jambit Abendvortrag – "Containers unplugged"

Privileged Programs

Michael Kerrisk, man7.org © 2019

mtk@man7.org

2 April 2019, Munich

Outline

Who am I?

- Contributor to Linux *man-pages* project since 2000
  - Maintainer since 2004
    - Maintainer email: `mtk.manpages@gmail.com`
  - Project provides ≈1050 manual pages, primarily documenting system calls and C library functions
    - *https://www.kernel.org/doc/man-pages/*
- Author of a book on the Linux programming interface
  - *http://man7.org/tlpi/*
- Trainer/writer/engineer
  - Lots of courses at *http://man7.org/training/*
- Email: `mtk@man7.org`
  Twitter: `@mkerrisk`

Outline

Process credentials

- Each process has a number of UIDs and GIDs:
  - Real UID + real GID [process ownership]
    - Login shell gets these IDs from `/etc/passwd`
  - Effective UID + effective GID [permission checking]
    - More on these IDs in a moment
  - Saved set-user-ID + saved set-group-ID
    - Initialized during *execve()*
    - (More on these IDs in soon)
  - Supplementary GIDs [permission checking]
    - Login shell gets group memberships from `/etc/group`
- Credentials are inherited by child of *fork()*

Retrieving process credentials

APIs for retrieving credentials:

- Real IDS:
  `ruid = getuid()`
  `rgid = getgid()`
- Effective IDs:
  `euid = geteuid()`
  `egid = getegid()`
- Real, effective, and saved set IDS:
  `getresuid(&ruid, &euid, &suid)`
  `getresgid(&ruid, &euid, &suid)`
  - Not in POSIX, but present on Linux, BSDs, + some others
- Supplementary group IDs:
  `ngroups = getgroups(size, gidlist[])`

Effective UID and GID

- Determine permissions for performing various operations (in conjunction with supplementary GIDs)
  - Example: files have user and group owner + RWX permissions for user/group/other
- Effective UID 0 is special: has many privileges
  - a.k.a. *root* or superuser
- Normally, effective IDs have same values as corresponding real IDs
- Can differ when set-user-ID or set-group-ID program is executed

Outline

Set-user-ID and set-group-ID programs

- Mechanism that allows a program to operate with privileges of another user or group
- Examples: *passwd(1)*, *mount(8)*, *su(1)*
- Let's distinguish two kinds of privilege:
  - Set-UID-*root* programs
    - Confer effective UID 0
    - Give full *root* privileges (dangerous!)
  - Set-UID (or set-GID) programs that confer privileges of another (nonzero) UID (or another GID)

[TLPI §9.3]

Set-user-ID and set-group-ID programs

Overview of operation:

- Like any file, an executable has a user and a group owner
- Program is made set-UID by enabling set-UID mode bit:
  - `chmod u+s file`
  - For set-GID programs: `chmod g+s file`
- When executing set-UID program, kernel makes effective UID of process same as UID of file
  - ⇒ Process obtains same privileges as owner of executable
  - (If set-UID bit is not enabled, then process effective UID is not changed during *exec()*)
- Analogously for set-GID bit...
- ⚠ Set-UID and set-GID bits are ignored for shell scripts

Saved set-user-ID and saved set-group-ID

- Designed for use with set-UID/set-GID programs
- When a program is execed:
  1. Set-UID bit enabled on executable? $\Rightarrow$ process effective UID made same as file UID
  2. Set-GID bit enabled on executable? $\Rightarrow$ process effective GID made same as file GID
  3. Effective IDs are copied to corresponding saved set IDs
     - (Done regardless of whether set-UID or set-GID bit is set)
- IOW: Saved set IDs record state of effective IDs at program start up

Saved set-user-ID and saved set-group-ID

- When set-UID program is executed, credentials look like this:

Real UID (unchanged by *exec()*)	**Effective UID** (copied from file owner)	**Saved set-user-ID** (copied from eff. UID at program start-up)
Unprivileged ID	Privileged ID	Privileged ID

- A process can switch its effective UID back and forth between real UID and saved set-user-ID
  - i.e., between unprivileged and privileged states
- Analogously for set-GID programs and saved set-group-ID
- What is the design mistake in initial set-up of process UIDs in above picture?
  - In other words: what is the first thing that a set-UID / set-GID program should do on start-up?

Outline

Changing process credentials

- It's a mess....
- Various APIs for updating process credentials, but:
  - Set of IDs changed by some APIs differs according to whether process is privileged
    - Privileged $\approx$ process has effective UID 0
  - For some of the APIs, rules about which IDs are changed are surprisingly complex
  - The "best" APIs are not standardized (and are unavailable on some systems)

[TLPI §9.7]

Changing process credentials

- **Be very careful!!**
- Best practice
  - Call *set*id()*
  - Check if call succeeded
  - Use *get*id()* to verify change

Changing process credentials

General principle for all APIs that change credentials:

- **Privileged processes** can make any changes to IDs
  - Privileged process $\approx$ process effective user ID 0
    - More precisely: process has appropriate Linux capability (`CAP_SETUID` for UID changes, `CAP_SETGID` for GID changes)
- **Unprivileged processes** can change an ID to same value as another of its current IDs
  - e.g., unprivileged *setuid()* can change effective UID to same value as real or saved set UID

[TLPI §9.7]

Changing process UIDs

There are various APIs for changing process UIDs:

- `setuid(u)`: in privileged process: change **real**, **effective**, and **saved set** UIDs to *u*
  - ⚠ Unprivileged process: changes only **effective** UID
  - Privileged == process has `CAP_SETUID` capability
- `seteuid(euid)`: change **effective** UID
- `setreuid(ruid, euid)`: change **real** & **effective** UID
  - -1 means "no change" in corresponding UID
  - If *ruid* != -1 or *euid* != [real UID before call], also changes **saved set-user-ID** (to *euid*)

Changing process UIDs

- `setresuid(ruid, euid, suid)`: change **real**, **effective**, and **saved** set UIDs
  - -1 means "no change" in corresponding UID
  - Most **precise** API: changes only specified UIDs
  - **Not standardized** and available on only some systems
    - (Linux, FreeBSD, OpenBSD, HP-UX)

Changing process GIDs

- Exactly analogous APIs for changing process group IDs:
  - `setgid(gid)`
    - If process has `CAP_SETGID`, all three GIDs are changed
  - `setegid(egid)`
  - `setregid(rgid, egid)`
  - `setresgid(rgid, egid, sgid)`

Exercises

1. Write a program (**[template: `proccred/ex.setuid_expt.c`]**) that retrieves (*getresuid()*) and prints out its real, effective, and saved set UIDs. Compile the program. Then change the ownership of the executable to be another user, set the set-UID bit on the executable, and make it executable by any user:

```
$ sudo chown <user> <file>
$ sudo chmod u+s,go+x <file>
```

   Run the program and verify that it executes with the effective UID of the owner of the program file.

2. Extend the previous program as follows, retrieving and displaying the real, effective, and saved set UIDs after each step:
   - Temporarily drop the privileged UID (i.e., set the effective UID to same value as the real UID, while retaining the privileged UID in the saved set-user-ID).
   - Regain the privileged UID.

   [Exercise continues on the next slide]

Exercises

- Permanently drop the privileged UID (i.e., the effective and saved set UIDs are set the same as the real UID).
- Attempt once more to regain the privileged UID. What happens?

Hints:

- You will need to reset the file ownership and reenable the set-UID mode bit each time you recompile the executable.
- **Don't forget to include error checking on each *set*id()* call**.
- If you are having problems making your set-UID program work, check that your filesystem is not mounted with the *nosuid* option.

Exercises

3. Suppose that a set-UID-*root* program creates a child process that uses *execve()* to execute a second program. What are the credentials (effective UID and saved-setUID) of the child process before and after it performs the *execve()*? Does the answer to the question change if the set-UID program drops privilege (i.e., makes its effective UID the same as its real UID, while retaining zero in the saved set-UID) before performing the *execve()*? Write programs to verify your answers. (The program `proccred/idshow.c` may be useful.)

Outline

Operate with least privilege

- Generally best to hold privilege only when required
  - "Principle of least privilege"
  - If program is compromised while unprivileged, potential for damage is minimized
- Drop privilege when not needed, and raise temporarily as required
  - i.e., switch effective ID back and forth between real and saved set ID
- If privilege will never again be needed, drop it permanently
  - i.e., set effective and saved set IDs to same value as real ID

Dropping and raising privileges

- Drop and raise privileges:

```
euid = geteuid();     /* Save copy of eUID */
seteuid(getuid());    /* Drop (switch to rUID) */

seteuid(euid);        /* Raise (restore eUID) */
/* Do privileged work */
seteuid(getuid());    /* Drop (switch to rUID) */
```

- Alternatively (non-POSIX):

```
euid = geteuid();                /* Save eUID */
setresuid(-1, getuid(), -1);     /* Drop */

setresuid(-1, euid, -1);         /* Raise */
/* Do privileged work */
setresuid(-1, getuid(), -1);     /* Drop */
```

Dropping privileges permanently

- Irrevocably drop privileges:

```
setreuid(getuid(), getuid());
        /* Make all UIDs same as rUID */
```

  - **Remember:** *setreuid()* also changes saved-set-UID (to new eUID) if *ruid != -1* or *euid* != real UID before call(!!)

- Alternatively (non-POSIX):

```
setresuid(-1, getuid(), getuid());
```

Security of set-user-ID and set-group-ID programs

Set-UID program owned by *root* (UID 0) gives superuser privileges

- Useful and powerful technique, but...
- Opens door for security exploits in poorly written programs
  - Many pitfalls (especially in C)
  - See TLPI Ch. 38, and also sources listed in TLPI §38.12
- Avoid set-UID-*root* programs if possible
  - Use dedicated user ID instead

Capabilities

Capabilities are another alternative to set-UID-*root*

- Divide superuser privilege into small pieces
  - Capabilities can be associated with executable files
  - Linux-specific
- See TLPI Ch. 39 and *capabilities(7)*
- But:
  - More work to program
  - Some capabilities can be leveraged to full *root* in some circumstances
  - Some capabilities are too broad (e.g., `CAP_SYS_ADMIN`)
    - See https://lwn.net/Articles/486306/

Thanks!

Michael Kerrisk mtk@man7.org @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/

The Linux Programming Interface, http://man7.org/tlpi/