Programmation Systèmes
Cours 4 — Runtime user management

Stefano Zacchiroli
zack@pps.jussieu.fr

Laboratoire PPS, Université Paris Diderot - Paris 7

20 Octobre 2011

URL http://upsilon.cc/zack/teaching/1112/progsyst/
Copyright © 2011 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 3.0 Unported License
http://creativecommons.org/licenses/by-sa/3.0/


Access control on UNIX — generalities

Traditionally in UNIX systems, access permissions have been granted to specific users or groups of users. Each of those realm is identified by IDs: user IDs and group IDs.

As resources are accessed by processes, processes are associated to user and group IDs that are used for access control.

Accessing a forbidden resource usually results in a EPERM error.

If a process wants to access a resource which is currently forbidden to him, it has to change its sets of user and group IDs.

Principle of least privilege

Principle of least privilege

Programs should use the least privilege necessary to accomplish any given task.

The application of this principle reduces the impact of security breaches to a minimum (by definition) of privileged actions an attacker will be able to perform.

Example

- binding Internet ports < 1024 is a privileged operation on UNIX
- a web server that needs to serve on port 80 file system content should
  1. start as a user that can bind port 80 (usually root)
  2. bind port 80
  3. change its user ID to a non-privileged user that can read the files it should serve (bonus point: the change is irreversible)

User IDs

Every process is associated to 6 or more IDs that are relevant for user management:

who we really are

1. real user ID
2. real group ID

access permission checks

3. effective user ID
4. effective group ID

... supplementary group IDs

"backup" copies

5. saved set-user-ID
6. saved set-group-ID

User IDs (cont.)

Real IDs are tied to the identify of the user initiating processes

- they are set upon login
- retrieved from `/etc/passwd`
- can be changed (by super user)

Effective IDs

- used for most kinds of access checks, e.g. file access permissions
- allow to complement user's primary group ID with supplementary group IDs
- they are usually equal to real IDs, but can diverge from that

Set-user-ID

2 of the file access permission bits can be used to change effective IDs upon execution of a program.

- the set-user-ID bit sets the *effective user ID* of the process to the owner of the program file
- the set-group-ID bit sets the *effective group ID* of the process to the owner of the program file

Typical use case: enable non privileged users to perform an action that requires specific privileges via the usage of a (carefully crafted!) set-user-ID or set-group-ID program.

Example (passwd)

The passwd program manipulates sensitive files (e.g. /etc/passwd, /etc/shadow) that should be writable and/or readable only by root. Still, passwd should allow random users to change their password.

Set-user-ID — example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include "apue.h"

int main(int argc, char **argv) {
        char cmd[1024];

        if (argc < 2) {
                printf("Usage: grep-shadow USERNAME\n");
                exit(EXIT_FAILURE);
        }
        /* Retrieve date of last password change; see shadow(5). */
        /* Do not try this at home! */
        if (snprintf(cmd, sizeof(cmd),
                        "grep ^%s: /etc/shadow | cut -d: -f 3", argv[1]) < 0)
                err_sys("snprintf error");
        if ((system(cmd)) == -1)
                err_sys("system error");

        exit(EXIT_SUCCESS);
}
```

Set-user-ID — example (cont.)

```
$ ./grep-shadow zack
grep: /etc/shadow: Permission denied

$ su
Password:
# cp grep-shadow grep-shadow2
# chown root grep-shadow2
# chmod u+s grep-shadow2
# exit
$ ls -l grep-shadow grep-shadow2
-rwxr-xr-x 1 zack zack 7907 ott 18 14:56 grep-shadow
-rwsr-xr-x 1 root root 7907 ott 18 14:58 grep-shadow2

$ ./grep-shadow2 zack
14611
```

- for `grep-shadow`, real user ID = effective user ID = zack
  - not enough to access `/etc/shadow`, which is `-rw-r-----`
- for `grep-shadow2`, real user ID = zack, effective user ID = root
  - it is enough to access `/etc/shadow`

Set-user-ID programs are risky

In some sense, set-user-ID programs adhere to the principle of least privilege: the user has set-user-ID privileges only for the duration of program execution and only for that process. Is that enough?

Exercise (unexpected usage of grep-shadow2)

- *./grep-shadow2 ''‘echo zack‘''*
- *./grep-shadow2 ''‘cat /etc/shadow ; echo zack‘''*
- *./grep-shadow2 ''‘cat /etc/shadow > shadow.txt ; echo zack‘''*

what's the problem with grep-shadow2?

Set-user-ID programs are risky (cont.)

In some sense, set-user-ID programs adhere to the principle of least privilege: the user has set-user-ID privileges only for the duration of program execution and only for that process. Is that enough?

Exercise (unexpected usage of grep-shadow2)

- *./grep-shadow2 ''‘echo zack‘''*
  - *system("grep ''‘echo zack‘'': /etc/shadow | cut ...");*
- *./grep-shadow2 ''‘cat /etc/shadow ; echo zack‘''*
  - *system("grep ''‘cat /etc/shadow ; echo zack‘'': /etc/shadow | cut ...");*
- *./grep-shadow2 ''‘cat /etc/shadow > shadow.txt ; echo zack‘''*
  - *system("grep ''‘cat /etc/shadow > shadow.txt ; echo zack‘'': /etc/shadow | cut ...");*

what's the problem with grep-shadow2?

Shell script injection!

Set-user-ID programs are risky (cont.)

- The shell language is a very powerful language
- Allowing free form user input to be executed with high privileges as shell snippets it's a very dangerous security breach
- In this specific case, the problem can be solved sanitizing user input (i.e. `argv[1]`) before passing it to `system`, ensuring that:
  1. it is a valid user name
  2. it does not contain any shell meta-character
- in the general case, ensuring set-user-ID (or set-group-ID) programs won't be used for purposes other that the intended ones is very difficult

1. You should avoid set-user-ID architectures whenever possible.
2. You should never call `system` from a set-user-ID program.

Retrieving user/group IDs

#include <unistd.h>

uid_t getuid(**void**);

Returns: *real user ID of calling process*

gid_t getgid(**void**);

Returns: *real group ID of calling process*

uid_t geteuid(**void**);

Returns: *effective user ID of calling process*

gid_t getegid(**void**);

Returns: *effective group ID of calling process*

`uid_t` (`gid_t`) is an integer type representing user (group) IDs

Retrieving user/group IDs — example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
        printf("real user ID:\t%d\n",  getuid());
        printf("real group ID:\t%d\n", getgid());
        printf("eff. user ID:\t%d\n",  geteuid());
        printf("eff. group ID:\t%d\n", getegid());
        exit(EXIT_SUCCESS);
}
```

Retrieving user/group IDs — example (cont.)

```
$ ./getids
real user ID: 1000
real group ID: 1000
eff. user ID: 1000
eff. group ID: 1000

$ su
Password:
# cp getids getids2
# chgrp root getids2
# chmod g+s getids2
# exit

$ ./getids2
real user ID: 1000
real group ID: 1000
eff. user ID: 1000
eff. group ID: 0
$
```

Changing real IDs

Real user (and group) ID can be changed using the system calls:

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

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

Obviously, we need some safeguard measures:[1]

- if the invoking process has superuser privileges, setuid sets all 3 IDs to *uid*
- otherwise, if *uid*=real uid or *uid*=saved set-user-ID, setuid only sets the effective uid
- otherwise, EPERM is raised

[1] only stated for uid; similar rules exist for gid

Changing real IDs — discussion

A few general rules can be stated about ID maintenance in the kernel:

1. normally, the real user ID is set upon login and never changes throughout a session
2. only the superuser can change real user IDs; that is what `login` (a superuser process) does upon login
3. exec copies effective uid to saved set-user-ID
   - in case of set-user-ID programs, the copy happens *after* having inherited effective uid from the executable
4. a process can freely change its effective user ID to either its real user ID or its saved set-user-ID

Case study: man

The man utility can be installed set-user-ID to the man user, in order to maintain a system-wide cache of formatted manual pages.
To format a manual page, man invokes a number of external programs.

Goals

1. save formatted manual pages to a location writable only by the man user;
2. execute external programs as the invoking user, to avoid that problems with them affect man data.

We have to switch among two different privilege realms—man's privileges and user's privileges—in the following order:
man → user → man.

Case study: man (cont.)

1. we execute /usr/bin/man, which is set-user-ID man. We have:

real uid	=	our user ID
effective uid	=	man
saved set-user-ID	=	man

2. if needed, man can access restricted data

3. before executing external programs, man calls setuid(getuid()), which changes only effective uid.[2] We now have:

real uid	=	our user ID	(no change)
effective uid	=	our user ID	
saved set-user-ID	=	man	(no change)

[2] because we are not a superuser process and setuid's argument = real uid

Case study: man (cont.)

4. man execute external formatting programs, which are now run as user processes

5. once done, man calls `setuid(`*manuid*`)`. This is allowed because man's user ID has been saved in saved set-user-ID. We now have:

real uid	=	our user ID	(no change)
effective uid	=	man	
saved set-user-ID	=	man	(no change)

6. man can now update the cache (which is write-restricted to man)

If `setuid` didn't allow to go back to saved set-user-ID, man would be tempted to retain extra privileges while executing external programs, increasing security risks.

Effective IDs

For more fine grained control (and clearer code), two system calls allow to change only effective IDs:

```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

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

- A privileged user using them will leave real IDs unaffected.
- A non-privileged user can change its effective IDs only to its real or saved set-user-ID (as it happens with `setuid`).

Supplementary groups

Each user is associated to a primary group ID, according to `/etc/passwd`. Additional groups a user is member of are specified in `/etc/group`.

In early UNIXes, each user belonged to a single group at any point in time. The command `newgrp` was used to change it (choosing among all allowed groups for the user).

In modern UNIXes users belong at the same time to the primary and all supplementary groups (up to a maximum). Access permissions are checked against all those groups.

- the need of explicitly changing group is mostly gone
  (Which is why you never heard of `newgrp`)

Accessing supplementary groups

Supplementary groups can be retrieved with:

```
#include <unistd.h>

int getgroups(int size, git_d list[]);
                    Returns: number of supplementary groups if OK, -1 on error
```

Passing *size*==0 can be used to evaluate the size of the list.

POSIX offers no way to set supplementary groups. Only superuser can do that and there is little use of it beside `login` implementation. On Linux:

```
#include <grp.h>

int setgroups(size_t size, const git_t *list);

int initgroups(const char *user, gid_t group);
```
