Module 9: OS and Security

Computers, with their ubiquitous presence, have ceased to be a wonder they once were. Their usage is pervasive. Information access and delivery from, and to, a remote location via internet is common. Today many societal services like railway time-table or election results are rendered through computers. The notion of electronic commerce has given fillip to provisioning commercial services as well. Most individuals use computers to store private information at home and critical professional information at work. They also use computers to access information from other computers anywhere on the net. In this kind of scenario, information is the key resource and needs to be protected.

The OS, being the system's resource regulator, must provide for security mechanisms. It must not only secure the information to protect the privacy but also prevent misuse of system resources. Unix designers had aimed to support large-scale program development and team work. The main plank of design was flexibility and support tools. The idea was to promote creation of large programs through cooperative team efforts. All this was long before 9/11. Security has become a bigger issue now. Much of Unix provisioning of services was with the premise that there are hardly, if any, abuses of system. So, Unix leaves much to be desired in respect of security. And yet, Unix has the flexibility to augment mechanisms that primarily protect users resources like files and programs. Unix incorporates security through two mechanisms, user authentication and access control. We shall elaborate on both these aspects and study what could be adequate security measures. We begin with some known security breaches. That helps to put security measures in proper perspective.

9.1 Security Breaches

We first need to comprehend the types of security breaches that may happen. Breaches may happen with malicious intent or may be initiated by users inadvertently, or accidentally. They may end up committing a security breach through a mis-typed command or ill understood interpretation of some command. In both these instances the OS must protect the interest of legitimate users of the system. Unix also does not rule out a malicious access with the intent to abuse the system. It is well known that former disgruntled employees often attempt access to systems to inflict damages or simply corrupt some critical information. Some malicious users' actions may result in one of the following three kinds of security breaches:

1. Disclosure of information.
2. Compromising integrity of data.
3. Denial of service to legitimate users of the system.

To launch an attack, an attacker may correctly guess a weak password of a legitimate user. He can then access the machine and all HW and SW resources made available to that user. Note that a password is an intended control (a means to authenticate a user) to permit legitimate access to system resources. Clearly, a malicious user may employ password racking methods with the explicit intent to bypass the intended controls. He may access classified information and may also misuse the system resources. An un authorized access ay be launched to steal precious processor cycles resulting in denial of service. Or, he may be able to acquire privileged access to modify critical files corrupting sensitive data. This would be an act of active misuse. Some activities like watching the traffic on a system or browsing without modifying files may be regarded as an act of passive misuse. Even this is a breach of security as it does lead to disclosure. It may result in some deterioration, albeit not noticeable, in the overall services as well.

9.1.1 Examples of Security Breaches

Here we shall discuss a few well known attacks that have happened and have been recorded. Study of these examples helps us to understand how security holes get created. Besides, it helps us to determine strategies to plug security holes as they manifest. Next we describe a few attack scenarios. Not all of these scenarios can be handled by OS control mechanisms. Nonetheless, it is very revealing to see how the attacks happen.

- **External Masquerading:** This is the case of unauthorized access. The access may be via a communication media tap, recording and playback. For instance, a login session may be played back to masquerade another user. The measures require a network-based security solution.
- **Pest Programs:** A malicious user may use a pest program to cause a subsequent harm. Its effect may manifest at some specified time or event. The Trojan horse and virus attacks fall in this category. The main difference between a Trojan horse and a virus is that, a virus is a self reproducing program. Some virus writers have used the Terminate and Stay Resident (TSR) program facility in Micro-soft environments to launch such attacks. The pest programs require internal controls

to counter. Generally, the time lag helps the attacker to cover the tracks. Typically, a virus propagation involves the following steps:

- **Remote copy:** In this step a program is copied to a remote machine.
- **Remote execute:** The copied program is instructed to execute. The step requires repeating the previous step on the other connected machine, thereby propagating the virus.
- **Bypassing internal controls:** This is achieved usually by cracking passwords, or using compiler generated attack to hog or deny resources.
- **Use a given facility for a different purpose:** This form of attack involves use of a given facility for a purpose other than it was intended for. For example, in Unix we can list files in any directory. This can be used to communicate secret information without being detected. Suppose `userB' is not permitted to communicate or access files of `userA'. When `userB' access files of `userA' he will always get a message permission denied. However, `userA' may name his files as atnine, tonight, wemeet. When `userB' lists the files in the directory of `userA' he gets the message “at nine tonight we meet", thereby defeating the access controls.
- **Active authority misuse:** This happens when an administrator (or an individual) abuses his user privileges. A user may misuse the resources advanced to him in good faith and trust. An administrator may falsify book keeping data or a user may manipulate accounts data or some unauthorized person may be granted an access to sensitive information.
- **Abuse through inaction:** An administrator may choose to be sloppy (as he may be disgruntled) in his duties and that can result in degraded services.
- **Indirect abuse:** This does not quite appear like an attack and yet it may be. For instance, one may work on machine `A' to crack a protection key on machine `B'. It may appear as a perfectly legal study on machine `A' while the intent is to break the machine `B' internal controls.

We next discuss the commonly used methods of attacks. It is recommended to try a few of these in off-line mode. With that no damage to the operating environment occurs nor is the operation of an organization affected.

- **The Password spoof program:** We consider the following Trojan horse and the effect it generates. It is written in a Unix like command language.

```
B1='ORIGIN: NODE whdl MODULE 66 PORT 12'
B2='DESTINATION:'
FILE=$HOME/CRYPT/SPOOFS/TEST
trap '' 1 2 3 5 15
echo $B1
sleep 1
echo ''
echo $B2
read dest
echo 'login:
read login
stty -echo
echo 'password:
read password
stty echo
echo ''
echo $login $passwd >> spooffile
echo 'login incorrect'
exec login
```

  The idea is quite simple. The program on execution leaves a login prompt on the terminal. To an unsuspecting user it seems the terminal is available for use. A user would login and his login session with password shall be simply copied on to spooffile. The attacker can later retrieve the login name and password from the spooffile and now impersonate the user.

- **Password theft by clever reasoning:** In the early days passwords in Unix systems were stored in an encrypted form under /etc/password. The current practice of using a shadow file will be discussed later. So, in early days, the safety of password lay in the difficulty associated with decrypting just this file. So attackers used to resort to a clever way of detecting passwords. One such attack was through an attempt to match commonly used mnemonics, or use of

convenient word patterns. Usually, these are words that are easy to type or recall. The attacker generated these and used the encrypting function to encrypt them. Once the encrypted pattern matched, the corresponding password was compromised.

- **Logic Bomb:** A logic bomb is usually a set-up like the login spoof described earlier. The attacker sets it up to go off when some conditions combine to happen. It may be long after the attacker (a disgruntled employee for instance) has quit the organization. This may leave no trail. Suppose we use an editor that allows setting of parameters to OS shell, the command interpreter. Now suppose one sets up a Unix command rm *.* and puts it in a file called EditMe and sends it over to the system administrator. If the system administrator opens the file and tries to edit the file, it may actually remove all the files unless he opens it in a secure environment.

  Also, if the administrator attempts opening this as a user, damage would be less, compared to when he opens it as a root.
- **Scheduled File Removal:** One of the facilities available on most OSs is scheduled execution of a program or a shell script. Under Unix this is done by using at command. A simple command like : *rm -f /usr* at 0400 saturday attack This can result in havoc. The program may be kept in a write protected directory and then executed at some specified time. The program recursively removes files without diagnostic messages from all users under *usr.*
- **Field Separator Attack:** The attack utilizes some OS features. The following steps describe the attack :
  1. The attacker redefines the field separator to include backslash character so that path names such as /coo/koo are indistinguishable from coo koo.
  2. The attacker knowing that some system program, say *sysprog,* uses administrative privilege to open a file called /coo/koo creates a program coo and places it in an accessible directory. The program is coded to transfer privileges from the system to the user via a copied OS shell.
  3. The attacker invokes *sysprog* which will try to open /coo/koo with the administrative privileges but will actually open the file coo since the field

separator has been redefined. This will have the desired effect of transferring privileges to the user, just as the attacker intended.

- **Insertion of Compiler Trojan Horse:** To launch an attack with a very widely distributed effect an attacker may choose a popular filtering program based Trojan horse. A compiler is a good candidate for such an attack. To understand an attack via a compiler Trojan horse, let us first describe how a compiler works:

  Compile: get (line);
  Translate (line);

  A real compiler is usually more complex than the above description. Even this models a lexical analysis followed by the translating phases in a compiler. The objective of the Trojan horse would be to look for some patterns in the input programs and replace these with some trap door that will allow the attacker to attack the system at a later time. Thus the operation gets modified to:

  Compile : get (line);
  if line == "readpwd(p)" then translate (Trojan horse insertion)
  else
  translate (line);

- **The Race Condition Attack:** A race condition occurs when two or more operations occur in an undefined manner. Specifically, the attacker attempts to change the state of the file system between two file system operations. Usually, the program expects these two operations to apply to the same file, or expects the information retrieved to be the same. If the file operations are not atomic, or do not reference the same file this cannot be guaranteed without proper care.

  In Solaris 2.x's ps utility had a security hole that was caused by a race condition. The utility would open a temporary file, and then use the *chown()* system call with the file's full path to change its ownership to root. This sequence of events was easily exploitable. All that an attacker now had to do was to first slow down the system, and find the file so created, delete it, and then slip in a new SUID world writable file. Once the new file was created with that mode and with the ownership changed by *chown* to root by the insecure process, the attacker simply copies a shell into the file. The attacker gets a root shell.

The problem was that the second operation used the file name and not the file descriptor. If a call to *fchown*() would have been used on the file descriptor returned from the original *open*() operation, the security hole would have been avoided. File names are not unique. The file name /tmp/foo is really just an entry in the directory /tmp. Directories are special files. If an attacker can create, and delete files from a directory the program cannot trust file names taken from it. Or, to look at it in a more critical way, because the directory is modifiable by the attacker, a program cannot trust it as a source of valid input. Instead it should use file descriptors to perform its operations. One solution is to use the sticky bit (see Aside). This will prevent the attacker from removing the file, but not prevent the attacker from creating files in the directory. See below for a treatment of symbolic link attacks.

An Aside: Only directories can have sticky bit set. When a directory has the sticky bit turned on, anyone with the write permission can write (create a file) to the directory, but he cannot delete a file created by other users.

- **The Symlink Attack:** A security hole reported for SUN's license manager stemmed from the creation of a file without checking for symbolic links (or soft links). An *open*() call was made to either create the file if it did not exist, or open it if it did exist. The problem with a symbolic link is that an open call will follow it and not consider the link to constitute a created file. So if one had /tmp/foo. symlinked to /.rhosts or "/root/.rhosts), the latter file would be transparently opened. The license manager seemed to have used the O_CREAT flag with the open call making it create the file if it did not exist. To make matters worse, it created the file with world writable permissions. Since it ran as root, the .rhosts file could be created, written to, and root privileges attained.

9.2 Attack Prevention Methods

Attack prevention may be attempted at several levels. These include individual screening and physical controls in operations. Individual screening would require that users are screened to authenticate themselves and be responsible individuals. Physical controls involve use of physical access control. Finally, there are methods that may require configuration controls. We shall begin the discussion on the basic attack prevention with the defenses that are built in Unix. These measures are directed at user authentication and

file access control.

9.2.1 User Authentication

First let us consider how a legitimate user establishes his identity to the system to access permitted resources. This is achieved typically by username/password pair. When the system finishes booting, the user is prompted for a username and then a password in succession. The password typed is not echoed to the screen for obvious reasons. Once the password is verified the user is given an interactive shell from where he can start issuing commands to the system. Clearly, choosing a clever password is important. Too simple a password would be an easy give away and too complex would be hard to remember.

So how can we choose a nice password?

Choosing a good password: A malicious user usually aims at obtaining a complete control over the system. For this he must acquire the superuser or root status. Usually, he attacks vulnerable points like using badly installed software, bugs in some system software or human errors. There are several ways to hack a computer, but most ways require extensive knowledge. A relatively easier way is to log in as a normal user and search the system for bugs to become superuser. To do this, the attacker will have to have a valid user code and password combination to start with. Therefore, it is of utmost importance that all users on a system choose a password which is quite difficult to guess. The security of each individual user is closely related to the security of the whole system. Users often have no idea how a multi-user system works and do not realize that by choosing an easy to remember password, they indirectly make it possible for an attacker to manipulate the entire system. It is essential to educate the users well to avoid lackadaisical attitudes. For instance, if some one uses a certain facility for printing or reading some mails only, he may think that security is unimportant. The problem arises when someone assumes his identity. Therefore, the users should feel involved with the security of the system. It also means that it is important to notify the users of the security guidelines. Or at least make them understand why good passwords are essential.

Picking good passwords: We will look at some methods for choosing good passwords. A typical good password may consist up to eight characters. This means passwords like `members only' and `members and guests' may be mutually inter-changeable. A password should be hard to guess but easy to remember. If a password is not easy to remember then users will be tempted to write down their password on yellow stickers which makes it

futile. So, it is recommended that a password should not only have upper or lowercase alphabets, but also has a few non-alphanumeric characters in it. The non-alphanumeric characters may be like (%,,*, =) etc. The use of control characters is possible, but not all control characters can be used, as that can create problems with some networking protocols. We next describe a few simple methods to generate good passwords.

- Concatenate two words that together consist of seven characters and that have no connection to each other. Concatenate them with a punctuation mark in the middle and convert some characters to uppercase like in 'abLe+pIG'.
- Use the first characters of the words of not too common a sentence. From the sentence “My pet writers are Wodehouse and Ustinov", as an example, we can create password “MpwaW+U!”. Note in this case we have an eight-character password with uppercase characters as well as punctuation marks.
- Alternatively, pick a consonant and one or two vowels resulting in a pronounceable (and therefore easy to remember) word like 'koDuPaNi'.

This username/password information is kept traditionally in the /etc/passwd file, commonly referred to simply as the password file. A typical entry in the password file is shown below:

user:x:504:504::/home/user:/bin/bash

There are nine colon separated fields in the above line. They respectively refer to the user name, password x (explained later), UID, GID, the GECOS field 1, home directory and users' default shell. In the early implementations of the Unix, the password information was kept in the passwd file in plain text. The passwd file has to be world readable as many programs require to authenticate themselves against this file. As the expected trust level enhanced, it became imperative to encrypt the password as well. So, the password field is stored in an encrypted format. Initially, the crypt function was used extensively to do this. As the speed of the machines increased the encrypted passwords were rendered useless by the brute force techniques. All a potential attacker needed to do is to get the passwd file and then do a dictionary match of the encrypted password. This has led to another innovation in the form of the shadow suite of programs. In modern systems compatible with the shadow suite the password information is now kept in the /etc/shadow file and the password field in the passwd file is filled with an x (as indicated

above). The actual encrypted password is kept in the /etc/shadow file in the following format :

user:1UaV6PunD$vpZUg1REKpHrtJrVi12HP.:11781:0:99999:7:::

The second field here is the password in an md5 hash 2. The other fields relate to special features which the shadow suite offers. It offers facilities like aging of the passwords, enforcing the length of the passwords etc. The largest downside to using the shadow passwords is the difficulty of modifying all of the programs that require passwords from the appropriate file to use /etc/shadow instead. Implementing other new security mechanisms presents the same difficulty. It would be ideal if all of these programs used a common framework for authentication and other security related measures, such as checking for weak passwords and printing the message of the day.

Pluggable authentication modules Red Hat and Debian Linux distributions ship with “Pluggable Authentication Modules" (PAM for short) and PAM-aware applications. PAM offers a flexible framework which may be customized as well. The basic PAM-based security model is shown in Figure 9.1. Essentially, the figure shows that one may have multiple levels of authentication, each invoked by a separate library module. PAM aware applications use these library modules to authenticate. Using PAM modules, the administrator can control exactly how authentication may proceed upon login. Such authentications go beyond the traditional /etc/passwd file checks. For instance, a certain application may require the pass-word as well as a form of bio-metric authentication. The basic strategy is to incorporate a file (usually called /etc/pam.d/login) which initiates a series of authentication checks for every login attempt. This file ensures that a certain authentication check sequence is observed. Technically, the library modules may be selectable. These selections may depend upon the severity of the authentication required. The administrator can customize the needed choices in the script. At the next level, we may even have an object based security model in which every object access would require authentication for access, as well as methods invocation.

For now we shall examine some typical security policies and how Unix translates security policies in terms of access control mechanisms.

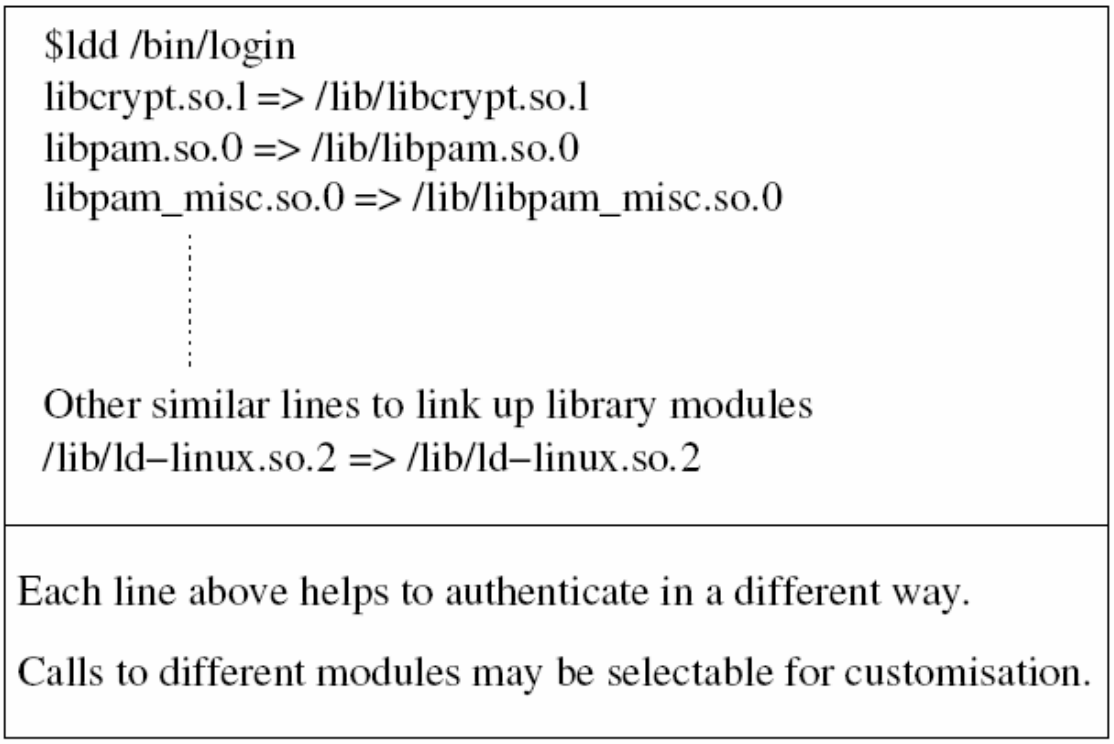

Figure 9.1: Pluggable authentication.

9.2.2 Security Policy and Access Control

Most access control mechanisms emanate from a stated security policy. It is important to learn to design a security policy and offer suitable access mechanisms that can support security policies. Security policy models have evolved from many real-life operating scenarios. For instance, if we were to follow a regime of defense forces, we may resort to a hierarchy based policy. In such a policy, the access to resources shall be determined by associating ranks with users. This requires a security-related labeling on information to permit access. The access is regulated by examining the rank of the user in relation to the security label of the information being sought. For a more detailed discussion the reader may refer where there is a discussion on how to specify security policies as well.

If we were to model the security policies based on commercial and business practices or the financial services model, then data integrity would take a very high precedence. This like, the accounts and audit practices, preserves the integrity of data at all times. In practice, however, we may have to let the access be governed by ownership (who own the information) and role definitions of the users. For instance, in an organization, an individual user may own some information but some critical information may be owned

by the institution. Also, its integrity should be impregnable. And yet the role of a system manager may require that he has access privileges which may allow him a free reign in running the system smoothly.

Almost all OSs provide for creating system logs of usage. These logs are extremely useful in the design of Intrusion Detection Systems (IDS). The idea is quite simple. All usages of resources are tracked by the OS and recorded. On analysis of the recorded logs it is possible to determine if there has been any misuse. The IDS helps to detect if a breach has occurred. Often this is after the event has taken place. To that extent the IDS provides a lot of input in designing security tools. With IDS in place one can trace how the attack happened. One can prevent attacks from happening in future. A full study and implementation of IDS is beyond the scope of this book. We would refer the reader to Amoroso's recent book on the subject.

Defenses in Unix: Defenses in Unix are built around the access control 3. Unix's access control is implemented through its file system. Each file (or directory) has a number of attributes, including a file name, permission bits, a UID and a GID. The UID of a file specifies its owner. In Chapter 2, we had explained that the permission bits are used to specify permissions to read (r), write (w), and execute (x). These permissions are associated with every file of every user, for the members of the user's group, and for all other users of that system. For instance, the permission string rwxr-x--x specifies that the owner may read, write and execute, the user's group members are allowed to read and execute it, while all the other users of the system may be permitted to only execute this file. A dash (`-') in the permission set indicates that the access rights are not permitted. Furthermore, each process in Unix has an effective and a real UID as well as an effective and a real GID associated with it. The real UID (and GID) are the primary identifications that Unix systems continually maintain based on the identifications assigned at the time of accounts creation. However, access rights and privileges evolve over time. The effective identifications precisely reflect that. A process's effective identification indicates the access privileges. Whenever a process attempts to access a file, the kernel will use the process's effective UID and GID to compare them with the UID and the GID associated with the file to decide whether or not to grant the request.

As we stated earlier, Unix logs the systems' usage. Unix kernel, and system processes, store pertinent information in the log files. The logs may be kept either locally, or

centrally, on an network server. Sometimes logs are prepared for a fixed duration of time (like for 1 to 30 days) or archived. The logs may be analyzed on-line or off-line on a secured isolated system. An analysis on a secured isolated system has the advantage that it cannot be modified by an attacker (to erase his trace). Also, the analysis can be very detailed as this is the only purpose of such a system.

With the security concerns coming into focus, security standards have emerged. Usually the security standards recommend achieving minimal assured levels of security through some form of configuration management. Most OSs, Unix included, permit a degree of flexibility in operations by appropriately configuring the system resources. In addition, modern Unix systems support a fairly comprehensive type of auditing known as C2 audit. This is so named because it fulfils the audit requirements for the TCSEC C2 security level.

Networking concerns: Realistically speaking almost all machines are networked. In any case every machine has built-in network (NW) support. The default NW support is TCP/IP or its variant. This is very assuring from the point of compatibility. The range of NW services support includes remote terminal access and remote command execution using *rsh, rlogin* commands and remote file transfer using *ftp* command. The remote service soliciting commands are collectively known as the r commands. The NW File System (NFS) is designed to offer transparency to determine the location of the file. This is done by supporting mounting of a remote file as if it was on the local file system. In fact, NFS technically supports multiple hosts to share files over a local area network (LAN). The Network Information System (NIS), formally known as the Sun Yellow Pages, enables hosts to share systems and NW databases. The NW databases contain data concerning user account information, group membership, mail aliases etc. The NFS facilitates centralized administration of the

file system. Basically, the r commands are not secure. There are many reasons why these are insecure operations. We delineate some of these below.

- The primary one being that Unix was designed to facilitate usage with a view to cooperating in flexible ways. The initial design did not visualize a climate of suspicion. So, they assumed that all hosts in the network are trusted to play by the rules, e.g. any request arising out of a TCP/IP port below 1024 is considered to be trusted.

- These commands require a simple address-based authentication, i.e. the source address of a request is used to decide whether or not to grant an access or offer a service.
- They send clear text passwords over the network.

Now a days there are other better alternatives to the r commands, namely *ssh, slogin* and *scp,* respectively, which use strong *ssl* public key infrastructure to encrypt their traffic.

Before an NFS client can access files on a file system exported by an NFS server, it needs to mount the file system. If a mount operation succeeds, the server will respond with a file handle, which is later used in all accesses to that file system in order to verify that the request is coming from a legitimate client. Only clients that are trusted by the server are allowed to mount a file system. The primary problem with NFS is the weak authentication of the mount request. Usually the authentication is based on IP address of the client machine. Note that it is not difficult to fake an IP address!!. So, one may configure NIS to operate with an added security protocol as described below.

- Ensure minimally traditional Unix authentication based on machine identification and UID.
- Augment data encryption standard (DES) authentication .

The DES authentication provides quite strong security. The authentication based on machine identification or UID is used by default while using NFS. Yet another authentication method based on Kerberos is also supported by NIS. The servers as well as the clients are sensitive to attacks, but some are of the opinion that the real security problem with NIS lies in the client side. It is easy for an intruder to fake a reply from the NIS server. There are more secure replacements for the NIS as well (like LDAP), and other directory services.

Unix security mechanisms rely heavily on its access control mechanisms. We shall study the access control in more detail a little later. However, before we do that within the broad framework of network concerns we shall briefly indicate what roles the regulatory agencies play. This is because NWs are seen as a basic infrastructure.

Internet security concerns and role of security agencies: In USA, a federally funded Computer Emergency Response Team (CERT) continuously monitors the types of attacks that happen. On its site it offers a lot of advisory information. It even helps organizations whose systems may be under attack. Also, there is critical infrastructure

protection board within whose mandate it is to protect internet from attack. The National Security Agency (NSA) acts as a watchdog body and influences such decisions as to what level of security products may be shipped out of USA. The NSA is also responsible to recommend acceptable security protocols and standards in USA. NSA is the major security research agency in USA. For instance, it was NSA that made the recommendation on product export restriction beyond a certain level of DES security (in terms of number of bits).

In India too, we have a board that regulates IT infrastructure security. For instance, it has identified the nature of Public Key Infrastructure. Also, it has identified organizations that may offer security-related certification services. These services assure of authentication and support non-repudiation in financial and legal transactions. It has set standards for acceptable kind of digital signatures.

For now let us return to the main focus of this section which is on access control. We begin with the perspective of file permissions.

File permissions: The file permissions model presents some practical difficulties. This is because Unix generally operates with none or all for group permissions. Now consider the following scenario:

There are three users with usernames Bhatt, Kulish, and Srimati and they belong to the group users. Is there anyway for Bhatt to give access to a file that he owns to Kulish alone. Unfortunately it is not possible unless Bhatt and Kulish belong to an identifiable group (and only these two must be members of that group) of which Srimati is not a member. To allow users to create their own groups and share files, there are programs like sudo which the administrator can use to give limited superuser privileges to ordinary users. But it is cumbersome to say the least. There is another option in the BSD family of Unix versions, where a user must belong to the Wheel group to run programs like *sudo* or *su*. This is where the Access Control Lists (ACLs) and the extended attributes come into picture. Since access control is a major means of securing in Unix we next discuss that.

More on access control in Unix: Note that in Unix all information is finally in the form of a file. So everything in Unix is a file. All the devices are files (one notable exception being the network devices and that too for historical reasons). All data is kept in the form of files. The configuration for the servers running on the system is kept in files. Also, the authentication information itself is stored as files. So, the file system's security is the

most important aspect in Unix security model. Unix provides access control of the resources using the two mechanisms:

(a) The file permissions, uid, gid.

(b) User-name and password authentication.

The file access permissions determine whether a user has access permissions to seek requested services. Username and password authentication is required to ensure that the user is who he claims to be. Now consider the following *rwx* permissions for user, group and others.

$ ls -l

drwxrwxr-x 3 user group 4096 Apr 12 08:03 directory

-rw-rw-r-- 1 user group 159 Apr 20 07:59 sample2e.aux

The first line above shows the file permissions associated with the file identified as a directory. It has read, write and execute permission for the user and his group and read and execute for others. The first letter `d' shows that it is a directory which is a file containing information about other files. In the second line the first character is empty which indicates that it is a regular file. Occasionally, one gets to see two other characters in that field. These are `s' and `l', where `s' indicates a socket and `l' indicates that the file is a link. There are two kinds of links in Unix. The hard link and the soft link (also known as symbolic links). A hard link is just an entry in the directory pointing to the same file on the hard disk. On the other hand, the symbolic link is another separate file pointing to the original file. The practical difference is that a hard link has to be on the same device as the original but the symbolic link can be on a different device. Also, if we remove a file, the hard link for the file will also be removed. In the case of a symbolic link, it will still exist pointing no where.

In Unix every legitimate user is given a user account which is associated with a user id (Unix only knows and understands user ids here to in referred as UIDs).The mapping of the users is maintained in the file /etc/passwd. The UID 0 is reserved. This user id is a special superuser id and is assigned to the user ROOT. The SU ROOT has unlimited privileges on the system. Only SU ROOT can create new user accounts on a system. All other UIDs and GIDs are basically equal.

A user may belong to one or more groups up to 16. A user may be enjoined to other groups or leave some groups as long as the number remains below the number permitted

by the system. At anytime the user must belong to at least one group. Different flavors of Unix follow different conventions. Linux follows the convention of creating one group with the same name as the username whenever a new user id is created. BSDs follow the convention of having all the ordinary users belong to a group called users.

It is to be noted that the permissions are matched from left to right. As a consequence, the following may happen. Suppose a user owns a file and he does not have some permission. However, suppose the group (of which he also is a member) has the permission. In this situation because of the left to right matching, he still cannot have permission to operate on the file. This is more of a quirk as the user can always change the permissions whichever way he desires if he owns the file. The user of the system must be able to perform certain security critical functions on the system normally exclusive to the system administrator, without having access to the same security permissions. One way of giving users' a controlled access to a limited set of system privileges is for the system to allow the execution of a specified process by an ordinary user, with the same permissions as another user, i.e. system privileges. This specified process can then perform application level checks to insure that the process does not perform actions that the user was not intended to be able to perform. This of course places stringent requirements on the process in terms of correctness of execution, lest the user be able to circumvent the security checks, and perform arbitrary actions, with system privileges.

Two separate but similar mechanisms handle impersonation in Unix, the so called set UID, (SUID), and set-GID (SGID) mechanisms. Every executable file on a file system so configured, can be marked for SUID/SGID execution. Such a file is executed with the permissions of the owner/group of the file, instead of the current user. Typically, certain services that require superuser privileges are wrapped in a SUID superuser program, and the users of the system are given permission to execute this program. If the program can be subverted into performing some action that it was not originally intended to perform, serious breaches of security can result.

The above system works well in a surprising number of situations. But we will illustrate a few situations where it fails to protect or even facilitates the attacker. Most systems today also support some form of access control list (ACL) based schemes

Access control lists: On Unix systems, file permissions define the file mode as well. The file mode contains nine bits that determine access permissions to the file plus three special bits. This mechanism allows to define access permissions for three classes of users: the file owner, the owning group, and the rest of the world. These permission bits are modified using the *chmod* utility. The main advantage of this mechanism is its simplicity. With a couple of bits, many permission scenarios can be modeled. However, there often is a need to specify relatively fine-grained access permissions.

Access Control Lists (ACLs) support more fine grained permissions. Arbitrary users and groups can be granted or denied access in addition to the three traditional classes of users. The three classes of users can be regarded as three entries of an Access Control List. Additional entries can be added that define the permissions which the specific users or groups are granted.

An example of the use of ACLs: Let's assume a small company producing soaps for all usages. We shall call it Soaps4All. Soaps4All runs a Linux system as its main file server. The system administrator of Soaps4All is called Damu. One particular team of users, the Toileteers, deals with the development of new toilet accessories. They keep all their shared data in the sub-directory /home/toileteers/shared. Kalyan is the administrator of the Toileteers team. Other members are Ritu, Vivek, and Ulhas.

Username Groups Function

Damu users System administrator

Kalyan toileteers, jumboT, perfumedT administrator

ritu toileteers, jumboT

vivek toileteers, perfumedT

ulhas toileteers, jumboT, perfumedT

Inside the shared directory, all Toileteers shall have read access. Kalyan, being the Toileteers administrator, shall have full access to all the sub-directories as well as to files in those sub-directories. Everybody who is working on a project shall have full access to the project's sub-directory in /home/toileteers/shared.

Suppose two brand new soaps are under development at the moment. These are called Jumbo and Perfumed. Ritu is working on Jumbo. Vivek is working on Perfumed. Ulhas is

working on both the projects. This is clearly reflected by the users' group membership in the table above.

We have the following directory structure:

$ ls -l

drwx------ 2 Kalyan toileteers 1024 Apr 12 12:47 Kalyan

drwx------ 2 ritu toileteers 1024 Apr 12 12:47 ritu

drwxr-x--- 2 Kalyan toileteers 1024 Apr 12 12:48 shared

drwx------ 2 ulhas toileteers 1024 Apr 12 13:23 ulhas

drwx------ 2 vivek toileteers 1024 Apr 12 12:48 vivek

/shared$ls -l

drwxrwx--- 2 Kalyan jumbo 1024 Sep 25 14:09 jumbo

drwxrwx--- 2 Kalyan perfumed 1024 Sep 25 14:09 perfumed

Now note the following:

- Ritu does not have a read access to /home/toileteers/shared/perfumed.
- Vivek does not have read access to /home/toileteers/shared/jumbo.
- Kalyan does not have write access to files which others create in any project sub-directory.

The first two problems could be solved by granting everyone read access to the /home/toileteers/shared/ directory tree using the others permission bits (making the directory tree world readable). Since nobody else but Toileteers have access to the /home/toileteers directory, this is safe. However, we would need to take great care of the other permissions of the /home/toileteers directory.

Adding anything to the toileteers directory tree later that is world readable is impossible. With ACLs, there is a better solution. The third problem has no clean solution within the traditional permission system.

The solution using ACLs: The /home/toileteers/shared/ sub-directories can be made readable for Toileteers, and fully accessible for the respective project group. For Kalyan's administrative rights, a separate ACL entry is needed. This is the command to grant read access to the Toileteers. This is in addition to the existing permissions of other users and groups:

*$setfacl -m g:toileteers:rx **

*$getfacl **

 # file: jumbo

 # owner: Kalyan

 # group: jumbo

user::rwx

group::rwx

group:toileteers:r-x

mask:rwx

other:---

 # file: perfumed

 # owner: Kalyan

 # group: perfumed

user::rwx

group::rwx

group:toileteers:r-x

mask:rwx

other:---

Incidentally, AFS(Andrew File System), and XFS (SGI's eXtended File System) support ACLs.