Linux Capabilities and Namespaces

Capabilities

Michael Kerrisk, man7.org © 2023

February 2023

mtk@man7.org

Outline

Rev: # e2bf8f005a44

Outline

Rationale for capabilities

- Traditional UNIX privilege model divides users into two groups:
  - Normal users, subject to privilege checking based on UID and GIDs
  - Effective UID 0 (superuser) bypasses many of those checks
- Coarse granularity is a problem:
  - E.g., to give a process power to change system time, we must also give it power to bypass file permission checks
    - ⇒ No limit on possible damage if program is compromised

[TLPI §39.1]

Rationale for capabilities

- Capabilities divide power of superuser into small pieces
  - 41 capabilities, as at Linux 6.2
  - Traditional superuser == process that has full set of capabilities
- Goal: replace set-UID-*root* programs with programs that have capabilities
  - Set-UID-*root* binary compromised ⇒ very dangerous
  - Compromise in binary with file capabilities ⇒ less dangerous
- Capabilities are not specified by POSIX
  - A 1990s standardization effort was ultimately abandoned
  - Some other implementations have something similar
    - E.g., Solaris, FreeBSD

A selection of Linux capabilities

Capability	Permits process to
`CAP_CHOWN`	Make arbitrary changes to file UIDs and GIDs
`CAP_DAC_OVERRIDE`	Bypass file RWX permission checks
`CAP_DAC_READ_SEARCH`	Bypass file R and directory X permission checks
`CAP_IPC_LOCK`	Lock memory
`CAP_FOWNER`	*chmod()*, *utime()*, set ACLs on arbitrary files
`CAP_KILL`	Send signals to arbitrary processes
`CAP_NET_ADMIN`	Various network-related operations
`CAP_SETFCAP`	Set file capabilities
`CAP_SETGID`	Make arbitrary changes to process's (own) GIDs
`CAP_SETPCAP`	Make changes to process's (own) capabilities
`CAP_SETUID`	Make arbitrary changes to process's (own) UIDs
`CAP_SYS_ADMIN`	Perform a wide range of system admin tasks
`CAP_SYS_BOOT`	Reboot the system
`CAP_SYS_NICE`	Change process priority and scheduling policy
`CAP_SYS_MODULE`	Load and unload kernel modules
`CAP_SYS_RESOURCE`	Raise process resource limits, override some limits
`CAP_SYS_TIME`	Modify the system clock

More details: *capabilities(7)* man page and TLPI §39.2

Supporting capabilities

- To support implementation of capabilities, the kernel must:
  1. **Check process capabilities** for each privileged operation
     - Cf. traditional check: is process's effective UID 0?
  2. Provide **system calls** allowing a process to modify its capabilities
     - So process can *raise* (add) and *lower* (remove) capabilities
     - (Capabilities analog of *set*id()* calls)
  3. Support **attaching capabilities to executable files**
     - When file is executed, process gains attached capabilities
     - (Capabilities analog of set-UID-*root* program)
- Implemented as follows:
  - Support for first two pieces available since Linux 2.2 (1999)
  - Support for file capabilities added in Linux 2.6.24 (2008)
    - (Delay due to design concerns rather than technical reasons)

[TLPI §39.4]

Outline

Process and file capabilities

- Processes and (binary) files can each have capabilities
- **Process capabilities** define power of process to do privileged operations
  - Traditional superuser == process that has **all** capabilities
- **File capabilities** are a mechanism to give a process capabilities when it execs the file
  - Stored in `security.capability` extended attribute
    - (File metadata)

[TLPI §39.3]

Process and file capability sets

- Capability set: bit mask representing a group of capabilities
- Each **process**[†] has 3[‡] capability sets:
  - Permitted
  - Effective
  - Inheritable

[†] In truth, capabilities are a per-thread attribute
[‡] In truth, there are more capability sets

- An **executable file** may have 3 associated capability sets:
  - Permitted
  - Effective
  - Inheritable
- ⚠ Inheritable capabilities are little used; can mostly ignore

Viewing process capabilities

- `/proc/PID/status` fields (hexadecimal bit masks):

```
$ cat /proc/4091/status
...
CapInh: 0000000000000000
CapPrm: 0000000000200020
CapEff: 0000000000000000
```

  - See `<sys/capability.h>` for capability bit numbers
    - Here: `CAP_KILL` (bit 5), `CAP_SYS_ADMIN` (bit 21)
- *getpcaps(1)* (part of *libcap* package):

```
$ getpcaps 4091
Capabilities for `4091': = cap_kill,cap_sys_admin+p
```

  - More readable notation, but a little tricky to interpret
  - Here, single '=' means inheritable + effective sets are empty
- *capsh(1)* can be used to decode hex masks:

```
$ capsh --decode=200020
0x0000000000200020=cap_kill,cap_sys_admin
```

Modifying process capabilities

- A process can modify its capability sets by:
  - **Raising** a capability (adding it to set)
    - Synonyms: add, enable
  - **Lowering** a capability (removing it from set)
    - Synonyms: **drop**, **clear**, remove, disable
  - Mostly, we'll defer discussion of the APIs until later
- There are various rules about changes a process can make to its capability sets

Outline

Process permitted and effective capabilities

- *Permitted* : capabilities that process *may* employ
  - "Upper bound" on effective capability set
  - Once dropped from permitted set, a capability can't be reacquired
    - (But see discussion of *execve()* later)
  - Can't drop while capability is also in effective set
- *Effective* : capabilities that are currently in effect for process
  - I.e., capabilities that are examined when checking if a process can perform a privileged operation
  - Capabilities can be dropped from effective set and reacquired
    - Operate with least privilege....
    - Reacquisition possible only if capability is in permitted set

[TLPI §39.3.3]

File permitted and effective capabilities

- *Permitted*: a set of capabilities that may be added to process's permitted set during *exec()*
- *Effective*: ⚠ a **single bit** that determines state of process's new effective set after *exec()*:
  - If set, all capabilities in process's new permitted set are also enabled in effective set
    - Useful for so-called *capabilities-dumb* applications
  - If not set, process's new effective set is empty
- File capabilities allow implementation of capabilities analog of set-UID-*root* program
  - Notable difference: setting effective bit off allows a program to start in **unprivileged** state
    - Set-UID/set-GID programs always start in **privileged** state

[TLPI §39.3.4]

Outline

Setting and viewing file capabilities from the shell

- *setcap(8)* sets capabilities on files
  - Requires privilege (`CAP_SETFCAP`)
  - E.g., to set `CAP_SYS_TIME` as a permitted and effective capability on an executable file:

```
$ cp /bin/date mydate
$ sudo setcap "cap_sys_time=pe" mydate
```

- *getcap(8)* displays capabilities associated with a file

```
$ getcap mydate
mydate = cap_sys_time+ep
```

- To list all files on the system that have capabilities, use:
  `sudo filecap -a`
  - *filecap* is part of the *libcap-ng-utils* package

[TLPI §39.3.6]

cap/demo_file_caps.c

```
int main(int argc, char *argv[]) {
    cap_t caps = cap_get_proc();      /* Fetch process capabilities */
    char *str = cap_to_text(caps, NULL);
    printf("Capabilities: %s\n", str);
    ...
    if (argc > 1) {
        fd = open(argv[1], O_RDONLY);
        if (fd >= 0)
            printf("Successfully opened %s\n", argv[1]);
        else
            printf("Open failed: %s\n", strerror(errno));
    }
    exit(EXIT_SUCCESS);
}
```

- Display process capabilities
- Report result of opening file named in *argv[1]* (if present)

cap/demo_file_caps.c

```
$ id -u
1000
$ cc -o demo_file_caps demo_file_caps.c -lcap
$ ./demo_file_caps /etc/shadow
Capabilities: =
Open failed: Permission denied
$ ls -l /etc/shadow
----------. 1 root root 1974 Mar 15 08:09 /etc/shadow
```

- All steps in demos are done from unprivileged user ID 1000
- Binary has no capabilities ⇒ process gains no capabilities
- *open()* of `/etc/shadow` fails
  - Because `/etc/shadow` is readable only by privileged process
  - Process needs `CAP_DAC_READ_SEARCH` capability

cap/demo_file_caps.c

```
$ sudo setcap cap_dac_read_search=p demo_file_caps
$ ./demo_file_caps /etc/shadow
Capabilities: = cap_dac_read_search+p
Open failed: Permission denied
```

- Binary confers permitted capability to process, but capability is not effective
- Process gains capability in permitted set
- *open()* of `/etc/shadow` fails
    - Because `CAP_DAC_READ_SEARCH` is not in *effective* set

cap/demo_file_caps.c

```
$ sudo setcap cap_dac_read_search=pe demo_file_caps
$ ./demo_file_caps /etc/shadow
Capabilities: = cap_dac_read_search+ep
Successfully opened /etc/shadow
```

- Binary confers permitted capability and has effective bit on
- Process gains capability in permitted and effective sets
- *open()* of `/etc/shadow` succeeds

Notes for online practical sessions

- Small groups in **breakout rooms**
  - Write a note into Slack if you have a preferred group
- **We will go faster, if groups collaborate** on solving the exercise(s)
  - You can **share a screen** in your room
- I will circulate regularly between rooms to answer questions
- Zoom has an "**Ask for help**" button...
- **Keep an eye on the `#general` Slack channel**
  - Perhaps with further info about exercise;
  - Or a note that the exercise merges into a break
- When your room has finished, write a message in the Slack channel: **"***** Room X has finished *****"**
  - Then I have an idea of how many people have finished

Shared screen etiquette

- It may help your colleagues if you **use a larger than normal font!**
  - In many environments (e.g., *xterm*, *VS Code*), we can adjust the font size with `Control+Shift+"+"` and `Control+"-"`
  - Or (e.g., *emacs*) hold down `Control` key and use mouse wheel
- **Long shell prompts** make reading your shell session difficult
  - Use `PS1='$ '` or `PS1='# '`
- **Low contrast** color themes are difficult to read; change this if you can
- Turn on **line numbering** in your editor
  - In *vim* use: `:set number`
  - In *emacs* use: `M-x display-line-numbers-mode <RETURN>`
    - `M-x` means `Left-Alt+x`
- For collaborative editing, **relative line-numbering is evil**....
  - In *vim* use: `:set nornu`
  - In *emacs*, the following should suffice:

```
M-: (display-line-numbers-mode) <RETURN>
M-: (setq display-line-numbers 'absolute) <RETURN>
```

    - `M-:` means `Left-Alt+Shift+:`

Using *tmate* in in-person practical sessions

In order to share an X-term session with others, do the following:

- Enter the command *tmate* in an X-term, and you will see the following:

```
$ tmate
...
Connecting to ssh.tmate.io...
Note: clear your terminal before sharing readonly access
web session read only: ...
ssh session read only: ...
web session: ...
ssh session: ssh S0mErAnD0m5Tr1Ng@lon1.tmate.io
```

- Share the last "ssh" string with your colleagues via Slack or another channel
- Your colleagues should paste that string into an X-term...
  - After that, you will be sharing an X-term session in which anyone can type

Exercises

1. Compile and run the `cap/demo_file_caps` program, without adding any capabilities to the file, and verify that the process has no capabilities when it executes the binary:

```
$ cc -o demo_file_caps demo_file_caps.c -lcap
```

   - The string "=" means, all capability sets empty.)

2. Now make the binary set-UID-*root*:

```
$ sudo chown root demo_file_caps   # Change owner to root
$ sudo chmod u+s demo_file_caps    # Turn on set-UID bit
$ ls -l demo_file_caps             # Verify
-rwsr-xr-x. 1 root mtk 8624 Oct  1 13:19 demo_file_caps
```

3. Run the binary and verify that the process gains all capabilities. (The string "`=ep`" means "all capabilities in the permitted and effective sets".)

[Exercise continues on next slide]

Exercises

4. Take the existing set-UID-*root* binary, add a permitted capability to it, and set the effective capability bit:

```
$ sudo setcap cap_dac_read_search=pe demo_file_caps
```

5. When you now run the binary, what capabilities does the process have?
6. Suppose you assign empty capability sets to the binary. When you execute the binary, what capabilities does the process then have?

```
$ sudo setcap = demo_file_caps
```

7. Use the *setcap –r* command to remove capabilities from the binary and verify that when run, it once more grants all capabilities to the process.

Outline

Capabilities-dumb and capabilities-aware applications

- **Capabilities-dumb** application:
  - (Typically) an existing set-UID-*root* binary whose code we can't change
    - Thus, binary does not know to use capabilities APIs (Binary simply uses traditional *set*uid()* APIs)
  - But want to make legacy binary less dangerous than set-UID-*root*
- Converse is **capabilities-aware** application
  - Program that was written/modified to use capabilities APIs
  - Set binary up with file effective capability bit **off**
  - Program "knows" it must use capabilities APIs to enable effective capabilities

Adding capabilities to a capabilities-dumb application

To convert existing set-UID-*root* binary to use file capabilities:

- Setup:
  - Binary remains set-UID-*root*
  - Enable a subset of file permitted capabilities + set effective bit **on**
  - (Note: code of binary isn't changed)
- Operation:
  - When binary is executed, process gets (just the) specified subset of capabilities in its permitted and effective sets
    - IOW: file-capabilities override effect of set-UID-*root* bit, which would normally confer **all** capabilities to process
  - Process UID changes between zero and nonzero automatically raise/lower process's capabilities
    - (Covered in more detail later)

Outline

Textual representation of capabilities

- Both *setcap(8)* and *getcap(8)* work with **textual representations** of capabilities
  - Syntax described in *cap_from_text(3)* man page
- Strings read left to right, containing space-separated clauses
  - (The capability sets are initially considered to be empty)
- Clause: *caps-list operator flags* [*operator flags*] ...
  - *caps-list* is comma-separated list of capability names, or *all*
  - *operator* is =, +, or −
  - *flags* is zero or more of *p* (permitted), *e* (effective), or *i* (inheritable)

Textual representation of capabilities

- Operators:
  - `=` operator:
    - Raise capabilities in sets specified by *flags*; lower those capabilities in remaining sets
    - *caps-list* can be omitted; defaults to *all*
    - *flags* can be omitted ⇒ clear capabilities from all sets ⇒ Thus: `"="` means clear all capabilities in all sets
  - `+` operator: raise capabilities in sets specified by *flags*
  - `-` operator: lower capabilities in sets specified by *flags*
- Clause can contain multiple [*operator flags*] parts:
  - E.g., `cap_sys_time+p-i`
- What does `"=p cap_kill,cap_sys_admin+e"` mean?
  - All capabilities in permitted set, plus `CAP_KILL` and `CAP_SYS_ADMIN` in effective set

Exercises

1. What capability bits are enabled by each of the following text-form capability specifications?
   - `"=p"`
   - `"="`
   - `"cap_setuid=p cap_sys_time+pie"`
   - `"cap_kill=p = cap_sys_admin+pe"`
   - `"cap_chown=i cap_kill=pe cap_setfcap,cap_chown=p"`
   - `"=p cap_kill-p"`
2. The program `cap/cap_text.c` takes a single command-line argument, which is a text-form capability string. It converts that string to an in-memory representation and then iterates through the set of all capabilities, printing out the state of each capability within the permitted, effective, and inheritable sets. It thus provides a method of verifying your interpretation of text-form capability strings. Try supplying each of the above strings as an argument to the program (**remember to enclose the entire string in quotes!**) and check the results against your answers to the previous exercise.

Outline

Transformation of process capabilities during *exec*

- During *execve()*, process's capabilities are transformed:

```
P'(perm) = F(perm) & P(bset)

P'(eff) = F(eff) ? P'(perm) : 0
```

  - *P()* / *P'()*: process capability set before/after *exec*
  - *F()*: file capability set (**of file that is being execed**)
- New permitted set for process comes from file permitted set ANDed with *capability bounding set (bset)*
  - ⚠ Note that *P(perm)* has no effect on *P'(perm)*
- New effective set is either 0 or same as new permitted set
- ⚠ Transformation **rules above are a simplification** that ignores process+file inheritable sets and process ambient set
  - In most cases, those sets are empty (i.e., 0)

Transformation of process capabilities during *exec*

- Commonly, process bounding set contains all capabilities
- Therefore transformation rule for process permitted set:

```
P'(perm) = F(perm) & P(bset)
```

commonly simplifies to:

```
P'(perm) = F(perm)
```

[TLPI §39.5]

Outline

The capability bounding set

- Per-process attribute (actually: per-thread)
- A "safety catch" to limit capabilities that can be gained during *exec*
    - Limits capabilities that can be granted by file permitted set
    - Limits capabilities that can be added to process inheritable set (later)
- Use case: remove some capabilities from bounding set to ensure process never regains them on *execve()*
    - E.g., *systemd* reduces bounding set before executing some daemons
        - Guarantees that daemon can **never** get certain capabilities
    - `cap/reduced_bounding_set_procs.sh` displays list of processes that have a reduced bounding set

The capability bounding set

- Inherited by child of *fork()*, preserved across *execve()*
  - *init* starts with capability bounding set containing **all capabilities**
- Two methods of getting:
  - *prctl()* `PR_CAPBSET_READ` (for self)
    - Higher-level *libcap* API: *cap_get_bound(3)*
  - `/proc/PID/status` `CapBnd` entry (any process)
- Can (irreversibly) drop capabilities from bounding set
  - *prctl()* `PR_CAPBSET_DROP`
  - Requires `CAP_SETPCAP` effective capability
  - Doesn't change permitted, effective, and inheritable sets
  - Higher-level *libcap* API: *cap_drop_bound(3)*

[TLPI §39.5.1]