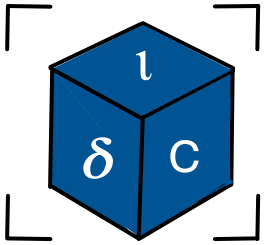
Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi



Real User ID and Read Group ID

- The real user ID and group ID identify the user and group to which the process belongs.
  - A login shell gets its real user and group IDs from the third and fourth fields of the user's password record in the /etc/passwd file
- When a new process is created (e.g., when the shell executes a program), it inherits these identifiers from its parent.

Effective User ID and Effective Group ID

- Used to determine the privilege granted to a process when it tries to perform various operations
  - Privilege granted to a process when it accesses resources such as files and System V interprocess communication (IPC) objects, which themselves have associated user and group IDs determining to whom they belong.
  - The effective user ID is also used by the kernel to determine whether one process can send a signal to another.
- Process with effective userid = 0 has all the privileges of a superuser.
  - Certain system calls can be executed only by privileged processes.

Changing EID and EGID

- Normally, the effective user and group IDs have the same values as the corresponding real IDs.
- Two ways in which the effective IDs can assume different values.
  - use of system calls
  - execution of set-user-ID and set-group-ID programs

Effective User ID and Effective Group ID

- When a process executes a file by execve it keeps its 3 userids unless the set user id bit on the file is set in which case:
  - If the file executed is a set-userID file, the effective and saved user IDs of the process are set to the owner of the file executed.
  - If the file executed is a set-group-ID file, the effective and saved group IDs of the process are set to the group of the file executed.
- If the file executed is not a set-user-ID or set-group-ID file, the effective user ID, saved user ID, effective group ID, and saved group ID are not changed.

Saved set user ID and Saved set group-ID

If the set-user-ID (set-group-ID) permission bit is enabled on the executable, then

- the effective user (group) ID of the process is made the same as the owner of the executable.

If the set-user-ID (set-group-ID) bit is not set, then

- no change is made to the effective user (group) ID of the process.
- The values for the saved set-user-ID and saved set-group-ID are copied from the corresponding effective IDs. This copying occurs regardless of whether the set-user-ID or set-group-ID bit is set on the file being executed.

Example

- **sudo -i**
  Password:

- **ls -l prog**
- **chmod u+s prog**
- **chmod g+s prog**
- **ls -l prog**

Example 1

- Suppose that a process whose real user ID, effective user ID, and saved set-user-ID are all 1000 execs a set-user-ID program owned by root (user ID 0). After the exec, the user IDs of the process will be changed as?

Modifying

when executing a set-user-ID program

Table 9-1: Summary of interfaces used to change process credentials

Interface	Purpose and effect within:	
	unprivileged process	**privileged process**
setuid(u) *setgid(g)*	Change effective ID to the same value as current real or saved set ID	Change real, effective, and saved set IDs to any (single) value
seteuid(e) *setegid(e)*	Change effective ID to the same value as current real or saved set ID	Change effective ID to any value
setreuid(r, e) *setregid(r, e)*	(Independently) change real ID to same value as current real or effective ID, and effective ID to same value as current real, effective, or saved set ID	(Independently) change real and effective IDs to any values
setresuid(r, e, s) *setresgid(r, e, s)*	(Independently) change real, effective, and saved set IDs to same value as current real, effective, or saved set ID	(Independently) change real, effective, and saved set IDs to any values

One-way trip!
once a privileged process has changed its identifiers in this way, it loses all privileges and therefore can't subsequently use *setuid()* to reset the identifiers back to 0.

Example 2

Setuid-ex.c

Principle of least privilege

- A process may only reduce its privileges.
- A process may not gain any privileges, with one exception: a process that execs a program from a file that has a setuid or setgid flag set gains the privileges expressed by this flag.

- Helps reduce the "attack surface" of the computer by eliminating unnecessary privileges that can result in network exploits and computer compromises.

Hold privileges only when required

- In a set-user-ID program

```
uid_t orig_euid;

orig_euid = geteuid();
if (seteuid(getuid()) == -1)            /* Drop privileges */
    errExit("seteuid");

/* Do unprivileged work */

if (seteuid(orig_euid) == -1)           /* Reacquire privileges */
    errExit("seteuid");

/* Do privileged work */
```

- The first call makes the effective user ID of the calling process the same as its real ID.
- The second call restores the effective user ID to the value held in the saved set- user-ID.

Drop them when not required

- If a set-user-ID or set-group-ID program finishes all tasks that require privileges, then it should drop its privileges permanently in order to eliminate any security risk that could occur because the program is compromised by a bug or other unexpected behavior.

General points on changing process credentials

- Drop privileges permanently before execing another program
- Avoid executing a shell (or other interpreter) with privileges
- Close all unnecessary file descriptors before an *exec()*

Difference between su and sudo

- su: "become" root (su = superuser)
- su - username: "become" username, using initialization files
- **sudo command: Execute command as root (if youre in /etc/sudoers and you give your password.)**

Chroot syscall

```
#include <unistd.h>
int chroot(const char *pathname);
```

- Returns 0 on success, or –1 on error

- chroot changes apparent root directory for current running process and its children.
- System call defined in uninstd.h

Chroot command

- `chroot <root for process> <command to run>`

- Launch a process but can tell it another directory to adopt and record as its root.

chroot

- Each PCB stores the root directory of a process.
  - Just the name of some directory in the filesystem.
  - The process's root directory, unlike the filesystem's, is virtual rather than physical.
- Most processes' root directories are /, the filesystem's root.
  - The process has visibility over the entire filesystem.
- When the operating system interprets filenames that appear in the process's code, it does so relative to whatever physical directory the process names as its root.
- Example: root: /home/joe

  fopen(/etc/passwd, rw)
  - Technically opening /home/joe/etc/passwd

chroot

- The effect is to blind the process to any part of the physical filesystem except the part under the process's root.
- **New processes inherit their parent process's root directory**, any spawned processes will be similarly blind. The security implication is protection of most of the filesystem from the code in the process.

Why?

- For programs like ftp.
  - As a security measure, when a user logs in anonymously under FTP, the ftp program uses chroot() to set the root directory for the new process to the directory specifically reserved for anonymous logins.
  - After the chroot() call, the user is limited to the file-system subtree under their new root directory, so they can’t roam around the entire file system.

Example

- Running ls vs running date

```
# mkdir /tmp/example
# cp /bin/ls /tmp/example/ls
# chroot /tmp/example /ls
        chroot: failed to run command ‘ / l s ’: No such file or directory
# cp −r /lib64 /tmp/example/lib64
# mkdir −p /tmp/example/lib
# cp −r /lib/x86_64−linux−gnu /tmp/example/lib/x86_64−linux−gnu
# chroot /tmp/example /ls
        /ls: error while loading shared libraries: libpcre2−8.so.0: cannot o
# cp /usr/lib/x86_64−linux−gnu/libpcre2−8* /tmp/example/lib/x86_64−l
# chroot /tmp/example /ls / lib lib64 ls
# chroot /tmp/example /ls /.. lib lib64 ls
```

Chroot is a jail

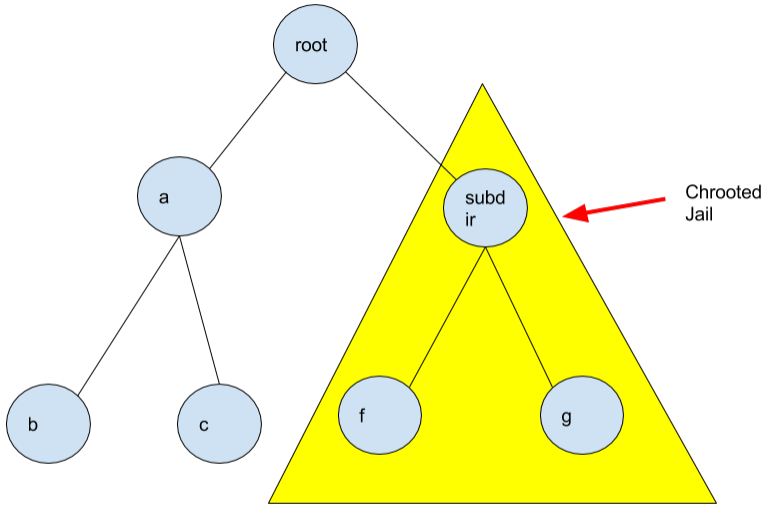
- Once you are inside a jail, you cannot see any file outside of the jail.
- Therefore, you need to copy a number of commands and libraries into the jail first; otherwise, there is not much you can do.
- It is not sufficient to just copy commands; their dependencies must also be copied.
  - most programs are dynamically linked against shared libraries. Therefore, we must either limit ourselves to executing statically linked programs, or replicate a standard set of system directories containing shared libraries (including, for example, /lib and /usr/lib) within the jail

Example

- Running ls vs running date

Jails can be broken: Links and chroot

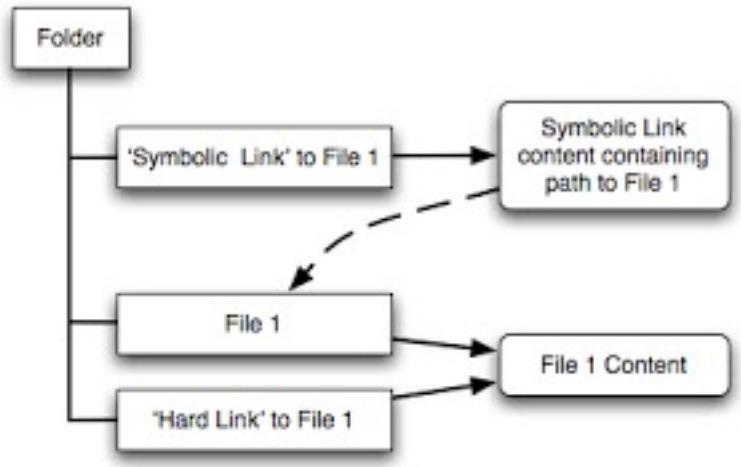
- Symbolic links to directories outside the jail can’t be reached
- Hard links that reaches outside the jail directory tree compromises the jail.



$ln /usr/bin/emacs subdir/bin/emacs
$ln /dev/tty subdir/dev/tty
$ln /dev/disk/00 subdir/dev/disk/00

Access as root and then go to other parts of root filesystem

Sympolic Vs Hard Links



Symbolic vs Hard Links

- Symbolic links:
  - has only the path of the original file, not the contents
  - permissions of symlinks don’t matter, permissions cascade to the source
  - can cross the file system
  - allows you to link between directories
- Hard Link
  - has the same inodes number and permissions of original file
  - permissions will be updated if we change the permissions of source file
  - has the actual contents of original file, so that you still can view the contents, even if the original file moved or removed.
  - can’t cross the file system boundaries
  - **can’t link directories**

Why hardlinks?

- rm command
  - ls –li | grep <inode>

- $ ls -id .
  1069765 ./
- $ mkdir tmp ; cd tmp
- $ ls -id ..
  1069765 ../

- On Unix filesystems .. is a real directory entry; it is a hard link pointing back to the previous directory.
- no size or speed penalty

Other ways to break the jail-1

- Calling chroot() doesn’t change the process’s current working directory. Thus, a call to chroot() is typically preceded or followed by a call to chdir() (e.g., chdir(“/”) after the chroot() call).
- If this is not done, then a process can use relative pathnames to access files and directories outside the jail.

Other ways to break the jail-2

Open File descriptors: If a process holds an open file descriptor for a directory outside the jail, then the combination of fchdir() plus chroot() can be used to break out of the jail, as shown in the following code sample:

```
int fd;
fd = open("/", O_RDONLY);
chroot("/home/mtk");                    /* Jailed */
fchdir(fd);
chroot(".");                          /* Out of jail */
```

To prevent this possibility, we must close all open file descriptors referring to directories outside the jail.

Other ways to break a jail-3

- The jailed process can still use a UNIX domain socket to receive a file descriptor (from another process) referring to a directory outside the jail. By specifying this file descriptor in a call to fchdir(), the program can set its current working directory outside the jail and then access arbitrary files and directories using relative pathnames.

Chroot impracticaliity

- some things make chroot impractical in general: seems like one needs extra copies of most of the system hard to communicate between separate roots requires administrator permissions to configure dangerous to let normal users configure b/c they could confuse priviliged (set-user-ID) programs like sudo

Example

- What scenarios does chroot make most/least sense for?

A. the rendering part of web browser

B. a web server

C. a media player

D. a network time server (for other machines to set their clocks)

Capabilities: Problems with Privileges

- Previleges: a binary system of privileged and non-privileged processes
- Either your process could do everything—make admin-level kernel calls—or is restricted to the subset of a standard user
- Capabilities: more nuanced
- Only needed for system-level tasks.
- Come into action during the execution of the process.
- Most of the time under the hood

Capabilities

- Effective capabilities (CapEff): capabilities that will be verified for each privilege action (If the process/thread wants to perform the action, the capability needs to be in this set while doing it)
- Permitted capabilities (CapPrm): capabilities that can be introduced into effective when needed using syscalls; once dropped never acquired
- **Inherited capabilities (CapInh)**
- **Ambient capabilities set (CapAmb)**
- Bounding set (CapBnd): capabilities superset, nothing more than this can be done

Checking capabilities

$ capsh —print

$ grep Cap /proc/$BASHPID/status

$ capsh —decode=0000000000000000

$ sudo –l

$ sudo capsh --print

The case of privileged command

- Consider a privileged command that all users should be able to run
- Currently, there are the following ways to achieve this:
  - Add all users to sudo group
  - Add entries for all users in sudoers file (to allow users to run ping command)
  - Set setuid bit on ping binary
  - Set specific capability (CAP_NET_RAW) on ping binary

CAP_NET_RAW capability enables a process to

- use RAW and PACKET sockets;
- bind to any address for transparent proxying.