Assignment 4 CS 344

Name: Vignesh Ravichandra Rao **Roll Number**: 190101109

Topic: Protection and Security (Linux OS)

Video: https://youtu.be/Ac4V7Ey25CE

1. Introduction

Protection is a mechanism to control access to files, process and users defined in an operating system. It prevents improper use of the system by users and programs.
Security is the guarding of OS resources against unauthorised access, malicious destruction and data inconsistencies. Threats to a system can be device theft and subsequent access to user's files, network based threats like viruses and worms. To summarise, protection deals with separating resources between different users and processes while security involves protecting the resources against external threats.

2. Comparison between Linux OS and xv6

The Linux Operating system has several security and protection measures in place. These include:

- **User Accounts and Authentication** : Every person using the Linux OS can create an account which is password protected. Upon booting, the user must log into an existing account with correct password to use the OS.
- **File Permissions** : Every file created by a user has different access permissions which can be set by the owner of the file. This permissions control which processes/users can read, write and execute files. This is implemented using Access Control Lists and Permission bits within the file inode structure.
- **Netfilters** : This is an IP Layer framework that examines all packets that pass through the device NIC and can filter out malicious packets and act as a firewall.
- **Memory based protection**: The Linux kernel does not allow user programs to access the kernel address space. Moreover, user programs can not load kernel modules and make references to out of range pages.

In xv6:

- There is no notion of a user and anyone who boots the system has access to all files. (Difference from Linux)
- The kernel uses CPU's hardware protection mechanism to ensure a user program only accesses it's user space memory. (Similarity with Linux)
- The kernel pages and stack are protected from access by user programs by unsetting the PTE_U bit in the page table entry for kernel processes. Hence, user programs can not access kernel functions. (Similarity with Linux)

- The file system is not protected from unauthorised access and there is no idea of permissions for files/processes. (Difference from Linux)
- xv6 uses protection level 0 for kernel mode and level 3 (least privilege) for user mode. (Similarity with Linux)

3. Strategy

We will implement one security and one protection feature from Linux in xv6. These features are:

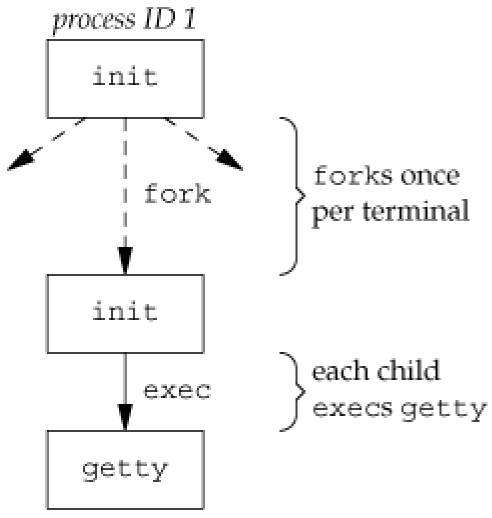
- **Security**: Users and User Authentication
- **Protection**: Filesystem protection using permissions

We will emulate the Linux kernel implementation for these aspects by first studying the Linux implementation and then defining how we will implement the same in xv6 – the data structures, system calls, etc.

3.1 Users and User Authentication

Linux OS implementation:

1. Users are assigned unique uid (user identifier) and have an associated user-defined password. These details are stored in a file called /etc/passwd
2. There exists a root user which has access to all system calls, processes and files by default.
3. The useradd command is used to create new users. The passwd command is used to create/update passwords for users. Both these commands can be invoked only by root user.
4. After booting, before the sh (shell) process is forked, the user is authenticated by the getty process. The image on the right shows the process tree at this point in a Linux OS.



5. getty acts as a login prompt and executes the shell process only if user enters correct password (authentication complete)

Proposed xv6 implementation

- A **/passwd** file will be created in the root directory to store contents similar to Linux OS like username, password, uid etc. (Since the assignment is not on filesystem, we will not be recreating the Linux directory structure)
- The /passwd file will initially contain an entry for root user with username = root and password = 123456
- In order to create a new user the following user program will be created:
  - **useradd.c** : Accepts username as command line argument and adds an entry for the user in the /passwd file.
  - **passwd.c** : Accepts username and password as command line arguments and updates password for corresponding user.
  - Note that we are creating user programs instead of system calls because, once we implement filesystem permissions we can enforce the

condition that only root can run these commands. Also, the open and close operations on files isn't implemented as a kernel function but as a system call.

- The **getty.c** process will be created and forked from the init.c process.
    - getty.c is the login prompt. It will ask for a username and password, confirm if the username, password pair is present in the /passwd file and then executes **sh.c** using **exec()** system call, as shown in the diagram.

3.2 Filesystem Permissions

In Linux:

1. Here, Linux introduces the concept of groups – a collection of users. Every group also has a group name, group id (gid) and the list of group members stored in **/etc/group** file.
2. Every file has permission bits in Linux. Permissions are broken into three domains, user – owner of the file, group – the group to which the file belongs and others – the rest of the users that are not in the group or user category. For each domain, the **rwx** (read-write-execute) permission bits are stored as an octal number. For example, if we wish to represent permission to read and execute the rwx bits would be 101 = $(5)_8$ in octal.
    - In total, one octal number represents the permissions for each domain and the three octal numbers together form the permission mode.
    - For example, if a file has permission mode = 752 it means the user(owner) can read,write and execute, the group can read and execute and others can only write.
3. The permission information for every file is stored in its respective inode structure in Linux.
4. In Linux, every process executes on behalf of a user and has a uid associated with it.
5. Processes also have an effective uid. A process' effective uid is used to check if it has permission to perform an operation on a file. Effective uid is set to the uid of the owner of the file if the file allows for setuid. Hence, with setuid, the process can effectively get permissions of the owner of the file.
6. The **setuid()** system call is used to change the effective uid of a process. Only a super user (root) can successfully invoke this function. Similarly, for the **setgid()** system call.
7. The **chmod()** system call is used to change the permission access mode of a file. This call is successful only if invoking process has superuser privilege. There is also a chmod command that uses this system call to allow users to change permissions through the terminal.
8. The **chown** system call and command are used to change the owner uid of a file in Linux.

9. The **groupadd** command creates a new group. The **usermod** command can be used to add a user to a group. Lastly, the **chgrp** command is used to change the group ownership of a file.

Implementation strategy for xv6:

- Create a **/group** file similar to the **/passwd** file described in secion 3.1
- Store permission access mode in the **struct inode** for every file. Also, include **uid, gid** and **effective uid** fields in struct proc.
- Create the following system calls:
  - **int setuid(int uid);**
    - Changes the uid of current process
    - Returns: 0 if successful and -1 if unsuccessful
  - **int setgid(int gid);**
    - Changes the gid of current process.
    - Same return value as setuid()
  - **int chown(char* path, int uid);**
    - Changes the owner uid of file at path.
    - Input:
      - path : complete path to file.
      - uid : uid of new owner.
    - Returns: 0 if successful, -1 if not.
  - **int chmod(char* path, int mode);**
    - Changes the access permissions to mode.
    - Input:
      - path: full path to file.
      - mode: new permission bits to set
    - Returns: 0 if successful, else -1
  - **int chgrp(char* path, int gid);**
    - Same as chown but for groups.
- Create the following user programs that serve as commands
  - **groupadd.c** : Takes groupname as command line argument and adds a new group with a unique gid in /group file.
  - **usermod.c** : Takes username and groupname as command line arguments and adds user into specified group.
  - **chmod.c, chgrp.c** and **chown.c** which make calls to the respective system calls.

4. Code Level Implementation Details

Note: Pseudocode will be used to explain some parts of the code.
Also, for all user programs, it is assumed that appropriate changes are made to Makefile to include the object files. Similarly, for all system calls, it is assumed that changes are made to syscall.h, syscall.c, usys.S and user.h

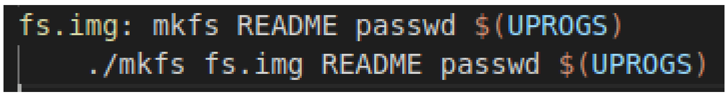
4.1 User Authentication

4.1.1 The /passwd file

The /passwd file will store information about users in a format similar to Linux. Each line will have the details of a user in the following format:

<username>:<password>:<uid>:<private-gid>
We need this file to be present in the filesystem right after boot.

- First, add a file called **passwd** in the xv6 source code directory.
- In this passwd file create an entry for the root user:
    - root:123456:0:0
        - Indicating that root password is 123456 and uid = 0 and gid = 0
- In the **Makefile**, we now need to add this file as a dependency for kernel image creation fs.img.



```
fs.img: mkfs README passwd $(UPROGS)
	./mkfs fs.img README passwd $(UPROGS)
```

4.1.2 useradd.c

This user program is responsible for adding new users into the system. Steps followed by useradd.c are:

1. Take username as input
2. Assign a unique uid, private gid for the user
3. Add a line in the /passwd file for the new user in specified format.

Note that every user creation is accompanied by the creation of a private group with same name as the username in Linux. This is known as the user private group. We will deal with groups in section 4.2.
Format of command: **useradd <username>**
Pseudocode :-
useradd.c:

```
Command Line Arguments: username
passwdFile = OPEN("/passwd",READ|APPEND_MODE)
groupFile = OPEN("/group",READ_ONLY_MODE)
uid = minimum unused UID in passwdFile
gid = minimum unused GID in groupFile
passwdFile.WRITE("%username::%uid,:%gid")
passwdFile.CLOSE()
```

4.1.3 passwd.c

This user program is used to change the password of an existing username.
Format of command: **passwd <username> <new_password>**
Note that in the pseudocode we refer to `line[password]` to refer to the password field in a line of the /passwd file. Similar notation will be followed for other fields. This can be achieved by tokenizing each line (**strtok**) with ‘:’ as delimiter. The following is its pseudocode:
passwd.c:

```
Command Line Arguments : uname, pass
passwdFile = OPEN("/passwd",READ_WRITE_MODE)
FOR EACH line IN passwdFile
    IF line[username] == uname
        line[password] = pass
        passwdFile.WRITE(line) // Update password field and rewrite
    ENDIF
ENDFOR
passwdFile.CLOSE()
```

4.1.4 getty.c

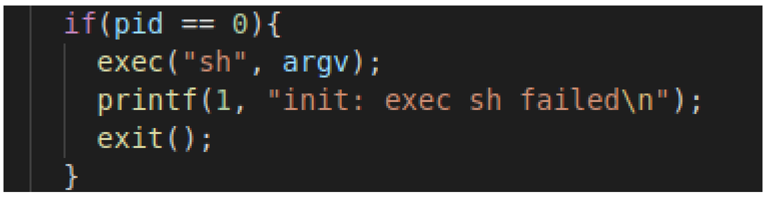
This user program is forked from init process and acts as a login prompt.
It executes the sh.c process, which is the terminal/shell only if the username and password are correct.

getty.c:

```
WHILE (True)
    uname = INPUT()
    pass = INPUT()
    passwdFile = OPEN("/passwd",READ_ONLY_MODE)
    FOR EACH line IN passwdFile
        IF (uname == line[username] AND pass == line[password])
            setuid(line[uid])
            setgid(line[gid])
            exec("/sh") // Execute shell if authentication success
        ENDIF
    ENDFOR
    PRINT("Authentication Failed, try again")
ENDWHILE
```

4.1.5 init.c

Originally, the init proc will fork a child and make it execute the shell process. However, since we want to authenticate before sh.c is run, we need to instead execute getty.c. Notice control is transferred between processes in the following order: init ---> getty ---> sh



```
if(pid == 0){
  exec("sh", argv);
  printf(1, "init: exec sh failed\n");
  exit();
}
```

The above exec() system call would have to be replaced with exec(“getty”,argc)

4.1.6 allocproc() and fork()

allocproc() is used to initialise a new process. Since, this function is the starting point for all new processes, we need to intialise the process uid, euid and gid to ROOT_USER (that is uid = 0) in this function.
Lastly, after init process is created, new process are created via forking which is done by the fork() system call. Hence, we need the child process to inherit the parent process’ uid, euid and gid in this function.

4.1.7 struct user

This is an optional step. A structure can be created to store all user details:

struct user {
    char name [25]; char password[25]
    int uid;
    int gid;
}

This **struct user** can be added into **struct proc** (in Section 4.2.4) instead of adding separate fields for each. This will modularise the code and this struct user can also be <u>used as return type while extracting details from /passwd</u>.
This completes User Authentication in xv6.

4.2 Filesystem Access Permissions

4.2.1 The /group file

The **/group** file will store information about groups in a format similar to Linux.
Each line will have the details :

<group_name>:<gid>:<members>

The list of members is comma separated.
This file is created and added to the initial xv6 filesystem in the same way /passwd was added in 4.1.1 (by changing the **Makefile**)
Initially, the /group file contains the following entry for the root's private group
root:0:root

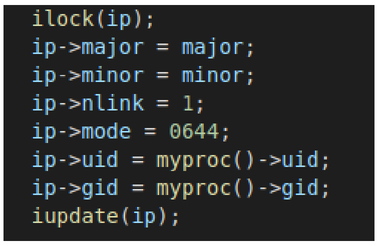
4.2.2 struct inode

It is necessary to store some ownership and permission related details for every file. We know that the inode stores metadata related to the file. Hence, we add a few more files in **struct inode**:

- int **uid** : This is the uid of the owner of the file
- int **gid**: This is the gid of the group that own the file
- int **mode**: These are the permission bits associated with the file

These fields must also be added to **struct dinode** which is the inode that is stored in disk.
Now, we have ensured that both disk and in-memory copies of an inode contain permission related information.

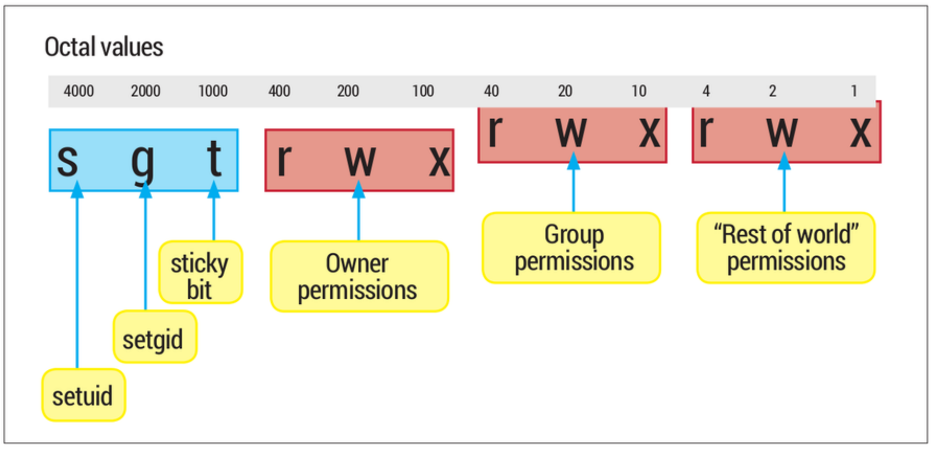
- **mkfs.c** : This is the make filesystem code. It creates the filesystem for xv6 on disk and has a function called ialloc();
    - This function should now be updated to have the prototype:
    - **ialloc(ushort type, int uid, int gid);**
    - The function should set the uid and gid fields of the inode created and give default permission of $(755)_8$ with setuid = 0. Since this function is called to mount the filesystem for the first time only, these default permissions allow all user programs to be executed by all users.
- **sysfile.c** : The **create()** function is used to allocate a new inode for a file. Make sure to assign an inode's uid,gid and default mode in this function.



```
ilock(ip);
ip->major = major;
ip->minor = minor;
ip->nlink = 1;
ip->mode = 0644;
ip->uid = myproc()->uid;
ip->gid = myproc()->gid;
iupdate(ip);
```

4.2.3 int mode

We have already seen how octal numbers are used to indicate permissions. Also, the int mode field of an inode is used to store these permissions.



We will use the same bits shown in the image for storing permissions as well. Define each bit as follows:

```
#define U_READ 1<<8      #define G_READ 1<<5      #define O_READ 1<<2
#define U_WRITE 1<<7     #define G_WRITE 1<<4     #define O_WRITE 1<<1
#define U_EXEC 1<<6      #define G_EXEC 1<<3      #define O_EXEC 1<<0
#define SETUID 1<<11
```

Now, we can use bitwise operators to add and remove permissions.
To add a permission, use the BITWISE OR and to remove perform BITWISE AND with the BITWISE NOT of the permission bit.
Eg: To add Group Execute permission: mode | G_EXEC
To remove Others Read permission: mode & ~O_READ

4.2.4 struct proc

Each process runs on behalf of a user/group .Hence add the following fields in **struct proc**:

- int **uid** : Real UID of the process
- int **euid**: Effective UID of the process used when SETUID is used.
- int **gid**: GID of the process

Otherwise add **struct user** (Section 4.1.7) and **int euid** in **struct proc.** Either way, user information is now stored in each process as well for comparisons to be made during file access.

4.2.5 setuid and setgid System Calls

The setuid system call can be used only by the root user. It changes the UID of the current running process.

sys_setuid()
```
    newUID = ARGINT(0) // Fetch argument from user stack
    IF myproc()->uid == ROOT_UID
        myproc()->uid = newUID
        RETURN 0
    ENDIF
    RETURN -1
```
Similarly, setgid is used to change the gid of current process and is a direct extension of setuid()

Recall that in section 4.1.4 these system calls are used. Getty runs as ROOT user until authentication is complete and hence has to change the uid and gid to that of logged in user.

4.2.6 Enforcing Access Control

Read Permission: Add a function **int allowRead(char* path)** which returns 1 if current user can real file at path or 0 if user can't read. Pseudocode:

```
int allowRead(char* path)
    ownUID = fileInode->uid
    ownGID = fileInode->gid
    mode = fileInode->mode
    fileInode = nameiparent(path) // Get inode
    IF fileInode->mode & SETUID
        myproc()->euid = ownUID
    ENDIF
    fMembers = list of UIDs belonging to ownGID group in /group
    currUID = myproc()->euid
    currGID = myproc()->gid
    IF currUID == ROOT_UID // Root can read all files
        RETURN 1
    ELSE IF currUID == ownUID AND mode & U_READ // Owner can read ?
        RETURN 1
    ELSE IF currGID == ownGID AND mode & G_READ // Group can read ?
        RETURN 1
    ELSE IF currUID IN fMembers AND mode & G_READ // Group can read ?
        RETURN 1
    ELSE if mode & O_READ // Others can read ?
        RETURN 1
    ENDIF
    RETURN 0
```

Use allowRead() in **sys_open()** to either allow or disallow read operation when file open mode is **O_RDONLY** or **O_RDWR**

Write Permission: Add a function **int allowWrite(char* path)** which returns similar to allowRead(). Also all the permissions checks done by this function are identical to allowRead() but with **U_WRITE, G_WRITE and O_WRITE** permission bits instead.

Use allowWrite() in **sys_open()** when the file open mode = **O_CREATE, O_RDWR or O_WRONLY** to allow or disallow write operation on file.

Execution Permission: Add a function **int allowExec(char *path)** which returns similar to above to functions. Even the permissions checks are identical but with **U_EXEC, G_EXEC** and **O_EXEC** instead.

allowExec() should be used in **exec.c** to allow or disallow user programs' execution.

4.2.7 chown and chgrp System Calls/User Programs

The chown system call is used to change the owner uid of a file. This system call is successful only for the root user.

```
int sys_chown()
    fileInode = nameiparent(path)
    IF myproc()->uid != ROOT_UID AND myproc()->uid != fileInode->uid
```

```
        RETURN -1
    ENDIF
    path = ARGSTR(0)
    newOwner = ARGINT(1)
    fileInode->uid = newOwner
    RETURN 0
```

Create a file chown.c for the chown user program. This user program will be used as a command in the xv6 terminal.
The format of this command is **chown <owner_username> <file_path>**

chown.c:

```
    Command Line Arguments: username, path
    fp = OPEN("/passwd",READ_ONLY_MODE)
    FOR EACH line IN fp
        IF line[username] == username
            newUID = line[uid]
        ENDIF
    ENDFOR
    chown(path,newUID)
```

The **chgrp()** system call and **chgrp.c** user programs are direct extensions of the above but with changes being made to gid of a file.

4.2.8 chmod System Call and User Program

The chmod system call is used to change the access permission mode of a file. Note that in order to intrepret numbers as Octal numerals, we need to prepend a 0 before the number. For example, setuid = 1 and accessmode = 754 will be expressed as **0**4754.

```
int sys_chmod()
    fileInode = nameiparent(path)
    IF myproc()->uid != ROOT_UID AND myproc()->uid != fileInode->uid
        RETURN -1
    ENDIF
    path = ARGSTR(0)
    newMode = ARGINT(1)
    fileInode->mode = newMode
    RETURN 0
```

The **chmod.c** user program just makes a call to the chmod() system call. It takes the path and mode as command line arguments. This is similar to chown.c above.
The format of the command is : **chmod <mode> <file_path>**
This completes kernel space management of file permissions. Some additional user programs for utility are:

4.2.9 groupadd.c

This userprogam will be used to add a new group in the OS. The format of the command is **groupadd <group_name>**
The implementation is similar to useradd.c.

groupadd.c

```
Command Line Argument: groupname
groupFile = OPEN("/group",APPEND_MODE)
gid = next available unused gid
currUser = username of myproc()->uid in /passwd
groupFile.WRITE("%groupname:%gid:%currUser")
```

4.2.9 usermod.c

This userprogam will be used to add a user into a group.

Format: **usermod <username> <groupname>**

`usermod.c`

```
Command Line Arguments: username, groupname
groupFile = OPEN("/group",READ_WRITE_MODE)
FOR EACH line IN groupFile
    IF line[groupname] == groupname
        line[members].APPEND(username)
        groupFile.WRITE(line)
    ENDIF
ENDFOR
```