Access Control Lists in Linux & Windows

Vasudevan Nagendra & Yaohui Chen

Categorization: Access Control Mechanisms

- **Discretionary Access Control (DAC):** Owner of object specifies who can access object (files/directories)
  - Control access on discretion of owner
  - Access privileges decided when file created
  - Ex: Windows, Linux, Mac, Unix
- **Mandatory Access Control (MAC):** system specifies which subjects(users/processes) can access which objects.
  - Based on security labels mechanism
    - Subjects are given clearance
    - Objects are given security classification
  - Matches clearance of subject with classification of object.
  - Examples: secret, top secret, confidential

Access Control List (ACLs)

- **Filesystem Access Control mechanisms:**
  - ACLs
  - Role Based Access (RBAC) - Can be Implemented as either DAC/MAC
- **ACL:** Fine-grained discretionary access rights given to files & directories.
  - Specifies, which users/processes are granted access to objects.
  - Access rights tied with objects.
- **RBACs:** System access on basis of authorization
  - specific roles are permitted to perform certain operations
  - Access rights not tied to objects
  - Example: Roles created for various job functions.
    - Consider multiuser systems with users of different roles are accessing.

ACLs Continued..

- **Network Access Control Mechanism:**
  - Netfilter
- **Netfilter (NACL):** network traffic filtering framework for Linux
  - Set of hooks in kernel to handle packets.
    - Intercept calls, events or messages
    - Between s/w components of OS or Applications.
  - Registers callbacks with n/w stack, called for every packet.
  - Access Controls / Filtering rules applied here.

Background: 9 bit permission Model

- Every file system is associated with:
  - 3 set of user groups(classes),
  - 3 set of permissions
  - 9 bits are used to determine the characteristics
  - Also called as base/*minimal* ACLs.
- Example: ls -la file.txt

-**rwx**rw-r-- 1 root cse506 2 Nov 19 05:55 file.txt

  - Owner class with read, write & execution access
  - Group class with read & write access
  - Others class with read only access.
  - For changing the file permissions we use the chmod.

Background: Other Access Control Options

- **Setuid**: Allows subjects to run executable with permission of file owner.
  - When subject doesn't have adequate permission
  - Examples: passwd/gpasswd/sudo/chsh/mount/ping/su/umount

- **Setgid**: Equivalent (as setuid) property for groups.
  - No matter which user starts it, program runs under group ID
  - All files & directories created in the setgid directory, will belong to the group owning the setgid directory.

- **Sticky bit**: Assigned to directories, prevents users from deleting each other's files.
  - Example: /tmp where any user can store files, but only owner of file has rights to modify or delete the file.

UMASK

- Consider default behavior of file and directory creation
  - 666 & 777 respectively.
  - To change this default behavior – use ***umask***
- Defines the permissions to be masked while object is created.
- Examples: umask 002
  - File creation: (666 - 002= 664) = rw- rw- r--
  - Directory creation: (777 - 002= 775) = rwx rwx r-x

Drawbacks & Limitations of 9 bit permission model

The price of playing tricks with this permission model:

- Setuid-root - Allows even ordinary users to perform administrative tasks.
  - Buggy application easily compromises system
  - Increase complexity of system configurations.
- Limitations of the base/9 bit permission model:
  - No fine grained control access to non-class users

Extended ACLs for finer-grain control

- **Extended ACLs provides:**
  - beyond simple user/group/other ownership.
  - more than 3 base classes
  - contains any number of named user & groups
  - contains mask entry.

Utilities/Library functions:

- **getfacl:** Check the current state of ACL on file/directory.
  getfacl test-dir

- **setfacl:** Modify/add ACL to additional user or group.
  setfacl -m user:student1:rwx,group:osclass:rwx test-file

- **chacl:** changes the ACL of file or directory
  chacl u::rwx,g::r-x,o::r– test-file

Access Control Entries (ACE)

- Set of entries that defines permissions for user or groups

Example of an ACL Entry in Linux system:

```
Type   |   TextForm
owner   user::rw-
owning group     group:rw-                    /*Base Class*/
other  other::--
named user  user::vasu:rwx
named group    group:vasu_grp:rwx             /*Extended Class*/
mask  mask::rw-
default:user::rwx
default:group::r-x
default:group:vasu_grp2:r-x                   /*Default class*/
default:mask::r-x
default:other::---
```

More details of Extended ACLs

- **Default ACL:**
  - defined for a directory
  - the objects in directory inherits it.

- Extended ACLs contains entries for additional users or groups.
  - What If permissions are not contained with in owning group?
  - Solution: Solved by virtue of Mask entry.

- **Mask Entry:** maximum access rights that can be granted for users and groups.
  - Mask applicable on:
    - Named user,
    - Named group &
    - Owning Group

Extended Attributes (EAs)

- Typically stored in separate datablock, referenced from inodes.
  - Attributes: Defines Properties of files

Examples:

1. For ext4 fs in linux,
   - inode has a field *i_file_acl* (type ext4_fsblk_t),
   - i_file_acl -> references to filesystem block with EAs stored

2. For Solaris with UFS file system,
   - inode has a field *i_shadow*
   - References to file system block with EAs stored
   - files with same ACL points to same shadow inode.
     - Implementation dependent optimization.

ACL Implementations

- How ACLs passed between user and kernel space?
  - FreeBSD, Solaris, Irix & HP-UX have separate ACL system calls.
  - Linux: Uses Extended Attributes.
  - Huge Performance degrade for file access at first.
  - ACL Caching is provided by some file system.
  - some filesystems limits # of ACEs. (Implementation Dependent)

Access Check Algorithm

- Subject's access request to object –

Step 1: Select ACL entry that closely matches requesting process

- ACL Entries Looked up in following order:
  - owner
  - named users
  - (owning or named) groups
  - Only single entry determines the access.

Step 2: checks if matching entry contains sufficient permissions.

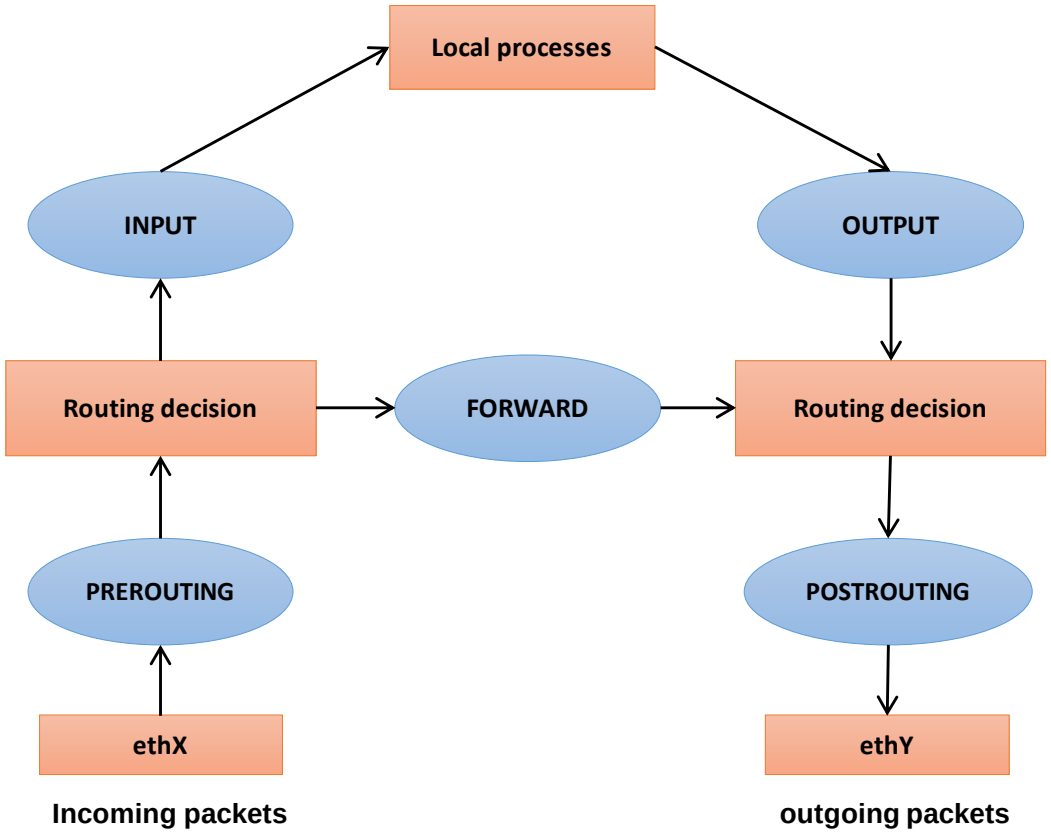
Netfilter - Network ACLs for Linux.

- Packet filtering framework inside Linux kernel.
- Enables following main functions:
  - Packet filtering: ACCEPT/ Drop / Log & other actions
  - NAT: Changing IP/Port (Source & Destination)
  - Mangling: Changing packet contents, ToS, Labeling, etc.,
- Support: Both stateless & stateful packet filtering
  - stateless: No track of the state of packets
  - stateful: Keeps track of packets
- Supports both IPv4 & IPv6

Netfilter Architecture for Network ACLs.

- Hooks & Custom Functions:
  - provided at several points of kernel network stack
  - Hooks: exploited to define custom functions
    - Manipulating Packets headers & data.
    - Actions on packets itself.
- Purpose of Hooks:
  - Debugging
  - Extending functionality
    - Intercepting keyboard/mouse events
    - Monitor system calls to analyze system behavior

Architecture: Netfilter Architecture



Kernel path for Incoming packets

Figure: Netfilter Architecture

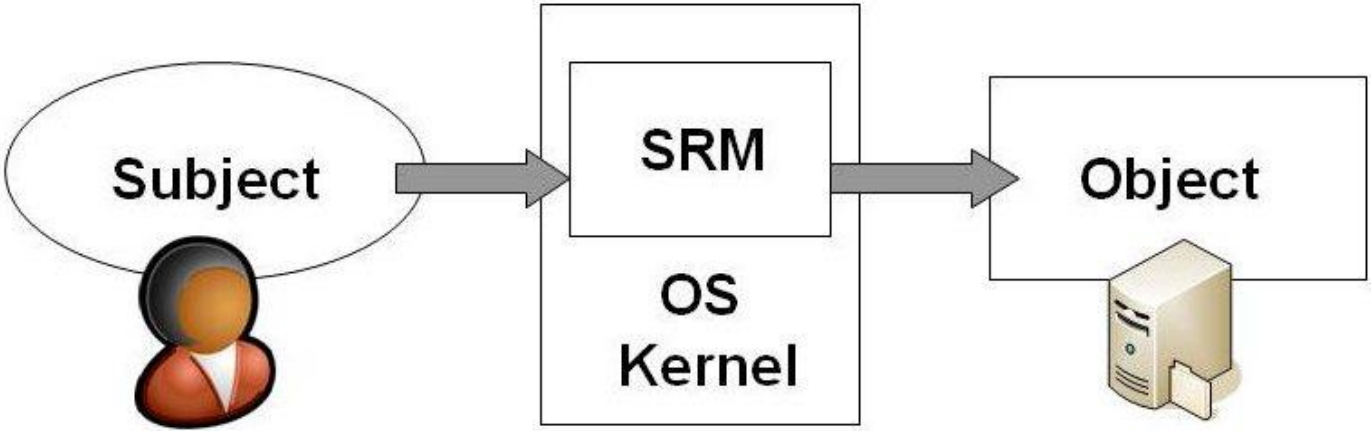
- **PREROUTING**: Functions triggered before routing decision
- **POSTROUTING**: triggered after routing decision.
- **FORWARD**: Action on forwarded packets - “ACLs”.
- **INPUT**: Action on Incoming packets
- **OUTPUT**: Actions on Outgoing packets.

Improving the Granularity of Access Control in Windows NT

Yaohui Chen

Access Control In Windows NT

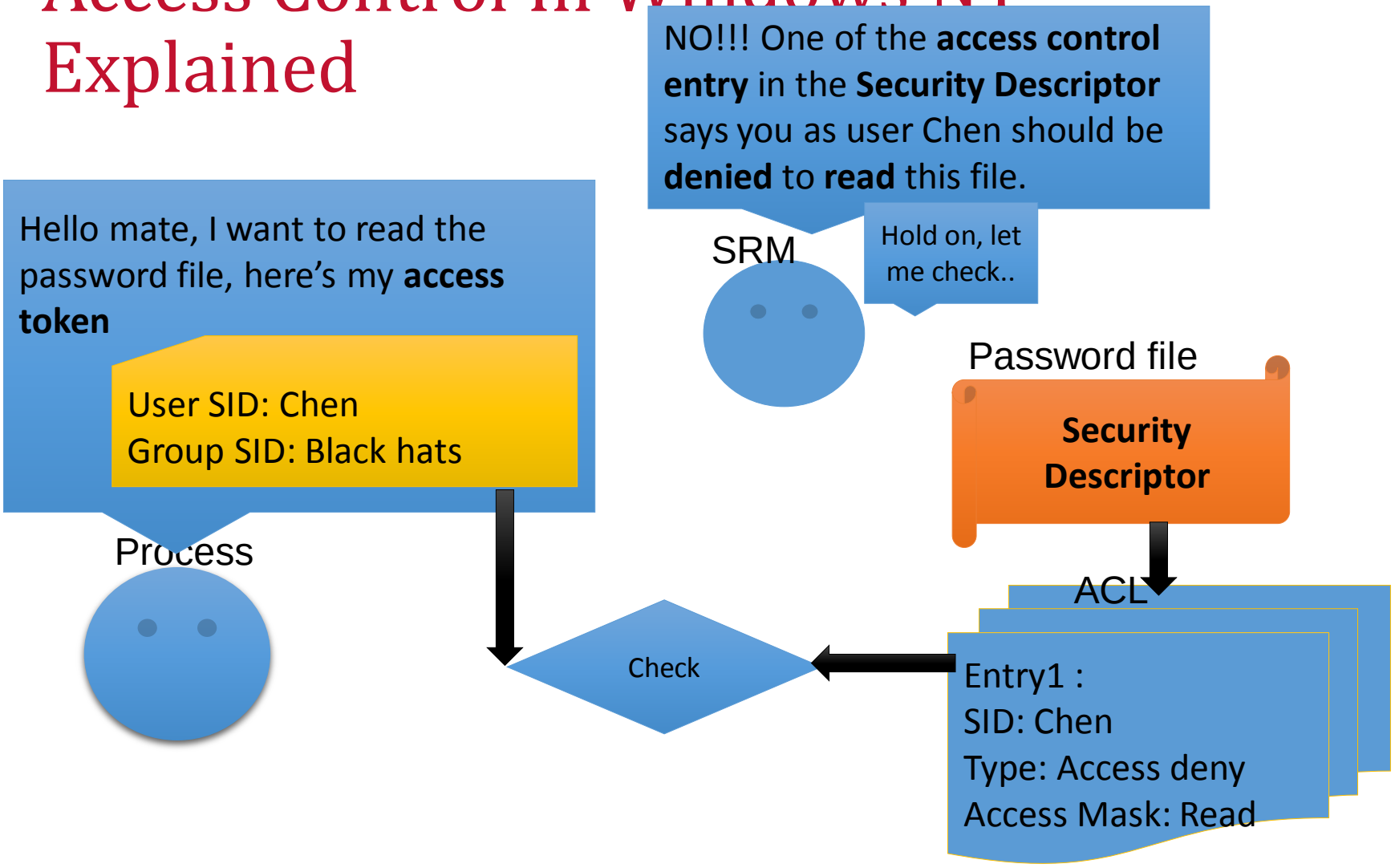
- Access Control Model
  - SubjectObject



*Storage Resource Management (SRM)

Graph from http://windowsitpro.com/security/q-windows-authorization-process-what-do-terms-access-token-security-descriptor-and-imperson

Access Control In Windows NT -- Explained



Access Control Entry (ACE)

Type	Inherit Flag	Access Mask	SID
Allow	Inherit_only	Read	Users (Chen)
Deny	No_Propagate	Write	Groups (admin)
Audit	Object_inherit	Execute	
	Directory_inherit	Create……	

Types of ACEs

- Access-denied
  - Used in an ACL to deny access
- Access-allowed
  - Used in an ACL to allow access
- System-audit
  - Used in an ACL to log attempts to access.

Access Control Entry (ACE)

Type	Inherit Flag	Access Mask	SID
Allow	Inherit_only	Read	Users (Chen)
Deny	No_Propagate	Write	Groups (admin)
Audit	Object_inherit	Execute	
	Directory_inherit	Create……	

Inherit Flags of ACEs

- Inherit_Only (For containers)
  - Only used for inheritance, not apply to this object
- No_Propagate (For containers)
  - Only Inherited onto sub-objects, but no further
- Object_Inherit (For objects)
  - Inherited onto sub-objects
- Container_Inherit (For containers)
  - Inherited onto sub-containers.

Access Control Entry (ACE)

Type	Inherit Flag	Access Mask	SID
Allow	Inherit_only	**Read**	Users (Chen)
Deny	No_Propagate	**Write**	Groups (admin)
Audit	Object_inherit	**Execute**	
	Directory_inherit	**Create……**	

Access Rights of ACEs

- Access Mask
  - Jointly-Used with the field **ACE types** and field **SID** when checking
  - **16-bit** long, can be turn on and off
  - Each bit corresponds to a specific access right.
- Example

/Chen's phone book

ACE	
SID:	**Chen**
Type:	**Access-allowed**
Access Mask:	**"Read + Write + Execute"**
Inherit Flag:	**"No_propagate + Directory_Inherit"**

Limitations In Windows NT

- Only support 16 different access rights
- Inherit flag does not distinguish between types of objects with different access rights
  - Containers and Non-Containers
- Propagating access control changes to a tree of objects will be ambiguous
  - Propagated change of ACE conflicts with locally added ACE
- No mechanism for restricting the rights of a process other than disabling privileges.

Access Control In Windows 2000

- What's new in ACL of Windows 2000

Type	Inherit Flag	Access Mask	Object Type	Inherited Object Type
Specify this ACE is for ALLOW/DENY purposes	Specify how this ace should be inherited	A mask to specify what kind of access rights this ACE is dealing with. e.g. Read, Write, Execute, Create,etc.	Identifies the type of object or property to which the ACE applies ***property** *explained in next slide*	Controls which types of objects can inherit the ACE

New feature In Windows 2000 – Property Sets

- Property Sets:
  - What is a property?
    - Attributes of an object, e.g Name, age and weight, ect
- Global Unique Identifier(GUID)
  - Microsoft used term for Universally Unique Identifier(UUID)
  - Each Access control target(objects, properties) will be assigned a GUID
- Property Sets are useful:
  - Properties could be grouped into property sets, identified by **ONE** GUID
  - Only ACEs with no GUID or matching GUIDs are evaluated.

New feature In Windows 2000 – Inheritance Control

- Annotation
  - A tag specified by sub-objects dealing with changes of the ACEs pass down from parent-objects
- Dynamic Inheritance Control
  - Without Annotation
- Static Inheritance Control
  - With Annotation
- Comparison
  - Centralized management access control
  - Space and time cost.

New feature In Windows 2000 – Protection from untrusted code

- Motivation
  - Let user decides which subject have access to certain objects
  - Limiting the damage caused by misbehaving subjects
- Introduction of ***restricted context***
  - A restricted context is an access token with a restriction
    - A new field in **Access Token**
    - Apply deny-only attribute to some SIDs that should be restricted
    - When access secure objects, double check SID and the restricted SID list
    - E.g Browser creates restricted thread to show webpage content.

Wrap Up

Windows NT	Windows 2000
Only support 16 different access rights.	Extended the length of mask
Inheritance does not distinguish between types of objects	Object-specific ACE has the filed “Inherited Object Type” to help differentiate that
Propagating access control changes to a tree of objects will be ambiguous	Using annotations and static inheritance to correctly propagate changed access control
No mechanism for restricting the rights of a process	Restricted context

Access control in Linux and Win 2000

	Linux	Windows 2000
Access rights	Read, write, execute	Support up to 32 different access rights
Inheritance	Mainly umask, but with setgid the objects inside can inherit	Support explicitly specified inheritance
ACE Types	Only have “allow”	Allow, deny, audit
Access control granularity	User level, controlled by uid	Thread level, controlled by restricted context in access token

UNIX / LINUX - USER ADMINISTRATION

In this chapter, we will discuss in detail about user administration in Unix.

There are three types of accounts on a Unix system –

Root account

This is also called **superuser** and would have complete and unfettered control of the system. A superuser can run any commands without any restriction. This user should be assumed as a system administrator.

System accounts

System accounts are those needed for the operation of system-specific components for example mail accounts and the **sshd** accounts. These accounts are usually needed for some specific function on your system, and any modifications to them could adversely affect the system.

User accounts

User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

Unix supports a concept of *Group Account* which logically groups a number of accounts. Every account would be a part of another group account. A Unix group plays important role in handling file permissions and process management.

Managing Users and Groups

There are four main user administration files –

- **/etc/passwd –** Keeps the user account and password information. This file holds the majority of information about accounts on the Unix system.
- **/etc/shadow –** Holds the encrypted password of the corresponding account. Not all the systems support this file.
- **/etc/group –** This file contains the group information for each account.
- **/etc/gshadow –** This file contains secure group account information.

Check all the above files using the **cat** command.

The following table lists out commands that are available on majority of Unix systems to create and manage accounts and groups –

Sr.No.	Command & Description
1	**useradd** Adds accounts to the system
2	**usermod** Modifies account attributes

3	**userdel** Deletes accounts from the system
4	**groupadd** Adds groups to the system
5	**groupmod** Modifies group attributes
6	**groupdel** Removes groups from the system

You can use Manpage Help to check complete syntax for each command mentioned here.

Create a Group

We will now understand how to create a group. For this, we need to create groups before creating any account otherwise, we can make use of the existing groups in our system. We have all the groups listed in ***/etc/groups*** file.

All the default groups are system account specific groups and it is not recommended to use them for ordinary accounts. So, following is the syntax to create a new group account –

```
groupadd [-g gid [-o]] [-r] [-f] groupname
```

The following table lists out the parameters –

Sr.No.	Option & Description
1	**-g GID** The numerical value of the group's ID
2	**-o** This option permits to add group with non-unique GID
3	**-r** This flag instructs **groupadd** to add a system account
4	**-f** This option causes to just exit with success status, if the specified group already exists. With -g, if the

	specified GID already exists, other (unique) GID is chosen
5	**groupname** Actual group name to be created

If you do not specify any parameter, then the system makes use of the default values.

Following example creates a *developers* group with default values, which is very much acceptable for most of the administrators.

```
$ groupadd developers
```

Modify a Group

To modify a group, use the **groupmod** syntax –

```
$ groupmod -n new_modified_group_name old_group_name
```

To change the developers_2 group name to developer, type –

```
$ groupmod -n developer developer_2
```

Here is how you will change the financial GID to 545 –

```
$ groupmod -g 545 developer
```

Delete a Group

We will now understand how to delete a group. To delete an existing group, all you need is the **groupdel command** and the **group name**. To delete the financial group, the command is –

```
$ groupdel developer
```

This removes only the group, not the files associated with that group. The files are still accessible by their owners.

Create an Account

Let us see how to create a new account on your Unix system. Following is the syntax to create a user's account –

```
useradd -d homedir -g groupname -m -s shell -u userid accountname
```

The following table lists out the parameters –

Sr.No.	Option & Description
1	**-d homedir** Specifies home directory for the account
2	**-g groupname**

	Specifies a group account for this account
3	**-m** Creates the home directory if it doesn't exist
4	**-s shell** Specifies the default shell for this account
5	**-u userid** You can specify a user id for this account
6	**accountname** Actual account name to be created

If you do not specify any parameter, then the system makes use of the default values. The **useradd** command modifies the **/etc/passwd**, **/etc/shadow**, and **/etc/group** files and creates a home directory.

Following is the example that creates an account ***mcmohd***, setting its home directory to ***/home/mcmohd*** and the group as ***developers***. This user would have Korn Shell assigned to it.

```
$ useradd -d /home/mcmohd -g developers -s /bin/ksh mcmohd
```

Before issuing the above command, make sure you already have the *developers* group created using the ***groupadd*** command.

Once an account is created you can set its password using the **passwd** command as follows –

```
$ passwd mcmohd20
Changing password for user mcmohd20.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

When you type ***passwd accountname***, it gives you an option to change the password, provided you are a superuser. Otherwise, you can change just your password using the same command but without specifying your account name.

Modify an Account

The **usermod** command enables you to make changes to an existing account from the command line. It uses the same arguments as the **useradd** command, plus the -l argument, which allows you to change the account name.

For example, to change the account name ***mcmohd*** to ***mcmohd20*** and to change home directory accordingly, you will need to issue the following command –

```
$ usermod -d /home/mcmohd20 -m -l mcmohd mcmohd20
```

Delete an Account

The **userdel** command can be used to delete an existing user. This is a very dangerous command if not used with caution.

There is only one argument or option available for the command **.r**, for removing the account's home directory and mail file.

For example, to remove account *mcmohd20*, issue the following command –

```
$ userdel -r mcmohd20
```

If you want to keep the home directory for backup purposes, omit the **-r** option. You can remove the home directory as needed at a later time.