CS631 - Advanced Programming in the UNIX Environment
–
Restricting Processes

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
`jschauma@stevens.edu`
`http://stevens.netmeister.org/631/`

Restricting Processes

The nature of UNIX being a multitasking multiuser OS implies the need for:

- user privileges
- file permissions
- process ownership
- management of all finite resources

That is, we have a constant need to *restrict* processes, to *control* process groups, and to *contain* applications.

What we know so far...

- Resource Limitations (Lecture 02 / Lecture 06), *e.g.* use of `getrlimit(2)`/`sysconf(2)` in `openmax.c`
  - per-process or per-user limits
  - system-wide hard-coded limits
  - system tunable configuration options
- UNIX Access Semantics based on File Ownership (Lecture 03)

Filesystem access

Recall from lecture 03 how access semantics are applied, in order:

1. If effective-uid == 0, grant access
2. If effective-uid == `st_uid`
   - 2.1. if appropriate user permission bit is set, grant access
   - 2.2. else, deny access
3. If effective-gid == `st_gid`
   - 3.1. if appropriate group permission bit is set, grant access
   - 3.2. else, deny access
4. If appropriate other permission bit is set, grant access, else deny access

Filesystem access

Limitations of the traditional Unix access semantics:

- a file can only have one group owner
- group membership quickly becomes convoluted
- different (file- and operating-) systems have different limits on the number of groups a user can be a member of
- any modification of group membership requires the sysadmin to make changes (add/remove members, create new groups, ...)

Access Control Lists

POSIX.1e Access Control Lists (ACLs) provide more fine-grained access control:

- user can specify individuals or groups with different access
- implemented as 'Extended Attributes' in the filesystem
- `ls(1)` indicates their presence via a '+' at the end of the permissions string

Access Control Lists

Example: `linux-lab.cs.stevens.edu`

```
$ whoami
jschauma
$ groups
professor abcxyz null nova one threedot sigsegv flag
$ ls -l hole.c
-rw------- 1 jschauma professor 984 Sep 10 19:50 hole.c
$ getfacl hole.c
# file: hole.c
# owner: jschauma
# group: professor
user::rw-
group::---
other::---
$ setfacl -m g:student:r hole.c
setfacl: hole.c: Operation not supported
```

Access Control Lists

Example: `linux-lab.cs.stevens.edu`

```
$ ls -l hole.c
-rw------- 1 jschauma professor 984 Nov 24 21:51 hole.c
$ setfacl -m g:student:r hole.c
-rw-r-----+ 1 jschauma professor 984 Nov 24 22:07 hole.c
$ ls -l hole.c
$ getfacl hole.c
# file: hole.c
# owner: jschauma
# group: professor
user::rw-
group::---
group:student:r--
mask::r--
other::---
$
```

Changing eUIDs

ACLs control access to files and directories by eUID/eGID. Recall from Lecture 03 that we can *change* those: `setuid.c`

Common examples:

- necessary access to privileged resources (*e.g.*, binding to a port < 1024)
- handling logins (*e.g.*, `login(1)`, `sshd(8)`)
- raising privileges (*e.g.*, `su(1)`, `sudo(8)`)

Changing eUIDs

Pitfalls:

- `setuid programs`
  - require careful raising and lowering privileges *only when needed* (Least Privilege)
  - rely on corect ownership and permissions (*i.e.*, factors outside of the control of the program)
- `su(1)`
  - requires sharing of a password
  - grants all or nothing access
- `sudo(8)`
  - often misconfigured granting too broad access (`ALL:ALL`)
  - additional authentication often dropped (`NOPASSWD`)
  - restrictions often overlook privilege escalations

Restricting eUID

Your eUID controls access to resources. But we can restrict certain access further via *e.g.* “file flags” (BSD: `chflags(1)`/`chflags(2)`) or “file atributes” (Linux: `chattr(1)`, `lsattr(1)`):

```
$ echo foo > append-only
$ chflags uappend append-only
$ echo bar > append-only
ksh: cannot create append-only: Operation not
permitted
$ echo bar >>append-only
$ cat append-only
foo
bar
$ ls -lo append-only
-rw-r--r--  1 jschauma  wheel  uappnd  8 Nov 28 01:04 append-only
$
```

Restricting eUID 0

You can even prevent eUID 0 from making changes:

```
$ echo "you can't touch this" >hammertime
$ su root -c "chflags schg hammertime"
$ touch hammertime
touch: hammertime: Operation not permitted
$ echo stop >>hammertime
ksh: cannot create hammertime: Operation not permitted
$ rm -f hammertime
rm: hammertime: Operation not permitted
$ su root -c "rm -f hammertime"
rm: hammertime: Operation not permitted
$ ls -lo hammertime
-rw-r--r--  1 jschauma  wheel  schg   21 Nov 28 01:05 hammertime
$
```

But, of course:

```
$ su root -c "chflags noschg hammertime; rm hammertime"
```

Restricting eUID 0

Some restrictions can be enforced on a per-filesystem level, *e.g.*, `noexec` or `nosuid`:

```
$ pwd
/mnt
$ ./a.out
Hello World!
$ su root -c "mount -u -o noexec /mnt"
$ ./a.out
-sh: ./a.out: permission denied
$ ls -l a.out
-rwxr-xr-x  1 root  wheel  8616 Nov 25 16:37 a.out
$
```

See `mount(8)` for a list of supported mount options.

Restricting eUID 0

To prevent even eUID 0 from *e.g.* changing the mount flags, you can employ *securelevels*:

- superuser can *raise* the securelevel
- *lowering* requires reboot
- four securelevels are defined; see `secmodel_securelevel(9)`

```
$ sysctl security.models.securelevel.securelevel
security.models.securelevel.securelevel = -1
$ su root -c "sysctl -w security.models.securelevel.securelevel=2"
security.models.securelevel.securelevel: -1 -> 2
$ su root -c "mount -u -o exec /mnt"
mount_ffs: /dev/wd1 on /mnt: Operation not permitted
$
```

Restricted Shells

Another way of restricting what a user can do is to only allow them to execute specific commands, for example via a *restricted* shell:

- prohibit `cd`
- prohibit changing *e.g.*, `PATH` etc.
- prohibit use of commands containing a `'/'` (*i.e.*, only commands found in the (fixed) `PATH` can be executed)
- redirecting output into files

Beware trivial break-outs via commands that allow invoking other commands!

Restricted Shells

```
$ ksh -r
restricted$ cd /
ksh: cd: restricted shell - can't cd
restricted$ echo foo >/tmp/out
ksh: /tmp/out: restricted
restricted$ /bin/csh
ksh: /bin/csh: restricted
```

But:

```
restricted$ csh
% cd /
% pwd
/
% echo foo >/tmp/out
% exit
restricted$ cat /tmp/out
foo
```

Restricted Shells

To properly restrict a user in this way:

- create a new directory, *e.g.* `/usr/local/rbin`
- carefully reviewed executables needed, then link them in there
- ensure those commands cannot shell out themselves
- set `PATH=/usr/local/rbin`
- mark user config files immutable via `chflags(1)`
- hope you didn't miss anything

Chroot

Expose a restricted copy or view of the filesystem to a process via `chroot(2)` / `chroot(8)`:

- restrict a process's view of the filesystem hierarchy
- restrict commands by only providing needed executables
- must provide full evironment, shared libraries, config files, etc.
- combine with null mounts / mount options
- open file descriptors may be brought into the chroot
- processes outside the chroot are visible!

Try breaking out of a chroot via `break-chroot.c`.

Chroot

```
$ sh mkchroot
$ su root -c "chroot /tmp/chroot /bin/sh"
# pwd
/
# ls
ls: not found
# echo *
bin lib libexec usr
# echo bin/*
bin/id bin/ps bin/sh
# id
uid=0 gid=0 groups=0,2,3,4,5,20,31
# cd /usr/bin
# pwd
/usr/bin
# echo *
*
```

Chroot

Note: inside of the chroot, you can still see the processes from the outside:

```
# ps
 PID TTY   STAT    TIME COMMAND
1296 pts/0 S    0:00.00 /bin/sh
1340 pts/0 S    0:00.01 sh -c chroot /tmp/chroot /bin/sh
1941 pts/0 O+   0:00.00 ps
 760 ?     Is+  0:00.00 /usr/libexec/getty Pc console
 558 ?     Is+  0:00.00 /usr/libexec/getty Pc ttyE1
 772 ?     Is+  0:00.00 /usr/libexec/getty Pc ttyE2
 739 ?     Is+  0:00.00 /usr/libexec/getty Pc ttyE3
# exit
```

Jails

FreeBSD added the `jail(2)` system call and `jail(8)` utility around 2000. Jails...

- enforce a per-jail process view
- prohibit changing sysctls or securelevels
- prohibit mounting and unmounting filesystems
- can be bound to a specific network address
- prohibit modifying the network configuration
- disable raw sockets

Jails effectively implement a process sandbox environment, forming the first OS-level virtualization.

Sun's ZFS capabilities combined with Jail concepts then lead to Solaris Containers and Solaris Zones.

Process Priorities

All processes (including those in a jail) compete for the same resource: CPU cycles, memory etc. Recall Lecture 06 / `getrlimit(2)` / `setrlimit(2)`

```
$ ulimit -a
time(cpu-seconds)    unlimited
file(blocks)         unlimited
coredump(blocks)     unlimited
data(kbytes)         262144
stack(kbytes)        4096
lockedmem(kbytes)    2026214
memory(kbytes)       6078644
nofiles(descriptors) 128
processes            160
threads              160
vmemory(kbytes)      unlimited
sbsize(bytes)        unlimited
```

Process Priorities

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

Returns: setpriority: 0 if OK, -1 on error

- default priority is 0
- *which* is one of `PRIO_PROCESS`, `PRIO_PGRP`, `PRIO_USER`; *who* is PID, PGID, or UID.
- *prio* is a value $-20 <= prio <= 20$
- only the superuser may lower values
- `getpriority(2)` may return -1; need to inspect `errno`

See also: `nice(1)`, `renice(8)`

Process Priorities

```
1$ dd if=/dev/urandom of=/dev/null bs=1024 count=2m
1$ ps -l | grep d[d]
1000 1512 2168 32813   7 20  9828  996 -      RN+ pts/1 0:22.03 dd if=/dev/uran

2$ dd if=/dev/urandom of=/dev/null bs=1024 count=2m

3$ renice 20 1512
1512: old priority 0, new priority 20
3$ ps -l | grep d[d]
1000 1829  871 28252  30  0  9832 1000 -     R+ pts/0 0:06.60 dd if=/dev/uran
1000 1512 2168  5171  21 20  9828  996 -     RN+ pts/1 0:08.91 dd if=/dev/uran
3$
```

See also: `priority.c`

Processor Affinity

In multi-processor systems, you may want to pin a process (group) to a certain CPU (subset). This is known as *Processor Affinity*, *CPU pinning*, or *cpusets*.

- useful to *e.g.* reserve a CPU for core system functionality
- not standardized, so incompatible implementations, APIs, and tools
  - NetBSD: `affinity(3)`, `cpuctl(8)`, `pset(3)`, `psrset(8)`, `schedctl(8)`
  - FreeBSD: `cpuset(1)`, `cpuset(2)`, `pthread_getaffinity_np(3)`
  - Linux: `taskset(1)`, `cpuset(7)`, `sched(7)`

Processor Affinity

Linux `/dev/cpuset` pseudo-filesystem interface:

```
$ mkdir /dev/cpuset
$ mount -t cpuset cpuset /dev/cpuset
$ cd /dev/cpuset
$ mkdir Charlie
$ cd Charlie
$ /bin/echo 2-3 > cpuset.cpus          # CPUs 2 and 3 only
$ /bin/echo 1 > cpuset.mems            # Memory node 1 only
$ /bin/echo $$ > tasks                 # Attach current shell
$ cat /proc/self/cpuset                # Verify the shell is now in the cpuset 'Charl
/Charlie
```

POSIX Capabilities

With so many things to try to restrict, one approach to more fine grained control are so-called *Capabilities*:

- `CAP_CHOWN` - the ability to chown files
- `CAP_SETUID` - allow setuid
- `CAP_LINUX_IMMUTABLE` - allow append-only or immutable flags
- `CAP_NET_BIND_SERVICE`- allow network sockets ¡1024
- `CAP_NET_ADMIN`- allow interface configuration, routing table manipulation, ...
- `CAP_NET_RAW` - raw packets
- `CAP_SYS_ADMIN`- broad sysadmin privs (mounting file systems, setting hostname, handling swap, ...)
- ...

Note the difference in implementation (again); *e.g.* POSIX, FreeBSD `capsicum(4)`, Linux `capabilities(7)`.

Control Groups and Namespaces

Originally termed *process containers*, `cgroups` allow:

- `blkio` - block device I/O
- `cpu` - ability to schedule tasks
- `cpuacct` - CPU usage accounting
- `cpuset` - CPUs and memory nodes
- `devices` - ability of tasks to create or use device nodes
- `freezer` - activity of control groups. Tasks in frozen groups would not be scheduled
- `hugetlb` - large Page support (HugeTLB) usage memory - memory, kernel memory, swap memory
- `net_cls` - ability to tag packets based on control group. These tags can be used by a traffic controller to assign priorities
- `net_prio` - ability to set network traffic priority
- `perf_event` - ability to monitor threads

Control Groups and Namespaces

`cgroups` are implemented as a virtual file system, often using the `/sys/fs/cgroup` mountpoint.

```
# create a new memory cgroup:
mkdir /sys/fs/cgroup/memory/group0
# move the current shell into the memory controller group:
echo $$ > /sys/fs/cgroup/memory/group0/tasks
# limit the shell's memory usage:
echo 40M > /sys/fs/cgroup/memory/group0/memory.limit_in_bytes
#
```

Containers

A *container* is an isolated execution environment providing a form of lightweight virtualization:

- use null and union mounts to provide the right environment
- restrict processes in their utilization
- restrict filesystem views
- restrict processes from what they can see
- restrict processes from what they can do

That is, the basis of many container technologies, such as CoreOS, LXC, or Docker, are `cgroups`, *namespaces*, and the application of all the various concepts discussed above.

Reading

`https://www.netmeister.org/blog/restricting-processes.html`