

Control against Program threats

Operating System Security: Memory and Address protection, File Protection Mechanism, User Authentication. Linux and Windows: Vulnerabilities, File System Security.

Controls for Security

- How to control security of pgms during their development and maintenance
 - a. Developmental controls for security
 - b. Operating system controls for security
 - c. Administrative controls for security

Developmental Controls for Security (1)

- Nature of s/w development
 - Collaborative effort
 - Team of developers, each involved in ≥ 1 of these steps:
 - Requirement specification
 - Regular req. specs: „do X”
 - Security req. specs: „do X *and nothing more*”
 - Design
 - Implementation
 - Testing
 - Documenting
 - Reviewing at each of the above stages
 - Managing system development thru all above stages
 - Maintaining deployed system (updates, patches, new versions, etc.)
- Both product and process contribute to quality incl. security dimension of quality

Developmental Controls for Security (4)

- Techniques for building solid software
 - 1) Peer reviews
 - 2) Hazard analysis
 - 3) Testing
 - 4) Good design
 - 5) Risk prediction & mangement
 - 6) Static analysis
 - 7) Configuration management
 - 8) Additional developmental controls



...

Operating System Controls for Security (1)

- Developmental controls not always used

OR:

- Even if used, not foolproof

=> Need other, complementary controls, incl. OS controls

- Such OS controls can protect against some pgm flaws

Administrative Controls for Security (1)

- They prohibit or demand certain human behavior via policies, procedures, etc.
- They include:
 - 1) Standards of program development
 - 2) Security audits
 - 3) Separation of duties

Controls for Security

- Developmental / OS / administrative controls help produce/maintain higher-quality (also more secure) s/w
- *Art* and science - no „silver bullet“ solutions
- „A good developer who truly understands security will incorporate security into all phases of development.”

Control	Purpose	Benefit
Develop- mental	Limit mistakes Make malicious code difficult	Produce better software
Operating System	Limit access to system	Promotes safe sharing of info
Adminis- trative	Limit actions of people	Improve usability, reusability and maintainability

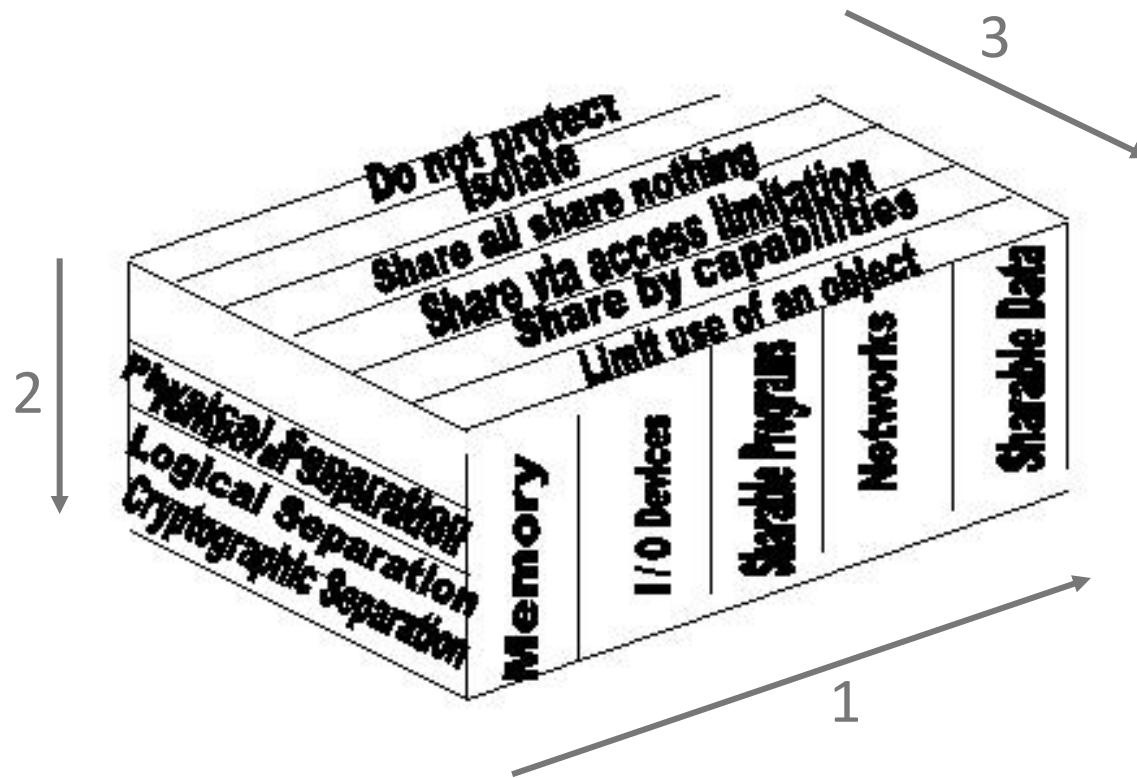
Three dimensions of protection in OSs

Dimensions:

1—protected objects

2—security methods

3—protection levels



Granularity of *data* protection

- Granularity of data protection

- Applicable only to data

- Protect by:

- Bit
- Byte
- Element/word
- Field
- Record
- File
- Volume



Ease of
implementation

Worse
(higher granularity)
data control (*)

(*) If no control at proper granularity level, OS must grant access to more data than necessary

E.g., if no field-level data control, user must be given whole record

Memory and Address Protection (1)

- Most obvious protection:
Protect pgm *memory* from being affected by other pgms
 - a. Fence
 - b. Relocation
 - c. Base/Bounds Registers
 - d. Tagged Architecture
 - e. Segmentation
 - f. Paging
 - g. Combined Paging with Segmentation

a. Fence

- Confining users to one side of a boundary
- E.g., predefined memory address n between OS and user

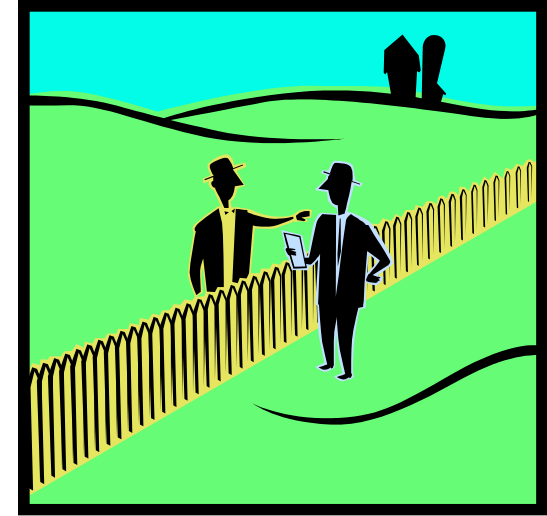
User pgm instruction at address $\leq n$ (OS's side of the fence) not allowed to execute

- *Fixed* fence (wastes space if unused by OS or blocks IOS from growing)

or

Variable fence

Using *fence register* — h/w register



b. Relocation

- Pgms written as if starting at location 0 in memory
- Actually, starting at location n — determined by OS
- Before user instruction executed, each address *relocated* by adding *relocation factor* n to it
 - Relocation factor = starting address of pgm in memory
- Fence register (h/w register) plays role of relocation register as well
 - Bec. adding n to pgm addresses prevents it from accessing addresses below n

c. Base/Bounds Registers

- **Base register** = variable fence register
 - Determines starting address, i.e. *lower limit*, for user pgm addresses
- **Bounds register**
 - Determines *upper limit* for user pgm addresses
- Each pgm address *forced* to be above base address
 - Bec. base reg contents added to it
- & each pgm address *checked* to be below bounds address
- To protect user's instructions from *user's own* data address errors – use *two pairs* of registers:
 - 1) Register pair for data
 - 2) Register pair for instructions




d. Tagged Architecture

- Problem with base/bounds registers:
high granularity of access rights (ARs)
 - Can allow another module to access *all or none* of its data
 - „All or none” data within limits of data base-bounds registers
- Solution: **tagged architecture** (gives *low* granularity of access rights)
 - Every word of *machine memory* has ≥ 1 **tag bits** defining access rights to this word (a h/w solution!)

(# of bits \sim # of different ARs)

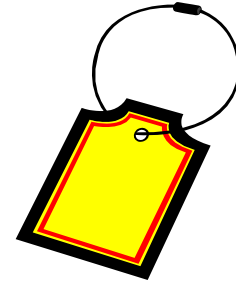
- Access bits set by OS
- Tested every time instruction accesses its location

Tag	Word
R	0001
RW	0137
R	4091
R	0002
X	

R = Read only

RW = Read/Write

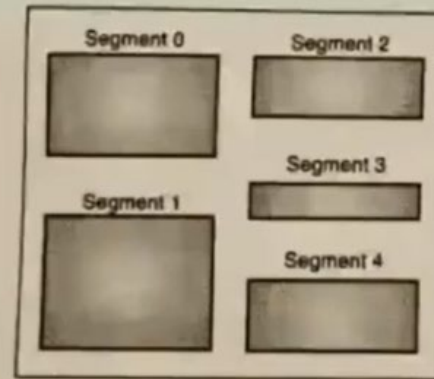
X = Execute only



- Benefit of tagged architecture:
Low (good!) granularity of memory access control
– at memory word level
- Problems with tagged architecture:
 - Requires special h/w
 - Incompatible with code of most OSs
 - OS compatible with it must:
 - Accommodate tags in each memory word
 - Test each memory word accessed
 - Rewriting OS would be costly
 - Higher memory costs (extra bits per word)
 - More modern solutions available (below)

e. Segmentation

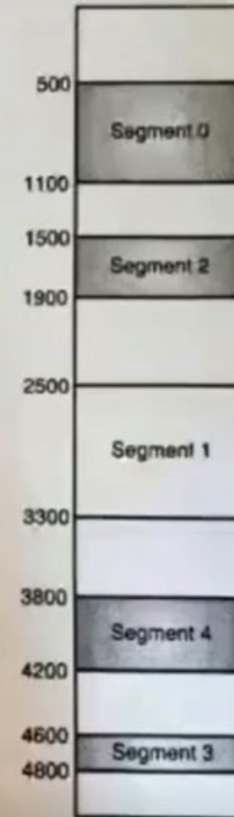
- Benefits addressing + enhances memory protection for free
- Effect of an unbounded number of base/bounds registers
- Pgm segmentation:
 - Program divided into logical pieces (called **segments**)
 - E.g. Pieces are: code for single procedure
/ data of an array / collection of local data values
 - Consecutive pgm segments can be easily stored in nonconsecutive memory locations



Logical address space

Segment Number		
	Base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment table



Physical address space

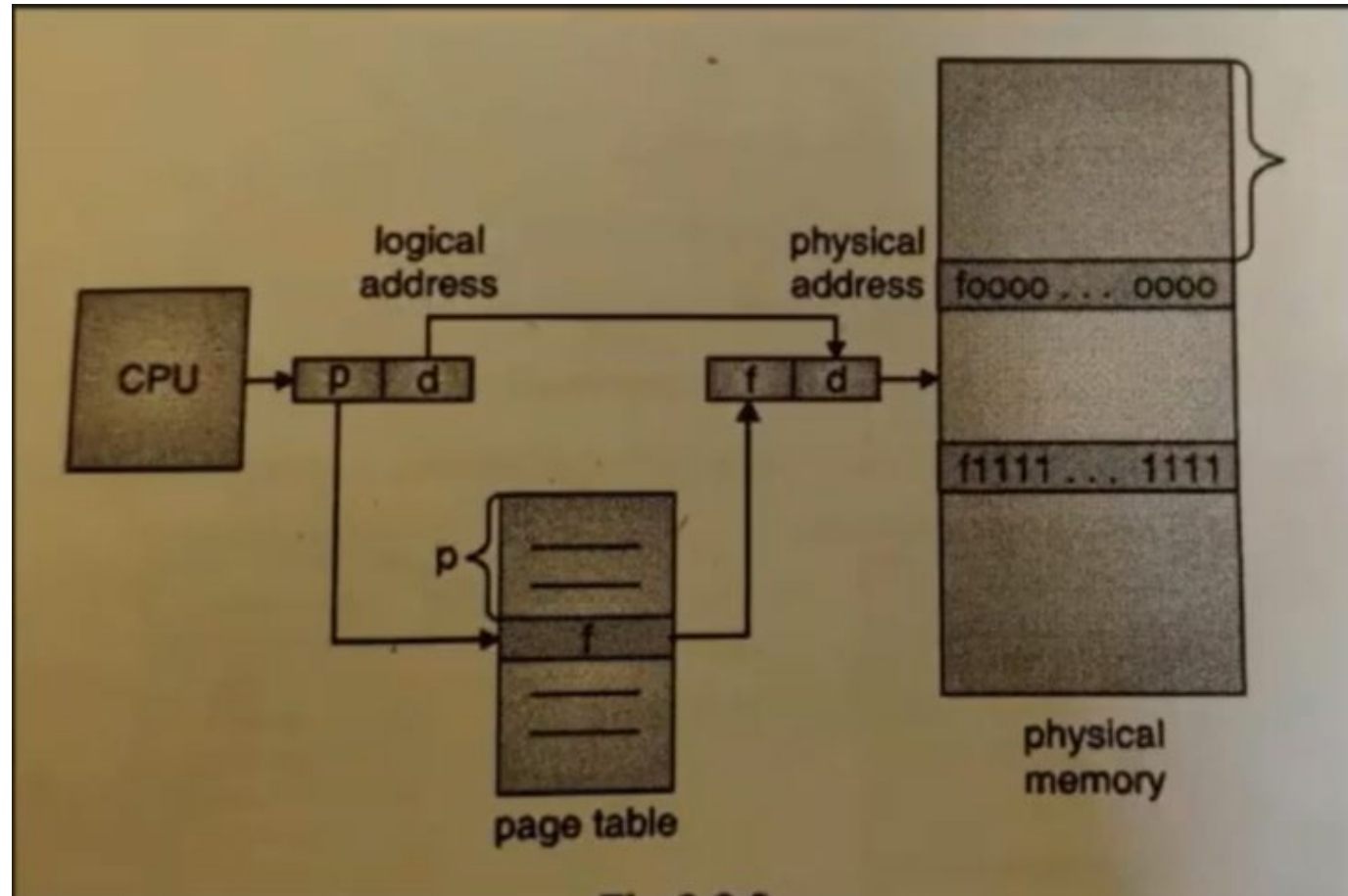
- Addressing w/ segmentation
 - Data item D addressed as:
(segment_name_of_D, offset_of_D_within_segment)
Instructions addressed analogously
 - For each process, OS keeps a separate
Segment Translation Table (STT)
Rows in STT: (segment_name, segment_offset)
segment_name – name of segment containing data item
segment_offset – starting location for named segment
 - OS translates each data or instruction address using STT
- Two processes can *share a segment S* by having the same segment_name and segment_offset value in their STTs

- Security-related benefits of segmentation
 - Strong segment protection
 - Bec.: STT under exclusive OS control
 - each address requires STT access to get segment_offset for segment S
 - OS checks that address translates into S's memory space (not beyond its end)
 - Different protection levels for different segments (approximates tagging at higher granularity)
 - E.g. segments with: R-only data / X-only code / W data
 - Different protection levels for different processes accessing the *same* segment

- Problems w/ segmentation
 - Programmer must be aware of segmentation
 - Efficiency
 - OS lookup of STT is slow
 - Symbolic segment names difficult to encode in pgm instructions
 - Fragmentation of main memory (by variable-sized holes left after „old” segments)

f. Paging

- Principles:
 - Programs divided into equal-sized *pages*
Memory divided into same-sized *page frames*
 - Size is usually 2^n , from 512 B to 4096 B
 - Address format for item (data or instruction) I:
(page_nr_of_I, offset_of_I_within_page)
 - OS maintains *Page Translation Table* (PTT)
 - maps pages into page frames
- Address translation similar as for segmentation
 - But *guaranteed* that offset falls within page limit
 - E.g., for page size of $1024 = 2^{10}$,
10 bits are allocated for page_offset



- Benefits of paging
 - Programmer can be oblivious to page boundaries (automatic)
 - Paging completely hidden from programmer
 - No fragmentation of main memory
- Problem w/ paging
 - Can't associate access rights with pages
 - Pages are *random collections of items* that require different protection level in general
 - Pages are *not 'access rights' units (logical units)* to be protected at the same level

f. Combined paging with segmentation

- Principle:
 - Paging offers efficiency
 - Hiding from programmer
 - No fragmentation
 - Segmentation offers 'logical protection'
 - Grouping items w/ similar protection needs within the same segment
- Paged segmentation:
 - Programmer defines segments
 - Segments broken into pages automatically
- Benefits of paging and segmentation
 - but* extra layer of address translation
 - Additional h/w deals with this overhead

File Protection Mechanisms

- a. Basic forms of protection
- b. Single file permissions
- c. Per-object and per-user protection

a. Basic forms of protection (1)

- Basic forms of protection

- 1) All-none protection
- 2) Group protection

1) All-none protection (in early IBM OS)

- Public files (all) or files protec'd by passwords (none)
 - Access to public files required knowing their names
 - Ignorance (not knowing file name) was an extra barrier
- Problems w/ this approach
 - Lack of trust for public files in large systems
 - Difficult to limit access to trusted users only
 - Complexity – for password-protected files, human response (password) required for each file access
 - File names easy to find
 - File listings eliminate ignorance barrier

Basic forms of protection (2)

2) Group protection

- Groups w/ common relationship:
 - I.e., group – if has need to share sth
 - User belongs to one group
 - Otherwise can leak info objects groups
- Example — In Unix: user, (trusted) group, others
 - E.g., u+r+w+x,g+r+w-x,o+r-w-x
- Advantage: Ease of implementation
 - OS recognizes user by user ID and group ID (upon login)
 - File directory stores for each file:
File owner's *user ID* and file owner's *group ID*

Basic forms of protection (3)

- Problems w/ group protection

- a) User can't belong to > 1 group

Solution: Single user gets multiple accounts

- E.g., Tom gets accounts Tom1 and Tom2
- Tom1 in Group1, Tom2 in Group2
- Problem: Files owned by Tom1 can't be accessed by Tom2 (unless they are public – available to 'others')

Problems: account proliferation, inconvenience, redundancy (e.g., if admin copies Tom1 files to Tom2 acct)

- b) User might become responsible for file sharing

E.g., admin makes files from all groups visible to a user (e.g., by copying them into one of user's accts and making them private user's files)

=> User becomes responsible for 'manually' preventing unauthorized sharing of his files between his different 'groups'

- c) Limited file sharing choices

Only 3 choices for any file: private, group, public

Single file permissions (1)

- Single permissions – associating permission with single file
- **Types** of single file permissions:
 - 1) Password or other token
 - 2) Temporary acquired permission

1) Password or other token

- Provide a **password** for each file
 - File pwd for **W** only
 - File pwd for **any** access
- Finer degree of protection
 - Like having a different group for each file
 - file X group = all those who know file X pwd

Single permissions (2)

- **Problems** with file pwds
 - **Loss** of pwd
 - Requires admin unprotecting file, then assigning new
 - Requires notifying all legitimate users
 - Using them **inconvenient**, takes time
 - Pwd **disclosure** allows unauthorized file accesses
 - Change of pwd requires notifying all legitimate users
 - **Revocation** of (just) a single user requires pwd change
 - Then, must notify all legitimate users

2) Temporary acquired permission

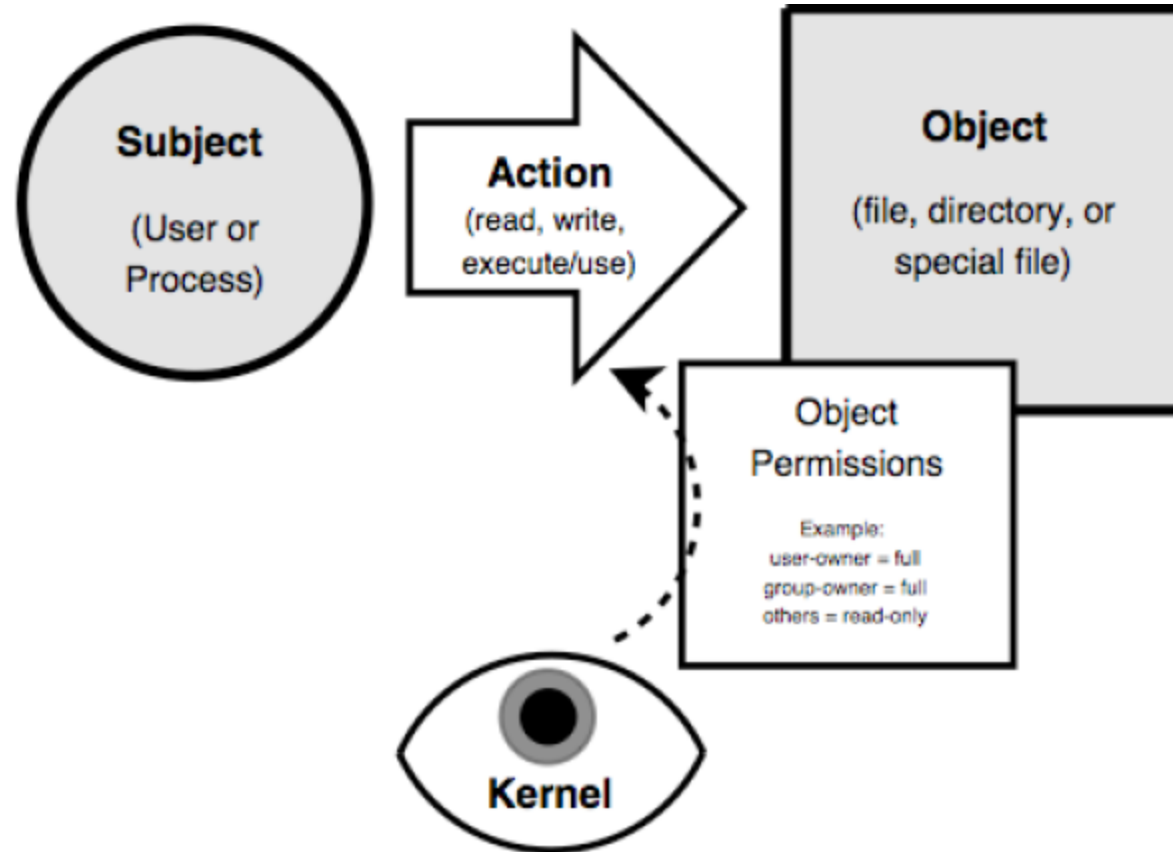
- Used in UNIX – the approach:
 - Based on **user-group-others** access hierarchy
 - Permission called *set userid* (**suid**)
 - If „user” (owner) of executable file X sets *suid* for X for his group, any group member executing X has „user” access rights (ARs) for X
 - Rather than having just „regular” group ARs for X
- Allows users to share data files
 - Access only via procedures that access them
 - Procedures encapsulate files
 - E.g., convenient for OS pwd file
 - Pwd change pgm with suid - any user can access own pwd record
 - OS owns this pgm (only OS, as „user” can access whole pwd file)

Per-object and per-user protection

- Per-object and per-user protection
 - Approach:
 - File owner specifies access rights (ARs) for *each file* he owns for *each user*
 - Can implement with ACL (access control list) or ACM (access ctrl matrix)
 - Advantages:
 - Fine granularity of file access
 - Allows to create groups of users with similar ARs
 - Problem: Complex to create and maintain groups
 - File owner's overhead to specify ARs for *each file* for *each user* he owns

Linux and Windows: Vulnerabilities, File System Security.

Linux Security Transactions



File System Security

- In Linux everything is a file
- I/O to devices is via a “special” file
 - Example: /dev/cdrom points to /dev/hdb which is a special file
- Have other special files like named pipes
 - A conduit between processes / programs
- Since almost everything a file – security very important

Users and Groups

- Users and Groups are not files
- Users
 - Someone or something capable of using files
 - Can be human or process
 - e.g. lpd (Linux Printer Daemon) runs as user lp
- Groups
 - List of user accounts
 - User's main group membership specified in `/etc/passwd`
 - User can be added to additional group by editing `/etc/group`
 - Command line -> `useradd`, `usermod`, and `userdel`

Understanding: /etc/password

Purvi:x:1021:1020:EECS:/home/purvi:bin/bash

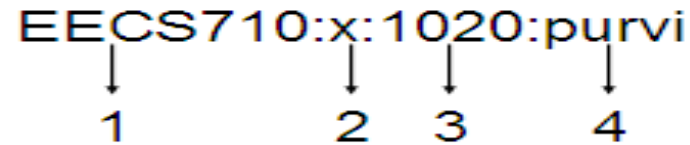
↓ ↓ ↓ ↓ ↓ ↓ ↓

1 2 3 4 5 6 7

1. username: Used when user logs in. It should be between 1 and 32 characters in length
2. password: An x character indicates that encrypted password is stored in /etc/shadow file
3. user ID (UID): Each user must be assigned a user ID (UID). UID 0 (zero) is reserved for root and UIDs 1-99 are reserved for other predefined accounts
 - UID 100-999 are reserved by system for administrative and system accounts/groups
4. group ID (GID): The primary group ID (stored in /etc/group file)
5. user ID Info: The comment field
 - Allows you to add extra information about the users such as user's full name, phone # etc
 - This field used by finger command
6. home directory: The absolute path to the directory the user will be in when they log in
 - If this directory does not exists then users directory becomes /
7. command/shell: The absolute path of a command or shell (/bin/bash)
 - Typically, this is a shell. Please note that it does not have to be a shell.

Understanding of /etc/group

EECS710:x:1020:purvi



1 2 3 4

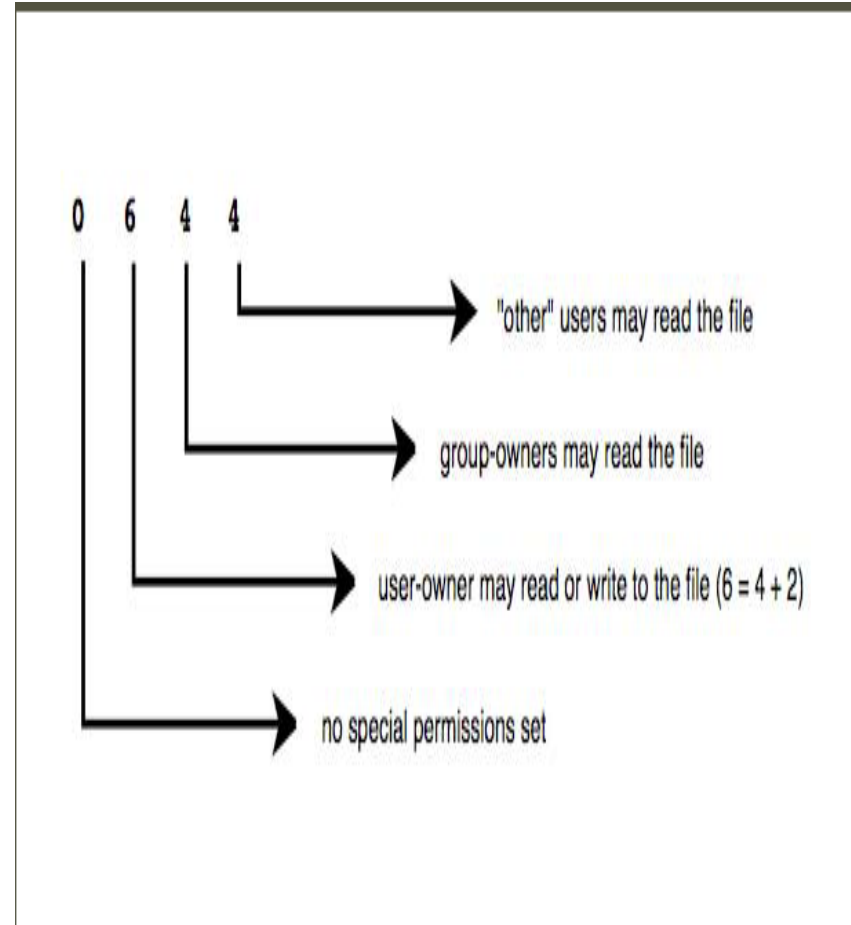
1. group_name: Name of group
2. password: Generally password not used, hence it is empty/blank. It can store encrypted password. Useful to implement privileged groups
3. group ID (GID): Group ID must be assigned to every user
4. group List: List of user names of users who are members of the group. The user names must be separated by commas

File Permissions

- Files have two owners: a user & a group
 - Each with its own set of permissions
 - With a third set of permissions for other
- Permissions are to read/write/execute in order user/group/other
 - `rw-rw -r-- 1 maestro user 35414 Mar 25 01:38 baton.txt`
- Permission can be changed using `chmod` command

Numeric File Permissions

- Read (r) = 4
- Write (w) = 2
- Execute (x) = 1
- Example :
drwxr-x--- 8 biff drummers
288 Mar 25 01:38
extreme_casseroles



Directory Permissions

- Permissions on folder slightly works different
 - read = list contents
 - write = create or delete files in directory
 - execute = use anything in or change working directory to this directory
- Example from textbook:

```
$ chmod g+rx extreme_casseroles  
$ ls -l extreme_casseroles  
drwxr-x--- 8 biff drummers 288 Mar 25 01:38 extreme_casseroles
```

Difference between File and Directory Permissions

Access Type	File	Directory
Read	If the file contents can be read	If the directory listing can be obtained
Write	If user or process can write to the file (change its contents)	If user or process can change directory contents somehow: create new or delete existing files in the directory or rename files.
Execute	If the file can be executed	If user or process can access the directory, that is, go to it (make it to be the current working directory)

....

- SetUID and SetGID
- SetGID and Directories

Kernel Space and User Space

- Kernel space: refers to memory used by the Linux kernel and its loadable modules (e.g device drivers)
- User space: refers to memory used by all other processes
- Since kernel enforces Linux DAC and security, its extremely critical to isolate kernel from user space
 - For this reason, kernel space never swapped to disk
 - Only root may load and unload kernel modules

Mandatory Access Controls

- Linux uses a DAC security model
- Mandatory Access Controls (MAC) imposes a global security policy on all users
 - Users may not set controls weaker than policy
 - Normal admin done with accounts without authority to change the global security policy
 - But MAC systems have been hard to manage
- Novell's SuSE Linux has AppArmor
- RedHat Enterprise Linux has SELinux
- "pure" SELinux for high-sensitivity, high-security

Windows Vulnerabilities

- Windows like all other OS has security bugs
 - Bugs have been exploited to compromise customer accounts
- Multiple versions of Windows
 - Each with substantial user-base
- Attackers are now (organized) criminals highly motivated by money
- Microsoft Security Bulletin Summaries and Webcasts provides latest vulnerabilities list and relative security updates (and status)

Linux Vulnerabilities

- Default Linux installations (un-patched and unsecured) have been vulnerable to
 - Buffer overflows
 - Race conditions
 - Abuse of programs run “SetUID root”
 - Denial of Service (DoS)
 - Web application vulnerabilities
 - Rootkit attacks