Assignment 4

Protection and Security in Linux and xv6

YouTube link - https://youtu.be/g_s9zHzunCM

Abhishek Agrahari

190123066

Comparison between xv6 and Linux protection and security features

- Linux protection and security features can be classified in two groups

1. Authentication
2. Access Control

Authentication

- Definition – Making sure that nobody can access the system without first proving that she has entry rights.

xv6

- No authentication is done, so anybody can access the system and without requiring any entry rights.

Linux

- Authentication is typically been performed through the use of a publicly readable password file.
- A user's password is combined with a random "salt" value, and the result is encoded with a one-way transformation function and stored in the password file.
- When a user presents a password to the system, the password is recombined with the salt value stored in the password file and passed through the same one-way transformation. If the result

matches the contents of the password file, then the password is accepted.

- A new security mechanism of *pluggable authentication modules (PAM) system* is based on a shared library that can be used by any system component that needs to authenticate users.
- PAM allows authentication modules to be loaded on demand as specified in a system-wide configuration file. If a new authentication mechanism is added at a later date, it can be added to the configuration file, and all system components will immediately be able to take advantage of it.
- PAM modules can specify authentication methods, account restrictions, session-setup functions, and password-changing functions (so that, when users change their passwords, all the necessary authentication mechanisms can be updated at once).

Access Control

- Definition – Providing a mechanism for checking whether a user has the right to access a certain object and preventing access to objects as required.

xv6

- no access control mechanism, so anybody can access any file the any checks.

Linux

- Access control under UNIX system, including Linux, is performed through the user of unique numeric identifiers. A user identifier (UID) identifies a single user or a single set access right. A group identifier (GID) is an extra identifier that can be used to identify rights belonging to more than one user.
- Access control is applied to various objects in the system. Every file available in the system is protected by the standard access-memory access control mechanism. In addition, other shared

objects, such as shared-memory sections and semaphores, employ the same access system.

- Every object in a UNIX system under user and group access control has a single UID and single GID associated with it. User processes also have single UID, but they may have more than one GID. If a process's UID matches the UID of an object, then the process has user rights or owner rights to that object. If the UID's do not match but any GID of the process matches the object's GID then group rights are conferred; otherwise, the process has world right to the object.
- Linux performs access control by assigning objects a protection mask that specifies which access modes-read, write, or execute – are to be granted to processes with owner, group, or world access.
- The only exception is the privileged root UID. A process with this special UID is granted automatic access to any object in the system, bypassing normal access checks. Such processes are also granted permission to perform privileged operations, such as reading any physical memory or opening reserved network sockets. This mechanism allows the kernel to prevent normal users from accessing these resources: most of the kernel's key internal resources are implicitly owned by the root UID.

Two missing features of Linux's protection and security component that I will add in xv6 would be –

- **Authentication** – We will implement login prompt where user have to give his/her username and password.
- **Access Control** – After logged in, user would have restricted access rights. User can only read, write or execute files if he/she has the corresponding right.

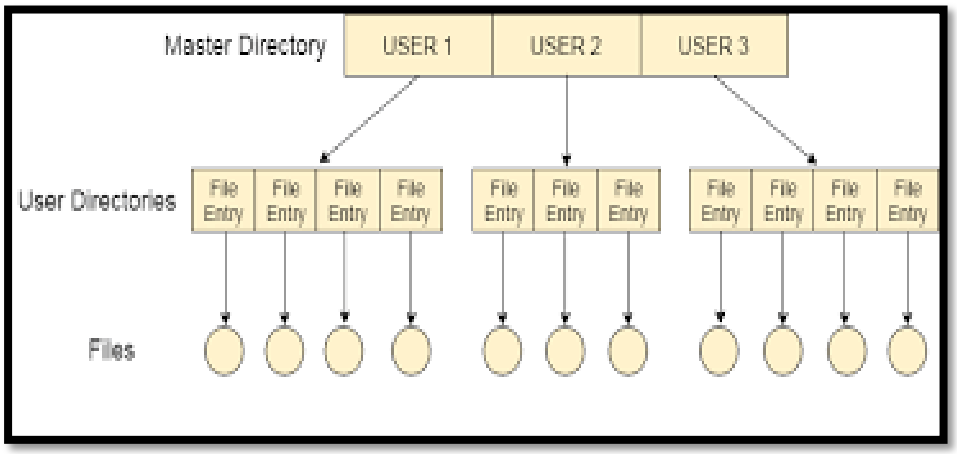
Authentication

- Create a file where all the username and password will be stored.

- For implementing authentication, we have to modify xv6 so that the user must first log in before being presented with the command line.
- To do this, we have to make changes in init.c which is the first program to run.
- Before starting “sh” (short for shell), we will do the authentication.
- Define a user struct –

```
struct user {
    char username[STRING_SIZE];
    char password[STRING_SIZE];
    int uid;
    int gid;
    char homedir[STRING_SIZE];
    struct user* next;
};
```

- “homedir” will be the User directory where the user will be sent after log in.



- Define group struct –

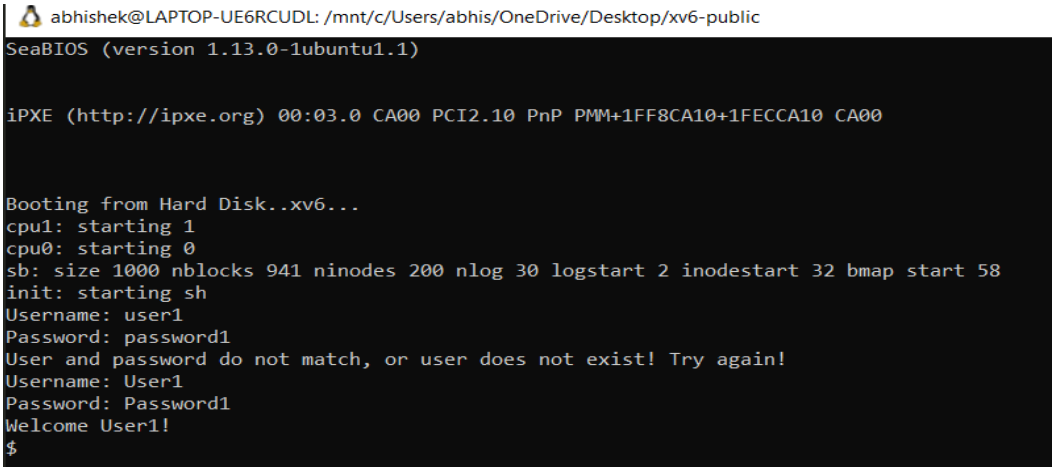
```
struct group {
    char groupname[STRING_SIZE];
```

```
    int gid;
    struct user* userList;
    struct group* next;
};
```

- We will ask from the user, his/her username and password. If username is present and password is correct we would consider the user logged in, otherwise show incorrect credentials message.
- If incorrect credentials were given, we should allow the user to try again.
- Only limited number of wrong attempts should be allowed.
- I have implemented this feature and got the following output.
- Psuedocode –

```
Login(char* user, char* password){
        for(int i = 0; i < registered_users_count; i++)
                if(!strcmp(user, regusers[i] and !strcmp(pass, regpass)))
                        return 1
        return 0;
}
int main(){
//previous code
pid = fork();
if(pid == 0){
        int count = 0;
        while(count < 3){
                char *user = (char *)malloc(BUFFLEN);
                  user = gets(user , 20);
                char *pass = (char *)malloc(BUFFLEN);
                pass = gets(pass , 20);
```

```
            loggedIn = login(user, pass);
                if (loggedIn) {
                  //opens shell for the user
                  printf(1, "Welcome %s!\n", user);
                  exec("sh", argv);
                  printf(1, "init: exec login failed\n");
                  exit();
                }
                else {
                  printf(1, "User and password do not match, or user does
        not exist! Try again!\n");
                  count++;
                }
            }
        }
    }
```



Access Control

- In the standard Linux security model, every process runs "on behalf of" a user. Typically, after logging in as user1, every process run within the logged-in console should run on behalf of user1.

- Add a field uid to struct proc in proc.h to track the user responsible for each process, and update xv6 so that this field contains the correct value.
- The login program should run as user root. We need to introduce a system call setuid() to allow login to change identity from root to the appropriate user after a successful login.
- Only root should use the setuid() system call. Hence, if the user id in struct proc is not 0 (uid of root), setuid() should return an error and not change the user id.
- Psuedocode of setuid –

```
int sys_setuid()
{
    int uid;
    if(argint(0, &uid) < 0)
        return -1;
    struct proc* currProc=myproc();
    if(currProc->uid!=ROOT)
        return -1;
    currProc->uid=uid;
    return 1;
}
```

- Also set uid of the current process to the uid of the user.
- Note that multiple users may be running processes simultaneously, and conceptually also be logged in simultaneously (though we only have a single console at this point). Hence, you can't use globals to track which user is logged in.
- We have to add two field uid and gid in struct inode (in memory copy of an file) and struct dinode (on disk structure of a file) to store the owner of the file.

- Within one digit, the highest bit refers to the read right (r = 4), the middle bit refers to the write right (w = 2), and the lowest bit to the program execution right (x = 1).
- We have to add a new field mode in the inode and dinode structure which is a int that serves as a bitfield permission that describes the users who have rights to that file. This parameter is often called file mode.
- Inode and dinode structure are present in file.h and fs.h respectively.
- In its higher bits we usually store additional special features of the file, in our case only the setuid bit. The file mode is usually illustrated in two ways: in octal form and in drwxrwxrwx form.
- By default mode of a file is set to 0644 (110 100 100). It can be set in create function in sysfile.c which create a file.
- For validation of an access, we have to create three function validate_read, validate_write and validate_execute in sysfile.c. In each of them we would check whether the uid of the process matches with the uid of the file. If it matches, owner (or user) bits of the mode would be checked, else match the gid of the file with gid's of the user. If any gid matches group bits of the mode would be checked, else other's bits would be checked.

To change the permissions (mode) we can add chmod (change mode) system call. It needs path of the file whose mode have to be changed. It have to be called the by the owner or the root user. If current process uid is not that of the owner or root, don't change the mode and return.

- Psuedocode –

int sys_chmod(void){

//path of file and new_mode would given as arguments

currProc = myproc();
inode = namei(path of file); //namei gives inode corresponding to a path

```
if(currProc->uid != inode->uid and currProc->uid != ROOT)
    return -1;
inode->mode = new_mode;
}
```