. Most financial sites now use "pre-authentication" for any "new" IP: they first ask set of predefined questions about account owner history (maiden name of your mother, name of High School that you attended, brand of your first car, etc). Only if you answered those questions correctly they allow you to enter password.
7. Using unique password for each account you have. In real life if somebody steals the shadow file from some server or database of account from a cloud provider or Web merchant the only realistic goal in this context is pretty evil: *to reuse cracked accounts for attacking other accounts.* Users and sysadmins often have the same password on multiple servers/sites; and in "after Snowden" world this is a very bad idea -- *you need to have some algorithm of individualization of passwords for each account you have* -- no more any two identical passwords can be used. Now this is a must.

Users and sysadmins often have the same password on multiple servers; and in "after Snowden" world this is a very bad idea -- you need to have some algorithm of individualization of passwords for each account you have -- no more any two identical passwords can be used. Now this is a must.

8. Especially bad idea is to have non-unique password for accounts with financial information.

Authentication is not limited to servers and web sites, of course. Network equipment like routers proved to be particularly difficult to secure and often is a point of attack.

Contrary to popular belief, Unix passwords cannot be decrypted from the content of `/etc/shadow` file, even if the file was stolen. Unix passwords are encrypted with a one way function. The login program accepts the text you enter at the "Password:" prompt and

then encrypt predefined string using specified cryptographic algorithm. The results of that algorithm are then compared against the encrypted form of your Unix password stored in the `/etc/shadow` file. Unfortunately not all cloud providers and Web merchants use this algorithms. Some store "real" passwords which is a blunder.

On a more technical level, the password that you enter is used as a key to encrypt a 64-bit block(s) of NULLs. The first seven bits of each character are extracted to form a 56-bit key. This means that only eight characters are significant in a standard Unix password (Modern Unixes switched to MD5 hashes; Linux now implemented longer passwords using MD5 hashes). The E-table is then modified using the salt, which is a 12-bit value, coerced into the first two chars of the stored password. The salt's purpose is to make precompiled password lists more time consuming to use. DES is then invoked for 25 iterations. The 64-bit output block is then converted to a 64-character alphabet (A-Z,a-z,".","/") string.

Unix password auditing software (aka Password Checkers/Crackers) uses wordlists to implement a dictionary attack against `/etc/shadow` file. *As such this attacks can be replicated only if hacker managed to stole* `/etc/shadow` *file.* But stealing it means that you should be root on the particular server. So there is a *Catch 22* situation.

Generally cracking passwords makes sense if and only if users reuse passwords on different servers. So the first rule here is not increasing the strength of the password to absurd levels, but requiring that user do not use the same password for accounts on different servers by adding server specific suffix or prefix much like salt in Unix password encryption scheme. Moreover, any password used for web site should never be used to secure your financial data. Good practice requires using three mnemonic schemes for passwords:

- simple on public servers (web sites, etc; here password can be short and easy),
- more complex for private servers (passwords should be longer -- two word type)
- preferably two factor authentication on servers with financial data. If single password is used it should be changed regularly and should have lower and upper case characters and at least one special character to increase size of alphabet). It is very dangerous (and stupid) to use the same password on two different financial sites. Keeping an encrypted spreadsheet of your passwords is much safer then that.

If passwords are unique for each server and number of login attempts is limited and after certain amount of attempts account is blocked for an hour or so even very weak passwords are "unbreakable" from a theoretical standpoint. *So excessive zeal in choosing "extra-secure" passwords often backfires as users often forget them and tend to use the same password on different servers. Using the same password for two or more different servers is a big "no-no" in "After Snowden" world.* Similar to use of non-encrypted protocol for connection. Essentially FTP and Telnet now needs to be outlawed for business critical servers.

Excessive zeal in choosing strong random passwords often backfire. Now you are better off using longer easily memorizable passwords for each server (AOL two word approach or similar is a good idea) than one strong password for all you accounts on multiple servers. In "After Snowden" world this is a big no-no.

When strong passwords are really needed two factor authentication should be used

To ensure better password security your can use the following measures (not all of them make sense in the particular organization but some definitely make sense):

1. Always enable the number of unsuccessful login protection, but do not set it too low (7 is OK, but 3 is too low; 5 is in between, but still pretty annoying as for number of helpdesk tickets generated). If you know how to use PAM instead of disabling the account *it is better to double the pause after each unsuccessful login attempt* or series of attempts. Implementation of "known IPs" feature requires custom PAM module, but it is a very effective and highly recommended measure. It prevents brute force attacks from "zombie" PCs and at the same time does not inconvenience users. Together those two measures limit typical for large organizations flow of useless tickets to helpdesk from users who lock themselves out, while still providing reasonable level of security.

2. It is important not to overdo increasing simple password security. If higher level of security is required, then the implementation of certificate, security token or card is a better solution that any tinkering with plain vanilla passwords. Again one of the key consideration in implementing any simple password related security measure is its effect on the on flow of tickets to the helpdesk. In "After Snowden" world smartcards are now the standard way to increase the level of password security in most large organizations.

3. Root password should have complex long password or no password (hash that can't be generated by the chosen encryption scheme, for example string "NA") at all (in this case SSH certificate still should be used instead). You can always use sudo on Linux or RBAC (on Solaris) to get to root from a regular account and this provides additional level of security. Do not allow telnet root logins. SSH root logins should require certificates. This is probably the simplest way to increase root password security. The other trick is completely disable root login and use sudo to get to root. That instantly creates opportunity to limit root access to selected group of people without playing with PAM. Right now only Ubuntu has sudo enabled by default and root password is disabled, but these scheme can be adopted in all major Linux distributions as sudo is now preinstalled on all of them (included in default installation and is supported package). Please remember that on most servers with additional remote control cards, remote control password is equivalent to root and would be protected with the same level of attention. Often organization are pretty lax and stupid in this area.

4. Inform users about "good" password selection criteria, prevent too short passwords (for example, less then 6 characters for regular users, 8 characters for root). It make sense to use a pro-active password checker to verify that passwords are compliant with the chosen policy. The key here is to use the ability to select longer AOL-style two word passwords. Don't overdo complexity of password criteria for example one non alphanumeric character is usually enough): too complex requirements backfire as complex passwords are more often forgotten by users and require help desk intervention to reset. They often written by users and instead of improving compromise security. For example, random generated passwords are justifiable mainly for standard accounts like root (and even in this case only half of the password (3 or four letter combination) should be random as longer random strings are really difficult to remember, see Root Security)
5. For root password always use at least eight characters as the minimum length. It large organizations it makes sense use a password generator which generates "good" root passwords and reset them automatically each quarter or even more often. Casio Watches can be used as a databank of critical passwords (but never any smartwatches). The key problem here is that when people leave organization it is not always possible to know correctly to which systems they have root access.
6. For regular users implement password aging. Interval should be long enough, for example 90 or 180 day days to make this measure less troublesome, but there should be an interval.
7. Place expiration dates on, and periodically "expire", all "human" accounts to eliminate unused accounts. The period should be long enough not to affect regular users (for example, one year). If this is not done then often accounts for users who left three years still exist on some systems no matter what measures you take and what audits were performed.
8. If you need to use guest accounts, use restricted shells for them.
9. Eliminate "default" guest accounts. They should have unique for the organization names like `mgb_guest`
10. Disable the ability to log in to system "placeholder" accounts like sys by using "fake" shell (like nologin). Delete well-known accounts that are not needed
11. Make sure all accounts have passwords or "*" in the password field and automatically check for this daily (all system accounts with UID less then from 1 to 100 generally should have "*" in the password filed of shadow file.
12. Use wheel group and corresponding PAM module for users who are allowed to switch to root.
13. Make sure default file protections for newly created files. Do not allow group/world read/write access by using a "`umask`" value of 022, 027, or 077.
14. Codify rules and policies for operating as the "super user" Monitor compliance with those use using automatic tools. Do not adopt rules that can't be automatically checked.
15. Write-protect the "root" account's startup files and home directory
16. Minimize the number of accounts on servers and "critical" hosts
17. Minimize the number of users with "super user" privileges

18. Periodically check from cron soundness of accounts settings using automatic tools. The simplest such tools is probably pwckcommand; better tools are scripts from hardening tools like Titan, JASS or Bastille

CH 4

Security Models

- A security policy is a document that expresses clearly and concisely what the protection mechanisms are to achieve. Its a statement of the security we expect the system to enforce.
- A security model is a specification of a security policy:
  - it describes the entities governed by the policy,
  - it states the rules that constitute the policy.
- There are various types of security models:
  - Models can capture policies for confidentiality (Bell-LaPadula) or for integrity (Biba, Clark-Wilson).
  - Some models apply to environments with static policies (Bell-LaPadula), others consider dynamic changes of access rights (Chinese Wall).
  - Security models can be informal (Clark-Wilson), semi-formal, or formal (Bell-LaPadula, Harrison-Ruzzo-Ullman).
- Model vs Policy
  - A security model maps the abstract goals of the policy to information system terms by specifying explicit data structures and techniques that are necessary to enforce the security policy. A security model is usually represented in mathematics and analytical ideas, which are then mapped to system specifications, and then developed by programmers through programming code
  - For Example, if a security policy states that subjects need to be authorized to access objects, the security model would provide the mathematical relationships and formulas explaining how x can access y only through the outlined specific methods
  - A security policy outlines goals without regard to how they will be accomplished. A model is a framework that gives the policy form and solves security access problems for particular situations.

Lattice Models

- A lattice is a mathematical construct that is built upon the notion of a group.
- A lattice is a mathematical construction with:
  - a set of elements
  - a partial ordering relation
  - the property that any two elements must have unique least upper bound and greatest lower bound → (sides, dimensions).
- A security lattice model combines multilevel and multilateral security
- Lattice elements are security labels that consist of a security level and set of categories

State Machine Models

- In state machine model, the state of a machine is captured in order to verify the security of a system.
- A given state consists of all current permissions and all current instances of subjects accessing the objects. If the subject can access objects only by means that are concurrent with the security policy, the system is secure.
- The model is used to describe the behavior of a system to different inputs. It provides mathematical constructs that represents sets (subjects, objects) and sequences. When an object accepts an input , this modifies a state variable thus transiting to a different state.
- Implementation tips
    - The developer must define what and where the state variables are.
    - The developer must define a secure state for each state variable.
    - Define and identify the allowable state transition functions.
    - The state transition function should be tested to verify that the overall m/c state will not compromise and the integrity of the system is maintained.

Noninterference Models

- The model ensures that any actions that take place at a higher security level do not affect, or interfere with, actions that take place at a lower level.
- It is not concerned with the flow of data, but rather with what a subject knows about the state of the system. So if an entity at a higher security level performs an action, it can not change the state for the entity at the lower level.
- The model also addresses the inference attack that occurs when some one has access to some type of information and can infer(guess) something that he does not have the clearance level or authority to know.

Bell—LaPadula Confidentiality Model

- It was the first mathematical model with a multilevel security policy that is used to define the concept of a secure state machine and models of access and outlined rules of access.
- It is a state m/c model that enforces the confidentiality aspects of access model.
- The model focuses on ensuring that the subjects with different clearances(top secret, secret, confidential) are properly authenticated by having the necessary security clearance , need to know , and formal access approval-before accessing an object that are under different classification levels (top secret, secret, confidential).
- The rules of Bell-Lapadula model
    - Simple security rule (no read up rule): It states that a subject at a given security level can not read data that resides at a higher security level.
    - Star property rule (no write down rule): It states that a subject in a given security level can not write information to a lower security levels.

- Strong star property rule: It states a subject that has read and write capabilities can only perform those functions at the same security level , nothing higher and nothing lower.
- *Tranquility principle* : subjects and objects can not change their security levels once they have been instantiated (created).
- All MAC systems are based on the Bell – Lapadula model because of it multilevel security.
- Designed US govt and mostly adopted by govt agencies

Biba Integrity Model

- It is developed after Bell – Lapadula model.
- It addresses integrity of data unlike Bell – Lapadula which addresses confidentiality.
- It uses a lattice of integrity levels unlike Bell – Lapadula which uses a lattice of security levels.
- It is also an information flow model like the Bell – Lapadula because they are most concerned about data flowing from one level to another.
- The rules of Biba model
    - simple integrity rule(no read down) : it states that a subject can not read data from a lower integrity level.
    - star integrity rule(no write up) : it states that a subject can not write data to an object at a higher integrity level.
    - invocation property : it states that a subject can not invoke(call upon) a subject at a higher integrity level.

Clark—Wilson Integrity Model

- It was developed after Biba and addresses the integrity of information.
- This model separates data into one subject that needs to be highly protected , referred to as a constrained data item(CDI)and another subset that does not require high level of protection , referred to as unconstrained data items(UDI).
- Components
    - Subjects (users): are active agents.
    - Transformation procedures (TPs): the s/w procedures such as read, write, modify that perform the required operation on behalf of the subject (user).
    - Constrained data items (CDI): data that can be modified only by Tp's.
    - Unconstrained data items (UDI): data that can be manipulated by subjects via primitive read/write operations.
    - Integrity verification procedure (IVP): programs that run periodically to check the consistency of CDIs with external reality. These integrity rules are usually defined by vendors.
- Integrity goals of Clark – Wilson model
    - Prevent unauthorized users from making modification (addressed by Biba model).

- Separation of duties prevents authorized users from making improper modifications.
- Well formed transactions: maintain internal and external consistency i.e. it is a series of operations that are carried out to transfer the data from one consistent state to the other.

Access Control Matrix

- This model addressed in access control.
- Commonly used in OS and applications.

Information Flow Models

- In this model, data is thought of as being held in individual discrete compartments.
- Information is compartmentalized based on two factors.
  - Classification and
  - Need to know
- The subjects clearance has to dominate the objects classification and the subjects security profile must contain the one of the categories listed in the object label, which enforces need to know.
- For example:
  - Bell – Lapadula which prevents information flowing from higher source level to lower source level.
  - Biba which prevents information flowing from lower integrity level to higher integrity level

Covert channels

- A covert channel is a way for an entity to receive information in an unauthorized manner.
- It is an information flow that is not controlled by a security mechanism.
- It is an unauthorized communication path that is not protected by the system because it was uncovered while developing the system.
- Types of covert channels
  - Covert timing: in this channel, one process relays information to another by modulating its use of system resources.
  - Covert storage: in this channel, one process writes data to a storage location and another process directly, or indirectly reads it.

Graham—Denning Model

- This model defines a set of basic rights in terms of commands that a specific subject can execute on an object.
- It proposes the eight primitive protection rights, or rules of how these types of functionalities should take place securely.

- How to securely create an object.
- How to securely create a subject.
- How to securely delete an object.
- How to securely delete a subject.
- How to provide read access rights.
- How to provide grant access rights.
- How to provide delete access rights.
- How to provide transfer access rights.

Harrison—Ruzzo—Ullman Model

- The HRU security model (Harrison, Ruzzo, Ullman model) is an operating system level computer security model which deals with the integrity of access rights in the system. The system is based around the idea of a finite set of procedures being available to edit the access rights of a subject s on an object o.
- The model also discussed the possibilities and limitations of proving safety of a system using an algorithm.

Brewer—Nash (Chinese Wall)

- This model provides access controls that can change dynamically depending upon a user's previous actions.
- The main goal of this model is to protect against conflicts of interests by user's access attempts.
- It is based on the information flow model, where no information can flow between subjects and objects in a way that would result in a conflict of interest.
- The model states that a subject can write to an object if, and only if, the subject can not read another object that is in a different data set.

Access Control

The access control is just another name for compartmentalization of resources.

It is useful to group general problems involved in making certain that files are not read or modified by unauthorized personnel under common umbrella -- *access control*. There are two aspect of access control:

1. Access control policies. Access control policy defined "whose data is to be protected from whom"
2. Access control mechanisms.the manner by which the operating system enforces the access control policy. Among them the following are the most important,
   - Accounts securitymechanisms
     - Root Security mechanisms
   - Classic Unix permissions model
   - Unix ACLs.
   - [redacted]

Classic Unix systems have at least one user with right to access (privilege) any file of the system -- root user. A protection domain is defined by its `UID` and `GID`. Provided with any (`UID`, `GID`) combination it is possible to build a complete list of all objects that can be accessed and each objects rights. Two processes with the same (`uid`, `gid`) combination, have access to exactly the same set of objects. Processes with different (`uid`, `gid`) combinations, have access to a different set of files.

Additionally, in UNIX each process has two halves: the user part and the kernel part. When a process does a system call it switches from the user role to the kernel role. The kernel role has access rights to a different set of objects from the user part. A system call therefore invokes a domain switch, from user to kernel.

At every instant in time each process runs in a protection domain. Therefore, there is some collection of objects that the process can access. In addition, each object has a set of access rights. Processes can switch from domain to domain during program execution.

The UNIX process division into user and kernel parts is a legacy of a more powerful domain switching mechanism that was used in MULTICS. In the MULTICS system the hardware supported up to 64 domains for each process, not the two (kernel and user) like in UNIX or MVS. A MULTICS process could be considered as a collection of procedures, each running in a domain, called a ring. The innermost ring was the most powerful, the operating system kernel. Radically, moving outwards from the operating system kernel the rings became successively less powerful. When a procedure in one ring called a procedure in a different ring a trap occurred. Once a trap occurred the system had the option to change the protection domain for that process.

(Linux)

Unix permissions model

Unix introduced a simple model of file permissions in the 70's which has proven to be quite effective and easy to understand. In this model each file has three attributes for each of three access categories (owner, group and world):

- Read
- Write
- Execute

There is also one implicit attribute: *the ability to delete file/directory*. It is property not a file/directory in question, but the directory in which it contained. It the user has write access to the directory he/she can delete any file in it.

Moreover the execution bit on the directory can work as partial access blocker to files in this directory. It in is not set there is no way to obtain a listing of the directory althouth if you know file name you can access file that is contains in the directory.

Due to popularity of Samba, recently there was a half-baked attempt to extend Unix "xwr" model using ACLs, that are used in Windows.

The result is a mess and very few organization have adopted this approach on Unix servers due to the complexity and bad integration with classic Unix model. Not only existence of the second, by-and-large, indeoendent model confuse administrators, it actually represents a serious security risk.

In the original Unix model, each file has three "rwx" access categories: *User (u)*, *Group (g)* and *Other (o)*. Group is essentially a role and primary group is a primary role. Any user can be a member of any number of groups, but in Unix groups are atomic -- they cannot contain other groups. This is a serious shortcoming of the classical Unix model.

System groups and pseudo-groups

There is also auxiliary concept of system groups which is similar to the concept of privileged ports. For example, all groups with GID below 100 are usually considered to be system groups. Some system groups are designed mainly for not to providing access to files for a group of users, but for partitioning of permission space. Among them are adm, sys, daemon, lp, mail, uucp, games, ftp. nobody, etc. Typically users which have such "pseudo-group" as primary group have no legitimate shell. Instead `/bin/noshell` or `/bin/false` is used; the former logs the access attempts), so nobody can login as such a user. They can be called pseudo-users.

Another severe limitation of this model is that a file can only be a member of one group. That can be partially rectified by usage of "metagroups" -- groups that are just

aggregations of existing groups, but that solution requires additional efforts and discipline (in this case /etc/group needs to be automatically generated from some template using macroprocessor). If we assume that the number of group allowed is large (approximately the same as the number of files/directories) metagroup approach is as powerful as ACL model and is much simpler. It requires relatively simple modification of the /etc/group file.

User Private Groups (UPG)

One interesting step toward this model is the concept of User Private Groups (UPG) introduced in Red Hat. UPG scheme makes UNIX groups easier to use. It does not add or change anything in the standard UNIX way of handling groups. This is simply a new convention for handling classic Unix groups groups: whenever you create a new user, by default, he or she has a unique group with GID identical to UID.

One of limitations of the Unix model is that ordinary users do not have the right to make their own groups. In this way, the model shoots itself in the foot. There are ways around this, but Unix has not introduced a standard solution to the problem. We can think of the Unix model as being a coarse approximation to the model of ACLs.

For more information see User Private Groups

Inheritance of permissions

One of the most interesting aspects of file security of a particular OS is the mechanism by which security attributes are inherited by new files. Newly created files have to start out with some kind of permissions and that this "default set" of permissions is a question that is answered differently by different flavors of Unix. One advance implemented by BSD is usage of setgid bit in directories to inherit permissions set for the directory for all new files created in it.

Full inheritance of permissions is possible only under root account. For user account inheritance is restricted. Permissions granted are further restricted by the current value of `umask` variable

Most systems are hybrid and support BSD behavior via setgid bit in directories.

- BSD: New files are owned by the creator of the file, default group set as primary group for the owner of of the file (from password file). Permissions are determined from the value of `umask`.
- SysV: New files are owned by the creator of the file, default group set either from a password file or inherited from the group of the parent directory (if `setgid` bit set on parent directory). Permissions are determined from the value of `umask`. So it contains one easily understood extension of BSD model.
- POSIX ACLs: This model is almost impossible to understand. It is badly integrated with classic Unix permissions as as such is more or the source of additional security vulnerabilities then additional security. Directories work as in classic Unix, but files

with ACLs inherit the ACLs of their parent directory. The default permissions set on files are set from umask, but the *effective* permissions are calculated from combination of the Unix permissions, the ACL permissions and an internal ACL mask. That's very confusing. Only the owner of a file can modify the ACL. Not even root can modify another user's ACL.

- NTFS ACLs: ACLs are inherited from parent objects unless overidden. New files with no ACL are open to everyone. The exact inheritance mechanism is not well-documented. The NTFS filesystem the whole access model is based on *ACLs*. Each file is owned by a specific user. Each user may belong to any number of groups. Each file has an ACL in which access can be *granted* or *denied* to any named list of users and/or groups (i.e. we can either code a rule or an exception to a rule). The access rights of each group or user are taken from the set:
  - Read
  - Write
  - Execute
  - Delete
  - Change permission
  - Change ownership

Note that some of the permissions restrict functionality (execute permission), others restrict access. The means that read permissions to a certain extent overwrite the execute permission: the execute permission protection is limited to a particular copy of a file. Any new copy created due to availability of read permissions can have execute permissions for the domain associated with the current owner. In other words protection against execution presuppose absence of read attribute.

Windows ACLs are closer to the original Multix implementation. Windows 2000 and later provides 14 ACL attributes (In the table below, the # character means this flag is selected only when the Full Control flag is set.):

ACL	Attribute
Full Control	#
Traverse Folder/Execute File	x
List Folder/Read Data	r
Read Attributes	r
Read Extended Attributes	r
Create Files/Write Data	w
Create Folders/Append Data	w
Write Attributes	w
Write Extended Attributes	w
Delete Subfolders and Files	w
Delete	#
Read Permissions	all

Change Permissions #
Take Ownership #

Process permissions

Access rights apply not only to secondary storage but to any resource. In particular, the right to communicate with, control or request services from a process. The same basic ideas apply, but access controls usually have different attributes. Process permissions are usually set by access control lists, or on the basis of understood protocols, such as passwords, keys, or *cookies* (the message in a fortune cookie).

Linux

Unix processes have an owner and a group membership. These are normally inherited from the the user who starts the process and from the group attribute of the program file respectively, (though see below about setuid/setgid programs). Each process has the privileges afforded to it by these labels.

Kernel two modes operation

The basic security of multiuser operating systems lies in the ability to restrict privilege. One of our design criteria was to try to prevent malicious users from circumventing security mechanisms. In some operating systems this has been possible by accidental or deliberate memory corruption. Since software is often buggy, we should not rely on software to behave properly. Stronger measures are needed to protect the system. This was the idea behind two-mode operation (see the operating system course notes). By having system protection hard-coded in hardware, the system is more secure. The Multics operating system generalized this idea to more than two-modes. A series of protection levels or rings was used, each of which encapsulated the inner rings. A typical use of protection rings would be the following:

- Ring 0: Operating system kernel
- Ring 1: Operating system
- Ring 2: Utilities
- Ring 3: User processes

In Unix's two-mode operation, only rings 0 (kernel) and 3 (user processes) are used. This has made it easy to port Unix to a variety of architectures which support 2-mode operation (e.g PC's including and later than i386).

Role based permissions

In a dynamical, interactive situation we could generalize the notion of access to allow users to use different permissions depending on what they are doing. This increase complexity of the model and can be done in several different ways:

- By asking a service to carry out an operation to run under user account whose abilities have the appropriate privileges for the task (e.g. `oracle` account, `ftp`, `www`,

java accounts). This approach is limited by severe limitations that regular Unix account have.

- By introducing umask or permissions for key files and or directories on per process basis.

If you think about it global static permission used in Unix are anachronism. There should be a mechanism that prevent process from accessing files in, say, /etc directory even if part of the process is running under the root privileges. This can be done in several different ways:

- You can separate certain parts of the tree using concept of disk partitions. This approach which is very transparent and pretty powerful is called "protective partitioning". By using mounting command restrictions you can also prohibit setting setuid/setgid attributes and even executable permissions on the files in this partition.
- You can use invented by FreeBSD concept of immutable files (which is now available in Linux too).
- You can use more complex dynamic approaches such as Solaris RBAC, SuseApparmor and RHEL CElinux. But here the key problem is the complexity of the approach. They dramatically increase complexity of the model often putting it beyond typical sysadmin ability to comprehend the model. Based on complexity we can define two group of such approaches:
    - Models, which a regular sysadmin can with effort still understand and use (albeit with difficulties and errors) such as Solaris RBAC and SuseAppArmor. The latter is a rather elegant model of extending the concept of umask to per directory and per file basis. It used regular expression for determining "effective permissions" for the process. Along with Suse this scheme is used in Ubuntu since April of 2007.
    - Approaches which regular sysadmin typically can't understand and is inclined never use such as RHEL SElinux

- By setting some attribute which increases the allowed permissions for a given task. (e.g Unix setuid programs). *Unix*setuid*programs* are an example where the privileges of a program can be changed (typically increased) by the superuser. They grant a specific program the right to operate with a different user-identity and thus privileges without authentication. The setgid is a corresponding mechanism for setting a different group ownership of a process. Note that typically setuid programs acquire more privileges than is necessary (acquiring root privileges is the most common usage of setuid/setgid attributes) and because of that have been the major cause of security problems on Unix platforms. See SUID/SGID attributes for more information.
- Use of OO language-style inheritance models (e.g. polymorphic methods in object oriented languages)