Security and Protection in xv6

CS 344 Assignment 4

Vignesh Ravichandra Rao 190101109

Contents

- Intro to Security and Protection
- Comparison between features of xv6 and Linux
- Drawing out a strategy
- Pseudocode for implementation

Protection

Internal Threats

- Controls the access to the system resources
- Uses principle of least privilege and is implemented using authorisation/permission policies

Security

External Threats

- Allows only legitimate users to use the system
- Implemented through authentication, encryption, firewalls, etc.

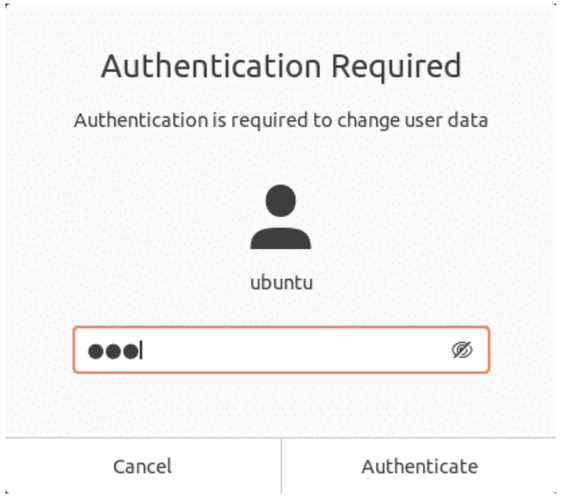
Linux OS

Security and Protection Features

- **User Authentication** - Usually a username password mechanism to identify user.
  - Authentication is done right after booting and before privileged operations.

- **File System Protection** - Defines which users/groups can read, write and execute given files.
  - Ensures that users can’t perform unauthorised operations on files.

- **Firewalls** - Linux uses the Netfilter framework to implement firewalls



xv6

Security and Protection Features

- Protection limited to memory access.
- Uses segmentation to protect parts of a program. Direct access to code and stack segment is not allowed.
- Implements protection of physical memory pages from improper access using PTE_U bit in page table entry.
- No notion of users (not multiuser OS). Hence, no form of security and authentication in place.
- **User Authentication**
- **File System Access Control**

Strategy

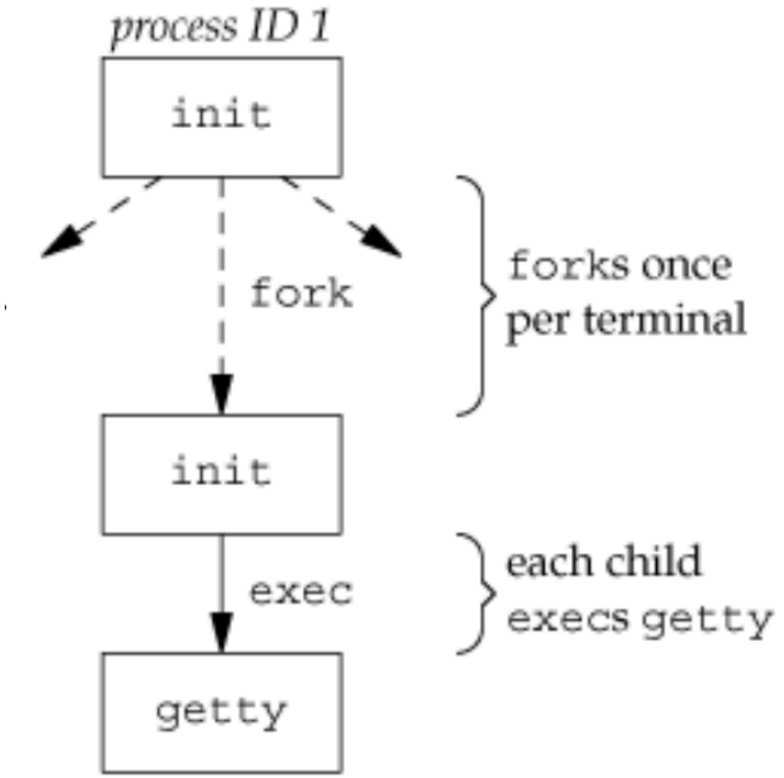
User Authentication

- Create a /passwd file in the root directory to store username, uid, private group and password
  - username = root and password = 123456 (arbitrary)
  - Format: **<username>:<password>:<uid>:<private-gid>**
- New user programs:
  - useradd - Create a new user
    - Format: **useradd <username>**
  - passwd - Change/add a password for a user
    - Format: **passwd <username> <passwd>**

Strategy

User Authentication

- Create getty.c for login prompt.
  - This process should access /passwd to verify the identity of the user.



Strategy

File System Protection

- We will add groups into xv6. Create a similar /group file in root to store group related details.
  - Format: **<group_name>:<gid>:<members>**
- Store access mode in struct inode and uid, gid and euid in struct proc.
- System calls:
  - int setuid(int uid) - allows root to change uid of process
  - int setgid(int gid)
  - int chown(char * path, int uid) -
    Changes owner uid of a file

Strategy

File System Protection

- System calls:
  - int chmod(char* path, int mode) - changes the access permission mode of a file
  - int chgrp (char* path, int gid) - changes group owner of a file
- User programs:
  - groupadd - Adds a group entry in /group file
  - usermod - Adds user into a group
    - Format: **usermod <username> <groupname>**
  - chmod, chgrp, chown - Allows user to call resp. syscalls

Implementation Details

User Authentication- Add /passwd file into xv6 file system

- mkfs.c is responsible for mounting the file system in xv6
- Create a passwd file in the xv6 source code folder
- Within this file, add the root user credentials in the specified format. Note that root user must have a uid = 0
- Update Makefile to reflect this change:

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

passwd

Implementation Details

User Authentication- struct user

- Create struct user to store user details.
- This struct can be used a return type while parsing the /passwd file later on

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

Implementation Details

User Authentication- useradd.c

- User program to add a new user.
- Command: **useradd <username>**

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()
```

Implementation Details

User Authentication- passwd.c

- Updates password of a user
- Command: **passwd <username> <password>**

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()
```

Implementation Details

User Authentication- getty.c

- This is login prompt process that is forked by the init process.

```
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
```

- In init.c, remove exec(“sh”) and replace with **exec(“getty”)**

Implementation Details

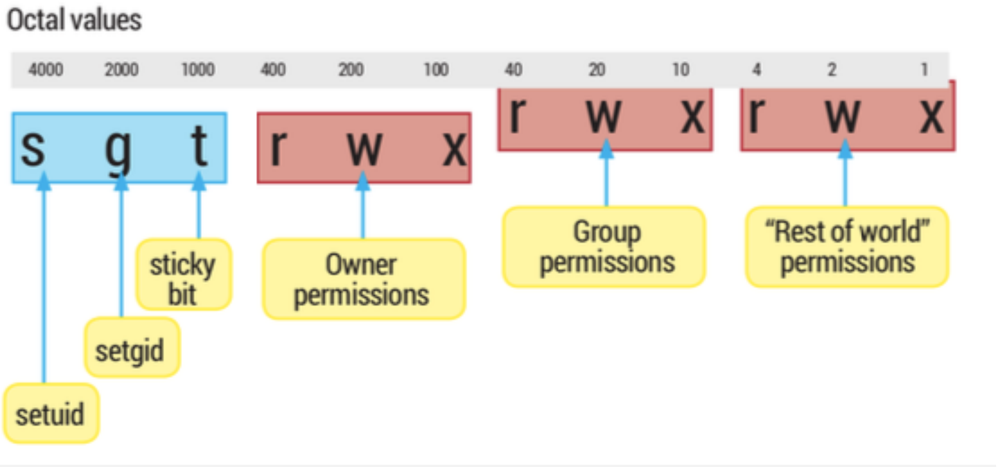
User Authentication- proc.c

- We need to make sure every process stores the logged in user's uid, gid and has an effective uid.
- Effective uid may not be equal to uid if setuid bit is used for a file.
- **allocproc()** : Initialise uid, gid and euid to 0.
- **fork()**: Make child process inherit uid, gid, euid from parent.

Implementation Details

File System Protection - Initial changes

- Create a /group file in a similar manner as /password.
- Add **uid**, **gid** field for ownership in struct inode and field **mode** for permissions
- In **sysfile.c** , update the **create()** function.
  - This function is used to allocate inodes for new files.
  - Make sure that a new inode, borrows the running process' uid and gid



Implementation Details

File System Protection - setuid() / setgid() system call

- System call to change uid/gid of current running process.
- Can be called only by root user.
- Hence, the pseudo code is:

```
sys_setuid()
    newUID = ARGINT(0) // Fetch argument from user stack
    IF myproc()->uid == ROOT_UID
        myproc()->uid = newUID
        RETURN 0
    ENDIF
    RETURN -1
```

- setgid() is almost identical but uses gid instead of uid.

Implementation Details

File System Protection - Enforcing permissions

```
int allowRead(char* path)
    fileInode = nameiparent(path) // Get inode
    ownUID = fileInode->uid
    ownGID = fileInode->gid
    mode = fileInode->mode
    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
```

- Call **allowRead()** in **sys_open()** of **sysfile.c** to ensure that read permission is allowed for current user.
- Similarly, create **allowWrite()** and use it in **sys_write()**.
- Similarly, create **allowExec**() function and check for permission in **exec.c**

Implementation Details

File System Protection - chown(), chgrp() and chmod() system call

- System calls to change ownership
- Implementation details: int chown(char* path, int uid)

  1. Open inode of file at path.

  2. Check if current user is root or owner of file. If not, return -1

  3. Update the uid file of inode

- chgrp() and chmod() implemented in the same fashion.
- Also add user programs, chown.c, chgrp.c and chmod.c
  to allow user to call these from shell process.

Implementation Details

File System Protection - groupadd and usermod

- Similar to useradd user program. Implementation of **groupadd.c** user program:
  - Open /group file. (Will be automatically disallowed if not root)
  - Find minimum unused gid.
  - Create new entry in /group file with given groupname and gid.
- **usermod.c** - user program to add user to a group:
  - Open /group file.
  - Seek to the line having entered group details.
  - In members field of line, add a comma separated entry for new user

Thank you!