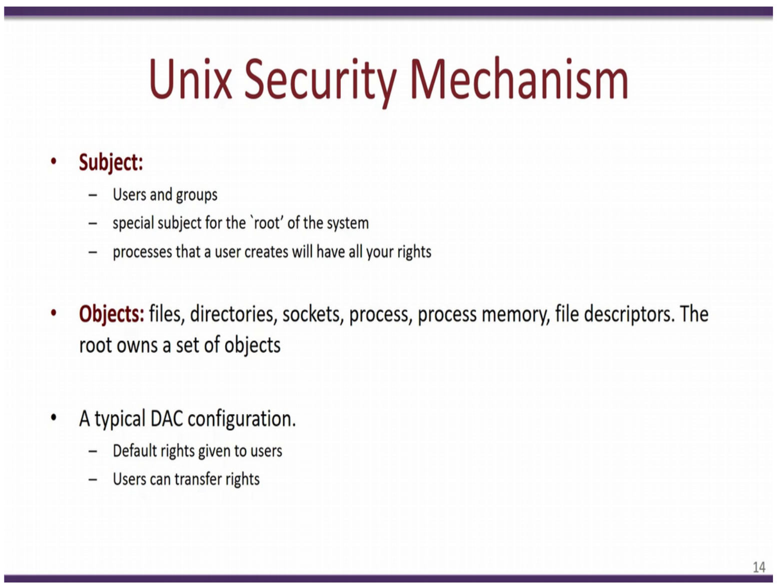
Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Access Control in linux
Mod05_Lec29

Hello and welcome to this lecture in the course for secure system engineering, in the previous lecture we had a brief introduction to access control and we looked at the discretionary access control policies and how it can be implemented using capabilities, access control list and the earliest form which was the access control matrix, so in this lecture we will be looking at Unix security mechanisms which essentially is discretionary access control technique to achieve access control in the Unix operating system.

(Refer Slide Time: 0:53)



Unix Security Mechanism

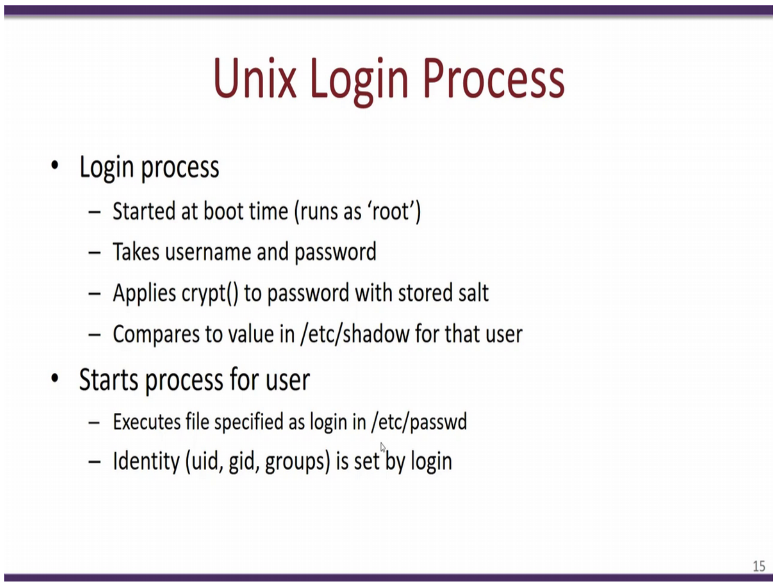
- **Subject:**
  - Users and groups
  - special subject for the `root' of the system
  - processes that a user creates will have all your rights
- **Objects:** files, directories, sockets, process, process memory, file descriptors. The root owns a set of objects
- A typical DAC configuration.
  - Default rights given to users
  - Users can transfer rights

14

So in Unix operating system you have subjects and objects, subjects are defined as users and groups and of course we have this special subject which is the superuser or the root of the system, processes that a user creates are also subjects and essentially the inherit all the permissions and privileges that particular user has.

So object on the other hand can vary from files, directories, sockets, processes, process memory, file descriptors and so on, each flavour of Unix defines a certain set of access rights that the subject has on the particular objects, so there will be subtle variations between one Unix flavour with respect to another.

(Refer Slide Time: 1:49)



Unix Login Process

- Login process
  - Started at boot time (runs as 'root')
  - Takes username and password
  - Applies crypt() to password with stored salt
  - Compares to value in /etc/shadow for that user
- Starts process for user
  - Executes file specified as login in /etc/passwd
  - Identity (uid, gid, groups) is set by login

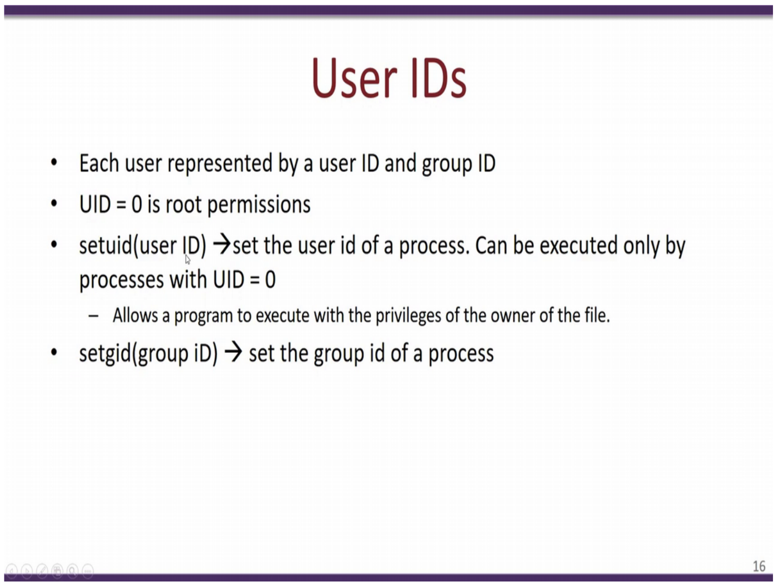
15

So let us start with the Unix login process, the login process for system is started at boot time and it runs with superuser privileges, so it request for a username and password, so when the user enters the password a cryptographic algorithm is used on the password with the stored salt and the result is something known as the hash of the password, now all users and their corresponding hash passwords are stored in a file/called etc/shadow, so what happens is that for the password that is actually entered and hash computed this is compared with the corresponding user entry in the/etc /shadow.

Login is permitted only if there is a match between this entry with in the stored file with the corresponding hashed password which the user enters, so once the login process successfully completes, what would happen is that another file is accessed, so this file is the/etc /password file which is present in the LINUX systems in particular and it contains various information about user, for example that is something known as the uid, gid and groups.

So uid is the user identifier it is a number and it is used to uniquely identify that particular user group is gid which is a group identifier which identifies the group in which the user wants to and it also along with other things the/etc /password also has entries corresponding to the home directory of that particular user and so on.

(Refer Slide Time: 3:45)



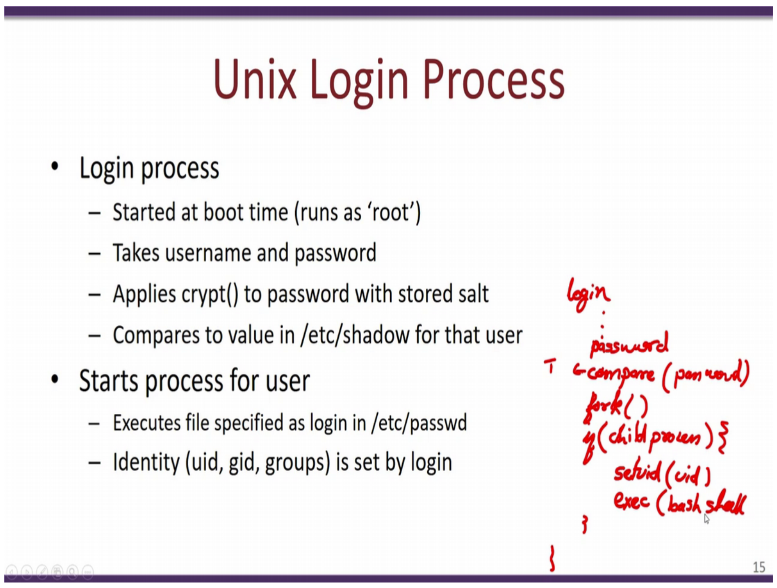
User IDs

- Each user represented by a user ID and group ID
- UID = 0 is root permissions
- setuid(user ID) →set the user id of a process. Can be executed only by processes with UID = 0
  - Allows a program to execute with the privileges of the owner of the file.
- setgid(group iD) → set the group id of a process

16

Will the look at the user identifiers or uids, each user is given a unique user identifier and a group identifier, a uid of 0 is considered as the roots user id, now there is a system called known as the setuid which is passed a user identifier is essentially what the system called would do is to set the user id of a particular process, so therefore when this process execute it will execute with the specified user id.

(Refer Slide Time: 4:24)



Unix Login Process

- Login process
  - Started at boot time (runs as 'root')
  - Takes username and password
  - Applies crypt() to password with stored salt
  - Compares to value in /etc/shadow for that user
- Starts process for user
  - Executes file specified as login in /etc/passwd
  - Identity (uid, gid, groups) is set by login

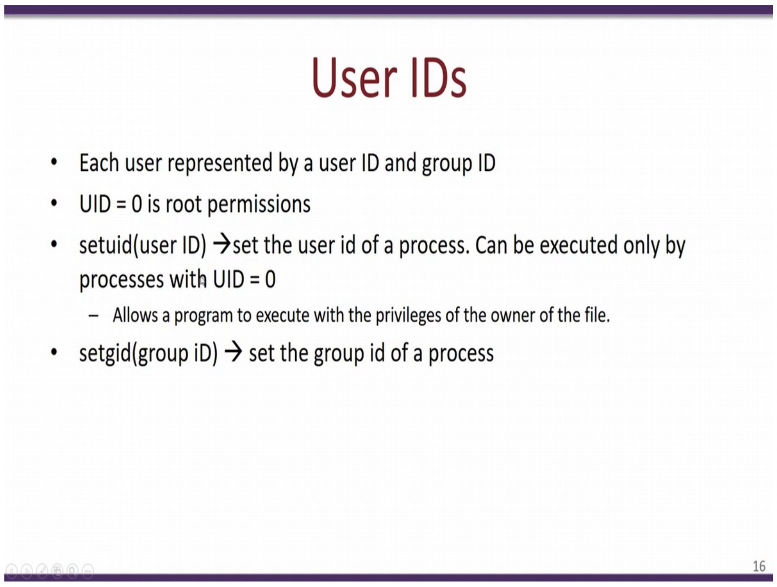
```
login
   :
   password
T ← compare (password)
   fork( )
   if(child process) {
       setuid (uid)
       exec (bash shell
   }
}
```

15

Now if we go back to this what you see is that the login process executes as root and was the password is verified it will then create a shell for that particular user, so essentially this shell is created something as follows, so you would have let us say the login process and within this login process you obtain the password and compare the password with the stored hash and if this is true, then we fork a process and in the child process.

What we do is that we would setuid corresponding to the users identifier and then execute the bash shell, so what we see over here is that even though the login process executes with superuser privileges when it actually compare this and successfully authenticates the user, it forks a process, changes the uid corresponding to that particular user and then executes the shell,

(Refer Slide Time: 6:17)



User IDs

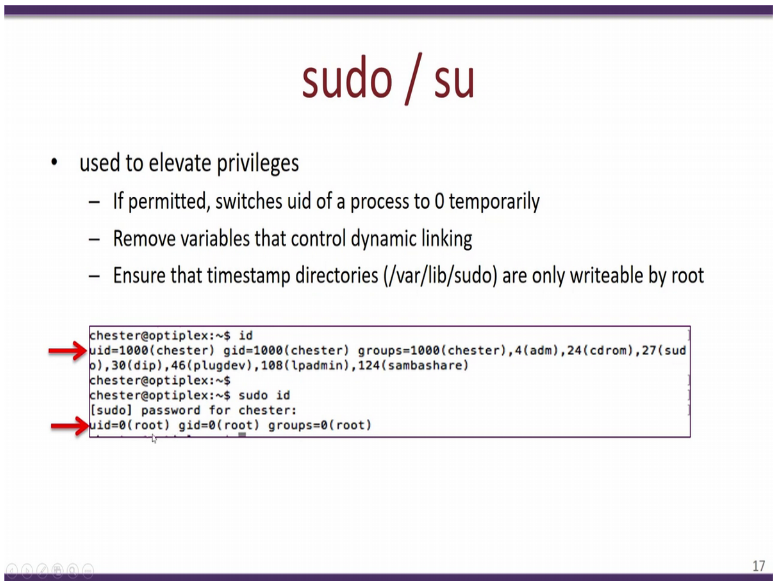
- Each user represented by a user ID and group ID
- UID = 0 is root permissions
- setuid(user ID) →set the user id of a process. Can be executed only by processes with UID = 0
  - Allows a program to execute with the privileges of the owner of the file.
- setgid(group iD) → set the group id of a process

16

So therefore when the shell executes, it executes with the uid of that particular user, now every program that gets executed by that particular user would automatically inherit the user id from the parent process, the for all the processes that the user actually executes would have a uid corresponding to that particular user, in a very similar way each user of that system is also associated with a particular group and therefore associated with a group identifier.

So similar to the setuid we also have a setgid, given a group identifier which sets the group id that particular process, now every time we fork and create a new process, which I will process would in the group id in a very similar way as it inherits the uid of its parent process.

(Refer Slide Time: 7:12)



sudo / su

- used to elevate privileges
  - If permitted, switches uid of a process to 0 temporarily
  - Remove variables that control dynamic linking
  - Ensure that timestamp directories (/var/lib/sudo) are only writeable by root

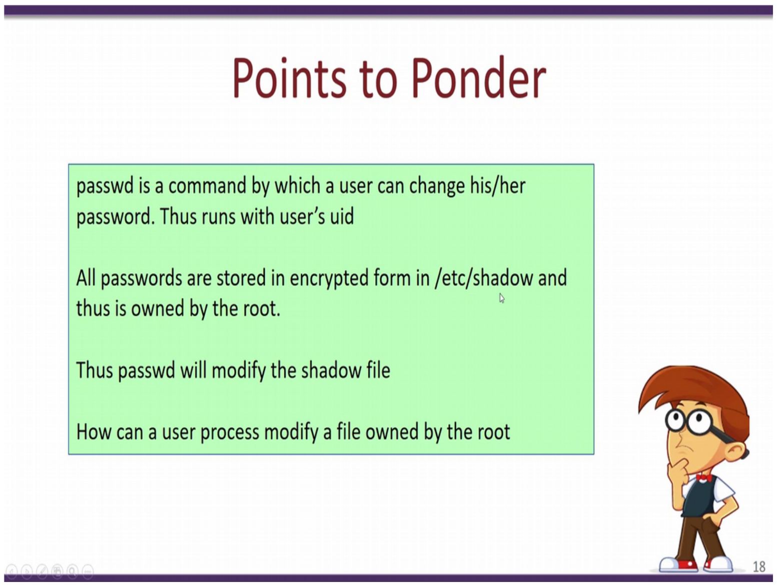
```
chester@optiplex:~$ id
uid=1000(chester) gid=1000(chester) groups=1000(chester),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),108(lpadmin),124(sambashare)
chester@optiplex:~$
chester@optiplex:~$ sudo id
[sudo] password for chester:
uid=0(root) gid=0(root) groups=0(root)
```

17

Two other important functions in this perspective are the sudo and su, the sudo and su, lets a particular user eliminate the privileges of a particular process, so on a completing a sudo successfully the process would then run with uid of 0 which would imply that it runs with all the privileges of a root user, so which is to you an example, if you run id over here, it tells me all the ids for that particular user.

So for example id over here tells me that the uid is 1000, my gid is 1000 and so on, now when I do sudo id what I obtained is that the uid is 0, the gid is 0 and so on, so what essentially is obtained over here is that a normal user who has a uid of 0 is getting privileges to run the id command with a privilege of a root user.

(Refer Slide Time: 8:16)



Points to Ponder

passwd is a command by which a user can change his/her password. Thus runs with user's uid

All passwords are stored in encrypted form in /etc/shadow and thus is owned by the root.

Thus passwd will modify the shadow file

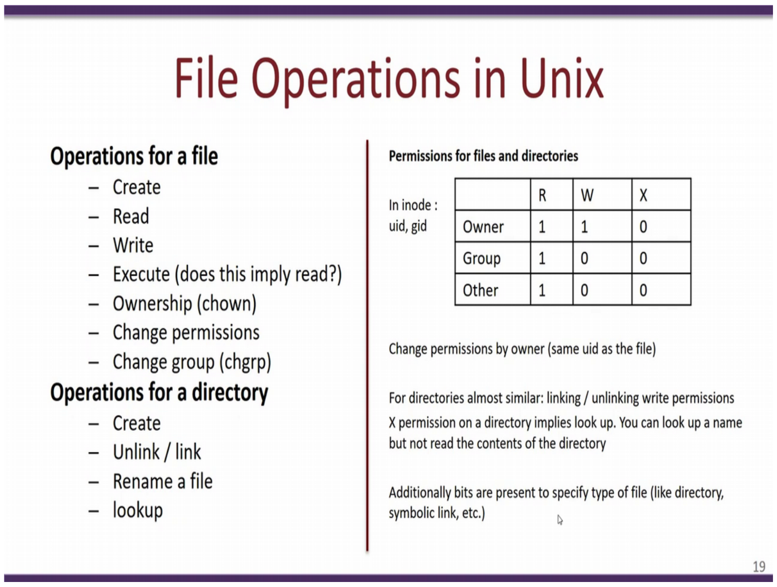
How can a user process modify a file owned by the root

18

So this is something for you to think about, password is a command that is run by a user to change his or her password, since this is a password link to a particular user, it should be run by a user and therefore the sudo permission should not be given for that particular user, in other words the password command is executed by a user with his normal uid and does not require to escalate privileges for that particular password.

Another point is that all passwords are stored in the encrypted form in the file/etc/shadow, now this file comprises of all passwords of all users and therefore should not be accessed by any ordinary user, thus in the Linux system this/etc/shadow is only owned by the root user, now to modify a password using the password command, it would require to write to this/etc/shadow file corresponding to the entry for that user, question to think about over here is that how can a password program which runs as the normal user program modify a file/etc/shadow which is owned by the root.

(Refer Slide Time: 9:42)



File Operations in Unix

Operations for a file

- Create
- Read
- Write
- Execute (does this imply read?)
- Ownership (chown)
- Change permissions
- Change group (chgrp)

Operations for a directory

- Create
- Unlink / link
- Rename a file
- lookup

Permissions for files and directories

In inode : uid, gid

	R	W	X
Owner	1	1	0
Group	1	0	0
Other	1	0	0

Change permissions by owner (same uid as the file)

For directories almost similar: linking / unlinking write permissions
X permission on a directory implies look up. You can look up a name but not read the contents of the directory

Additionally bits are present to specify type of file (like directory, symbolic link, etc.)

19

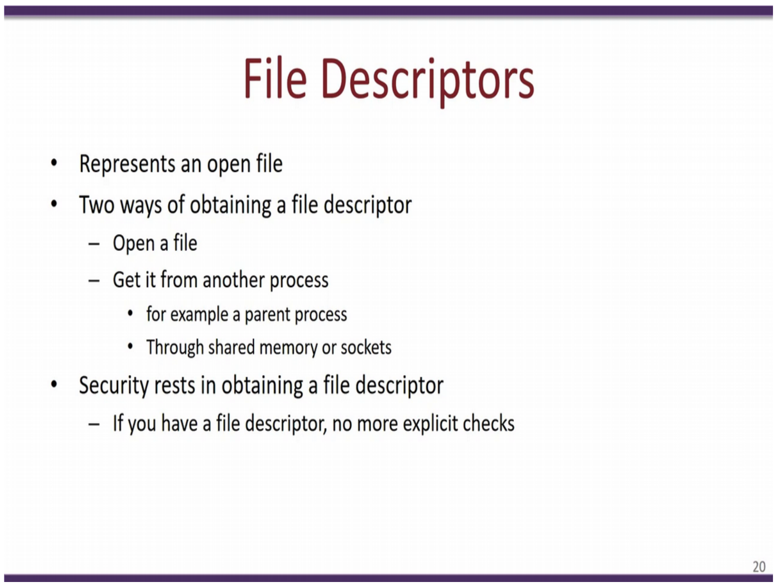
Now we will look at the various objects in the Unix and Linux system and we will see about how the various access control policies for the various objects are managed in a typical Unix system, so for example a file right, the various operations on a file are of course the creation, read, write, execute, we also have other operations which relate to ownership, change of permissions and change group, so one could change the ownership of a particular file by the chown command, we can similarly change the permissions for a file with a chmode command and change group of a file with chgrp command.

Similar kind of operations are specified for directories as well, one could create link or unlink directories, rename a file within a directory or look up a particular directory, so in order to manage this in the Unix system there is a small table which is maintained, where we have the permissions read, write and execute and we also have three different varieties in order to achieve the access control for these various files, we have the a small table which is maintained by the Unix system.

Where we have the read, write, execute operations that are permitted on three different types of users of that particular object, one is the owner of the object, second is the group which the owner belongs to and the third or are all the other users, so for each of this cases we can specify whether the file can be read, written to or executed.

So this can be specified for each file in the system as well as each directory as well, while it makes sense to actually have a read and write a directory execute permission or X permission on the directory has a different meaning, so it essentially means that one could lookup a particular directory, so essentially you cannot list the contents of the directory or read the contents of the directory but essentially you could lookup the name of that directory, in addition to these read, write, execute bits what is specified is other aspects such as symbolic links, directory files, sticky bits and so on.

(Refer Slide Time: 12:22)



File Descriptors

- Represents an open file
- Two ways of obtaining a file descriptor
  - Open a file
  - Get it from another process
    - for example a parent process
    - Through shared memory or sockets
- Security rests in obtaining a file descriptor
  - If you have a file descriptor, no more explicit checks

20

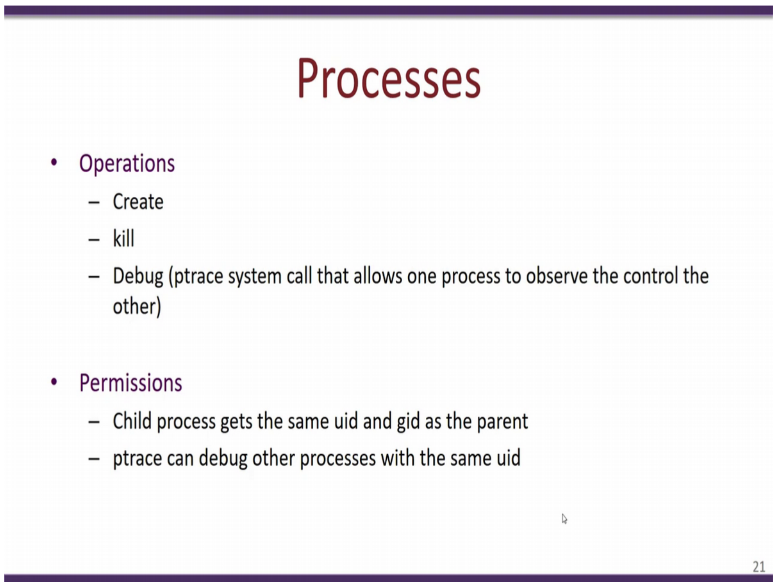
Will now look at file descriptors, once you open a file using the open system call in which is specify a file path along with permissions that you want to read or write or append to that particular file and what you obtain is a file descriptor, so note that the entire security rest in just obtaining the file descriptor, so for example once your open system call has successfully created or open that file and you have obtained the file descriptor for that particular file after that there are no special test that are done on that file.

You can read and write to that file without any access control test which are done, so based on this there are two ways in which you can obtain a file descriptor, one is through using the open system call during which access control permissions are checked by the operating system before giving you a successful file descriptor, a second way to obtain a file descriptor is from another process or from shared memory, for example when a process spawns a child process than a child process would can inherit all the file descriptors.

Thus a file descriptor obtained by the child process does not have to come from an open system call in its process but rather it is actually inheriting the file descriptor from the parent process, another way to obtain a file descriptor is through shared memory or sockets, so this would mean that a process could send a file descriptor to an other process through a shared memory and therefore that second process can then read or write to the file without anymore explicits checks done by the operating system.

So in this way what can be achieved is that I could create a particular file which is accessible only by root users but then I could open this particular file and send that file descriptor to a normal user who is not a root, now this normal user can then execute and read and write to this particular file, so thus what is achieved is that a normal user can read or write to a file which is owed by a root.

(Refer Slide Time: 14:58)



Processes

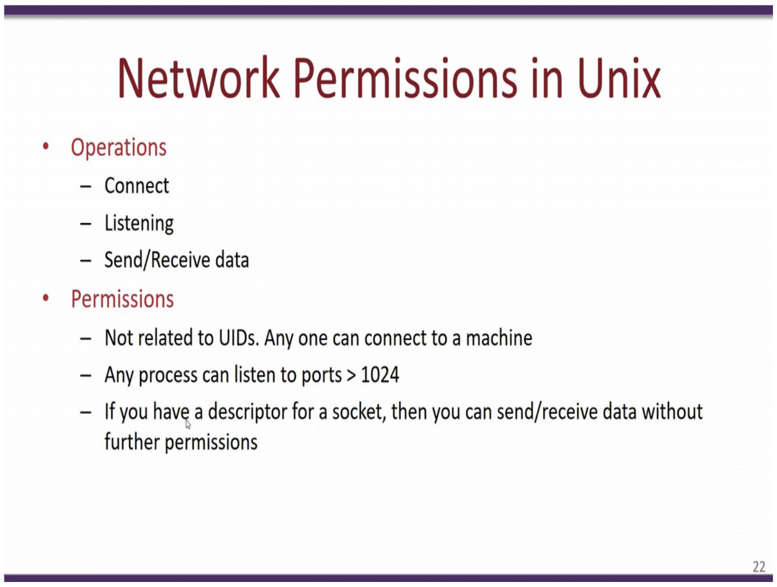
- Operations
  - Create
  - kill
  - Debug (ptrace system call that allows one process to observe the control the other)
- Permissions
  - Child process gets the same uid and gid as the parent
  - ptrace can debug other processes with the same uid

21

So let us look at processes, a process is both a subject and object in the UNIX nomenclature, so the operations that can be permitted on a process is to create a process, killer process or debug a process, so a debug is typically done with a ptrace system call that allows one process to observe and control the other process, your typical debuggers such as the GDB or all the different variants of GDB would typically use the ptrace system call in order to set a breakpoints, watch, look at the various memory locations, read the register contents and so on for the child process.

With respect to the permissions when a process gets created the child process gets the same uid and gid as the parent, essentially the child process inherits all the parents uid and gid as well, similarly when you are using ptrace, ptrace can debug other processes with the same uid, thus if I use GDB for example which internally uses the system call ptrace, GDB can only debug processes which have the same uid, therefore I will not be able to debug and other users process, so I will only be able to debug a process which is created with my same uid, this of course does not hold true for root, the root has access to all processes and therefore can run GDB on every process present in the system.

(Refer Slide Time: 16:47)



Network Permissions in Unix

- Operations
  - Connect
  - Listening
  - Send/Receive data
- Permissions
  - Not related to UIDs. Any one can connect to a machine
  - Any process can listen to ports > 1024
  - If you have a descriptor for a socket, then you can send/receive data without further permissions

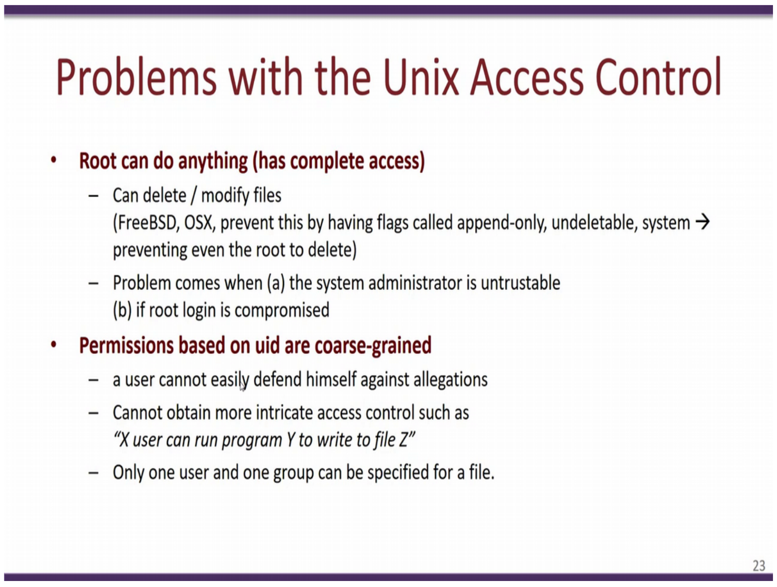
22

So let us look at the network permissions in a Unix system, the operations permitted on a network socket is to connect, listen, send and receive data, now with respect to permissions related to network sockets is like a unlike files or any other devices, network sockets are not related to uids, so this means anyone can actually connect to a machine, a particular person does not have to have a valid user account on that machine in order to connect to do particular machine.

So this is important with respect to say Webservers, now when we host a web server on a particular machine, any user could actually connect to do particular machine and view the pages on that particular web server, that user does not required to have a valid login on that particular web server, further any process can listen to ports which are greater than 1024, for network ports which are less than 1024, it requires superuser privileges because these ports have linked a special system processes.

While for processes which are greater than 1024, any process within the system could actually open and listen to that port, so once you have opened a socket successfully are no further checks which are done and therefore you can send and receive data through that socket without any further checks or any further permissions, thus we see the compared to files and other objects in the Unix system, network sockets are treated in a very different way.

(Refer Slide Time: 18:30)



Problems with the Unix Access Control

- **Root can do anything (has complete access)**
  - Can delete / modify files
    (FreeBSD, OSX, prevent this by having flags called append-only, undeletable, system → preventing even the root to delete)
  - Problem comes when (a) the system administrator is untrustable
    (b) if root login is compromised
- **Permissions based on uid are coarse-grained**
  - a user cannot easily defend himself against allegations
  - Cannot obtain more intricate access control such as
    "X user can run program Y to write to file Z"
  - Only one user and one group can be specified for a file.

23

While the Unix access control mechanisms are quite easy to understand, quite easy to implement and permits are lots of flexibility in the way, things are done, there are still a lot of problems in the Unix access control mechanisms when it comes to security, one of the main problems is that the root can do almost anything, so it can read and modify or create files, it can read any or it can delete or modify files, it can read or execute any files, independent of the owner of those particular files.

Now the problem with this is that if the system administrator is an trusted then it means that the entire systems secret data can be leaked, further if the root itself gets compromised for example if there is a buffer overflow wonderbility in one of the root programs, then that program gets compromised and then the malware gets root access to the system and thus compromises the entire system because the malware can read and write to all files in the system.

Another problem with Unix access control mechanisms is that it is highly coarse-grained, so for example more intricate access control mechanisms such as X user can run programs Y to write to files Z is not very easily specified in this Unix access control mechanisms, further user cannot easily defend himself against allegations, so let us say for example that a normal user working in an organisation and he happens to run a particular program which has a malware.

Now a malware deletes all the files of that particular organisation from that system, including all the sensitive files, now we cannot actually blame the user for that particular incident because it is not the users fault, the deletion of the files was done by the malware and therefore defending against such allegations becomes difficult, another problem with the Unix space access control mechanisms is that only one user and one group can be specified for a particular file, often we would require more flexibility where we want two users to own a particular file and this is not always possible with the Unix file systems.

So therefore there are multiple issues with standard Unix access control policies and there are several variants of Unix systems which have actually migrated to more stringent measures, so as we seen in this lecture there are various problems with the Unix access control mechanisms and therefore many Unix flavours has actually migrated to a much better access control policies using something known as information flow policies, so in the next lecture we look at more details about information flow policies. Thank you.