

Overview

- Programs we write interact with their environment and for us it is important that such interaction is secure, be it with users directly, with the network, with the file system etc.
- Here we focus on the file system and look at how file permissions work under Unix
- Material drawn from *Advanced Programming in the Unix Environment* by W. Richard Stevens (in the library)
- We also look at set user ID programming under Unix
- Material drawn from *Setuid Programming Demystified* by Sekar et al (linked off the further reading page)



Logging in

- To log in we enter a username and password
- The password entered is hashed, compared with that username's password in the password file and if they match access is granted
- An entry in the password file has the following structure...

```
username:encrypted password:user ID:group ID:
comment field:home directory:default shell
```
- A real world example might look like this...

```
mary:kjh76jkh88:222:333:Mary O'Brien:
/home/castaff/mary:/bin/bash
```



Logging in

- The user ID entry is a numeric value that identifies each user to the system
- Every user has a unique user ID that the kernel uses in deciding whether to grant permission to carry out requested operations
- User ID zero is reserved for `root` or the superuser; `root` has free rein over the system
- Also specified in the password file is a user's group ID
- Multiple users can be placed in the same group in order to facilitate resource sharing (more on this later)
- A user may be a member of supplementary groups



Logging in

- Upon logging in a command shell (e.g. `bash`) launches
- The shell's effective user and group IDs are set to the user ID and group IDs in the password file
- Thereafter child processes inherit effective user and group IDs from their parent (not always true, more on this later) and so each process (and its descendants) created by the shell runs with the user's privileges
- Furthermore, should any new files be created, their user and group IDs are inherited from the effective user and group IDs of the creating process (not always true...)
- What role do these effective user and group IDs play in controlling access to resources?



Unix File Permissions

- Below is an excerpt from a directory listing produced by issuing the `ls -l` command...

```
-rwxr-x--x dobrien castaff runme
```
- First off, this tells us that the program `runme` is owned, and was presumably created by user `dobrien` who is a member of the `castaff` group
- The first character on the left hand side indicates the file type with `-` denoting a regular file and `d` a directory (other file types include devices, sockets, pipes and symbolic links)



Unix File Permissions

- The next nine characters should be taken in sets of three and are, from left to right, the file permissions for:
 - The owner of the file (`dobrien`)
 - Users in the file owner's group (`castaff`)
 - Others (i.e. the world or everyone else)
- What do `r`, `w` and `x` mean, applied to a regular file?
 - `r` grants permission to `open` for reading
 - `w` grants permission to `open` for writing
 - `x` grants permission to `execute` (i.e. to serve as an argument to the `exec*` family of functions)



Unix Directory Permissions

- In Unix, a directory is a file and a directory entry is a filename and a corresponding inode number (where the real information about the file is stored)
- We can use the `stat*` functions to retrieve all available information for a file
- Since a directory is a file it has an owner and a group and permissions can be applied to it, what do `r`, `w` and `x` mean when applied to a directory?
 - `r` grants permission to list directory contents
 - `w` grants permission to create and delete files
 - `x` grants permission to traverse the directory boundary



Unix Directory Permissions

- Thus, permission to delete a file comes not from the file itself but from the directory in which it resides
- If a user has write permission on a directory she can delete any file in it irrespective of that file's owner, group or permission settings
- Without read permission on a directory a user cannot list directory contents
- However, she can open files in that directory provided she knows the filename (and of course has execute permission on that directory)



Unix Directory Permissions

- The permissions on your home directory are typically set to `drwx-----x` giving you full access, excluding your group and allowing others execute permission
- How secure are those settings? Why the `x` for others?
- Whenever we wish to open any file we must have execute permission on each directory in the path and the appropriate permission on the file itself



Access Rules

- The kernel sequentially applies these rules in deciding whether to grant a process access to a resource. . .
 1. If the EUID (effective user ID) of the process is 0 (i.e. the process is running as `root`), grant access
 2. If the EUID of the process matches the owner of the file and the appropriate permission bit is set, grant access; otherwise deny access
 3. If the EGID (effective group ID) of the process or one its supplementary group IDs matches the group ID of the file and the appropriate permission bit is set, grant access; otherwise deny access
 4. If the appropriate other permission bit is set, grant access; otherwise deny access



Ownership of New Files and Directories

- Ownership of a new file is assigned according to the EUID of the creating process
- A new file takes its group ID from the EGID of the creating process
- (Unless the `setgid` bit is set on the parent directory in which case the new file inherits its group ID from the parent directory)
- (More on the `setuid` and `setgid` bits later)



Setting Permissions

- Interpreting `r`, `w`, and `x` as bits leads to a handy octal format for setting file permissions
- Turning on all permissions gives 7 ($2^2 + 2^1 + 2^0 = 7$), turning on only read permission gives 4 ($2^2 + 0 + 0$), read and write permission gives 6 ($2^2 + 2^1 + 0$)
- We can use `chmod` to modify file permissions and using octal format we get e.g.
 - `chmod 644 filename`
 - `chmod 700 filename`
 - `chmod 777 filename`



Setting Permissions

- Some versions of Unix also support Access Control Lists (ACLs) which allow a file's owner to specify access rights for individual users thus getting around the rather coarse owner/group/other distinction
- ACLs can be manipulated in Linux through `getfacl` and `setfacl` but are not part of all Unix distributions so for now we stick to user ID model
- What permissions are applied to newly created files?



Permissions on New Files and Directories

- A file mode creation mask, `umask`, is associated with every process and is inherited from its parent
- It specifies the permissions that should be disabled on any new files (most Unix versions specify an octal file creation mode of `0666` giving everyone read and write permissions)
- Common `umask` values are `0022` and `0027`, respectively denying write permission to both group and others and denying write to group and read/write/execute to others
- We can set it with the `umask` command or system call



The 3 Other Bits

- “But why the leading zero in a `umask` of `0022`?”
- For every file and directory 3 additional bits can be turned on, they are:
 - The sticky bit
 - The set user ID bit
 - The set group ID bit



The Sticky Bit

- When applied to an executable under older Unixes it meant a copy of its text section was kept in swap after the program terminated so it would start faster next time
- We have seen that having write permission on a directory means we can delete any file from it; but what about `/tmp`?
- On a multiuser OS `/tmp` is shared and while we wish to allow file creation we do not want users deleting each other's files
- Setting the sticky bit on a directory means that in order to delete files from it we need to both have write permission on the directory and be the file's owner (set on `/tmp`)



The Set Group ID Bit

- We mentioned earlier that applied to a directory this bit causes new files to inherit group ownership from the parent directory and not from the parent process
- Applied to a non-executable this bit enables mandatory record locking (more about file locking later)
- Applied to an executable it plays a role similar to the set user ID bit (see below)



The Set User ID Bit

- In order to understand the effect of enabling this bit on an executable we need to consider how an ordinary user gets to change her password in Unix. . .



Set User ID Programs

- Encrypted user passwords are stored in `/etc/shadow`
- Although encrypted, this file should not be world readable since password crackers could be run against it to implement a dictionary attack
- Therefore `/etc/shadow` is readable only by `root`
- Yet an ordinary user can update her password through invoking the `passwd` command
- If the `passwd` process inherited its privileges (i.e. IDs) from the shell (like other processes) it would not be able to update the password file



Set User ID Programs

- So how does `passwd` get its special powers?
- The `passwd` utility is a `setuid root` program
- Unix systems offer a set of system calls, the UID-setting system calls, which allow a process to raise and drop privileges
- “Unfortunately these calls are poorly designed, widely misunderstood and insufficiently documented”
- Poor documentation is a serious issue since misuse of UID-setting system calls may result in a process not properly dropping privileges



Set User ID Programs

- Should such a process be compromised an attacker might succeed in executing malicious code with elevated privileges
- It is therefore important we understand both how to correctly use UID-setting system calls and the differences in API semantics across Linux, Solaris and FreeBSD



User ID Model

- Each Unix system user has a unique user ID
- Each process has three user IDs: the real user ID, the effective user ID and the saved user ID
- The real user ID identifies the process owner
- The effective user ID defines access rights
- The saved user ID stores a user ID for later restoration
- A process also has three group IDs: the real group ID, the effective group ID and the saved group ID and their role is similar to that of user IDs



User ID Model

- When a process is created by `fork` it inherits its three user IDs from its parent
- When a file is executed through `exec` it keeps its three user IDs unless the `setuid` bit is set on the executable
- If this bit is set then the process's effective and saved user IDs are set to the user ID of the owner of the file
- Thus a `setuid` program owned by `root` will run with `root` privileges
- Looking at the `passwd` program we see...

```
-rwsr-xr-x root shadow passwd
```



User ID Model

- Access control is based on effective user ID so a process gains privilege by assigning a privileged user ID to its effective user ID and drops privilege by removing the privileged user ID from the effective user ID
- Privileges may be dropped temporarily or permanently
- Temporarily dropping privileges involves a process removing the privileged user ID from the effective user ID and storing it in the saved user ID so it may be restored
- Permanently dropping privileges involves a process removing the privileged user ID from real, effective and saved user IDs



User ID-Setting System Calls

- They are `setuid`, `seteuid`, `setreuid` and on some systems `setresuid`
- Of them all, `setresuid` has the clearest semantics and can be used to individually set real, effective and saved user IDs
- It is not however available on all Unix flavours (e.g. Solaris) so we look at how the others can be used
- Clear too is `seteuid` which sets the effective user ID only, leaving real and saved user IDs unchanged



User ID-Setting System Calls

- Confusing is `setreuid`, it modifies the real and effective user IDs and in some cases also the saved user ID, the rules are a little confusing!
- Here is an excerpt from the `setreuid` man page... (huh?)

```
If the real user ID is set or the effective user ID is set to a value not equal to the previous real user ID, the saved user ID will be set to the new effective user ID
```



User ID-Setting System Calls

- Also confusing is `setuid`, it sets the effective user ID but if the effective user ID is zero then real and saved user IDs are also set, if the effective user ID is not zero then it alone is set
- Thus a `setuid` program can sometimes use `setuid(newuid)` to permanently drop privileges since real and saved user IDs are set, but why only sometimes?
- A `setuid somebody` program (where `somebody` is some privileged user other than `root`) can never use `setuid(newuid)` to implement a permanent drop in privilege



User ID-Setting System Calls

- In general if the effective user ID is zero we can set user IDs to any value otherwise we can only move between user IDs
- And this is all only on Linux! Semantics change across Solaris and FreeBSD



General Guidelines

- Since `setresuid` has the clearest semantics it should always be used where available
- Otherwise use `seteuid` to set effective user ID and `setreuid` to set all three user IDs
- Avoid `setuid` and its overloaded semantics
- Drop group before user privileges or you may never be able to drop group privileges

General Guidelines

- Semantics vary and may change with new kernels so always check return values: zero means success (make sure you know what “success” means)
- If available use `getresuid` to verify user IDs are as you would expect after any modifications
- Be paranoid: verify raising privileges is no longer possible after dropping them permanently
- Use elevated privileges early on and drop them permanently as soon as they are no longer required