9

PROCESS CREDENTIALS

Every process has a set of associated numeric user identifiers (UIDs) and group identifiers (GIDs). Sometimes, these are referred to as process credentials. These identifiers are as follows:

- real user ID and group ID;
- effective user ID and group ID;
- saved set-user-ID and saved set-group-ID;
- file-system user ID and group ID (Linux-specific); and
- supplementary group IDs.

In this chapter, we look in detail at the purpose of these process identifiers and describe the system calls and library functions that can be used to retrieve and change them. We also discuss the notion of privileged and unprivileged processes, and the use of the set-user-ID and set-group-ID mechanisms, which allow the creation of programs that run with the privileges of a specified user or group.

9.1 Real User ID and Real Group ID

The real user ID and group ID identify the user and group to which the process belongs. As part of the login process, 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

(Section 8.1). When a new process is created (e.g., when the shell executes a program), it inherits these identifiers from its parent.

9.2 Effective User ID and Effective Group ID

On most UNIX implementations (Linux is a little different, as explained in Section 9.5), the effective user ID and group ID, in conjunction with the supplementary group IDs, are used to determine the permissions granted to a process when it tries to perform various operations (i.e., system calls). For example, these identifiers determine the permissions 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. As we'll see in Section 20.5, the effective user ID is also used by the kernel to determine whether one process can send a signal to another.

A process whose effective user ID is 0 (the user ID of *root*) has all of the privileges of the superuser. Such a process is referred to as a *privileged process*. Certain system calls can be executed only by privileged processes.

> In Chapter 39, we describe Linux's implementation of capabilities, a scheme that divides the privileges granted to the superuser into a number of distinct units that can be independently enabled and disabled.

Normally, the effective user and group IDs have the same values as the corresponding real IDs, but there are two ways in which the effective IDs can assume different values. One way is through the use of system calls that we discuss in Section 9.7. The second way is through the execution of set-user-ID and set-group-ID programs.

9.3 Set-User-ID and Set-Group-ID Programs

A set-user-ID program allows a process to gain privileges it would not normally have, by setting the process's effective user ID to the same value as the user ID (owner) of the executable file. A set-group-ID program performs the analogous task for the process's effective group ID. (The terms *set-user-ID program* and *set-group-ID* program are sometimes abbreviated as *set-UID program* and *set-GID program*.)

Like any other file, an executable program file has an associated user ID and group ID that define the ownership of the file. In addition, an executable file has two special permission bits: the set-user-ID and set-group-ID bits. (In fact, every file has these two permission bits, but it is their use with executable files that interests us here.) These permission bits are set using the *chmod* command. An unprivileged user can set these bits for files that they own. A privileged user (`CAP_FOWNER`) can set these bits for any file. Here's an example:

```
$ su
Password:
# ls -l prog
-rwxr-xr-x    1 root     root       302585 Jun 26 15:05 prog
# chmod u+s prog                        Turn on set-user-ID permission bit
# chmod g+s prog                        Turn on set-group-ID permission bit
```

As shown in this example, it is possible for a program to have both of these bits set, although this is uncommon. When *ls −l* is used to list the permissions for a program that has the set-user-ID or set-group-ID permission bit set, then the *x* that is normally used to indicate that execute permission is set is replaced by an *s*:

```
# ls -l prog
-rwsr-sr-x    1 root     root       302585 Jun 26 15:05 prog
```

When a set-user-ID program is run (i.e., loaded into a process's memory by an *exec()*), the kernel sets the effective user ID of the process to be the same as the user ID of the executable file. Running a set-group-ID program has an analogous effect for the effective group ID of the process. Changing the effective user or group ID in this way gives a process (in other words, the user executing the program) privileges it would not normally have. For example, if an executable file is owned by *root* (superuser) and has the set-user-ID permission bit enabled, then the process gains superuser privileges when that program is run.

Set-user-ID and set-group-ID programs can also be designed to change the effective IDs of a process to something other than *root*. For example, to provide access to a protected file (or other system resource), it may suffice to create a special-purpose user (group) ID that has the privileges required to access the file, and create a set-user-ID (set-group-ID) program that changes the effective user (group) ID of a process to that ID. This permits the program to access the file without allowing it all of the privileges of the superuser.

Sometimes, we'll use the term set-user-ID-*root* to distinguish a set-user-ID program that is owned by *root* from one owned by another user, which merely gives a process the privileges accorded to that user.

> We have now started using the term *privileged* in two different senses. One is the sense defined earlier: a process with an effective user ID of 0, which has all of the privileges accorded to *root*. However, when we are talking about a set-user-ID program owned by a user other than *root*, we'll sometimes refer to a process as gaining the privileges accorded to the user ID of the set-user-ID program. Which sense of the term *privileged* we mean in each case should be clear from the context.
>
> For reasons that we explain in Section 38.3, the set-user-ID and set-group-ID permission bits don't have any effect for shell scripts on Linux.

Examples of commonly used set-user-ID programs on Linux include: *passwd(1)*, which changes a user's password; *mount(8)* and *umount(8)*, which mount and unmount file systems; and *su(1)*, which allows a user to run a shell under a different user ID. An example of a set-group-ID program is *wall(1)*, which writes a message to all terminals owned by the *tty* group (normally, every terminal is owned by this group).

In Section 8.5, we noted that the program in Listing 8-2 needed to be run from a *root* login so that it could access the `/etc/shadow` file. We could make this program runnable by any user by making it a set-user-ID-*root* program, as follows:

```
$ su
Password:
# chown root check_password              Make this program owned by root
# chmod u+s check_password               With the set-user-ID bit enabled
```

```
# ls -l check_password
-rwsr-xr-x    1 root   users    18150 Oct 28 10:49 check_password
# exit
$ whoami                                  This is an unprivileged login
mtk
$ ./check_password                        But we can now access the shadow
Username: avr                             password file using this program
Password:
Successfully authenticated: UID=1001
```

The set-user-ID/set-group-ID technique is a useful and powerful tool, but one that can result in security breaches in applications that are poorly designed. In Chapter 38, we list a set of good practices that should be observed when writing set-user-ID and set-group-ID programs.

9.4 Saved Set-User-ID and Saved Set-Group-ID

The saved set-user-ID and saved set-group-ID are designed for use with set-user-ID and set-group-ID programs. When a program is executed, the following steps (among many others) occur:

1. 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.
2. 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.

As an example of the effect of the above steps, 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 follows:

```
real=1000 effective=0 saved=0
```

Various system calls allow a set-user-ID program to switch its effective user ID between the values of the real user ID and the saved set-user-ID. Analogous system calls allow a set-group-ID program to modify its effective group ID. In this manner, the program can temporarily drop and regain whatever privileges are associated with the user (group) ID of the execed file. (In other words, the program can move between the states of potentially being privileged and actually operating with privilege.) As we'll elaborate in Section 38.2, it is secure programming practice for set-user-ID and set-group-ID programs to operate under the unprivileged (i.e., real) ID whenever the program doesn't actually need to perform any operations associated with the privileged (i.e., saved set) ID.

> The saved set-user-ID and saved set-group-ID are sometimes synonymously referred to as the *saved user ID* and *saved group ID*.

The saved set IDs are a System V invention adopted by POSIX. They were not provided on releases of BSD prior to 4.4. The initial POSIX.1 standard made support for these IDs optional, but later standards (starting with FIPS 151-1 in 1988) made support mandatory.

9.5 File-System User ID and File-System Group ID

On Linux, it is the file-system user and group IDs, rather than the effective user and group IDs, that are used (in conjunction with the supplementary group IDs) to determine permissions when performing file-system operations such as opening files, changing file ownership, and modifying file permissions. (The effective IDs are still used, as on other UNIX implementations, for the other purposes described earlier.)

Normally, the file-system user and group IDs have the same values as the corresponding effective IDs (and thus typically are the same as the corresponding real IDs). Furthermore, whenever the effective user or group ID is changed, either by a system call or by execution of a set-user-ID or set-group-ID program, the corresponding file-system ID is also changed to the same value. Since the file-system IDs follow the effective IDs in this way, this means that Linux effectively behaves just like any other UNIX implementation when privileges and permissions are being checked. The file-system IDs differ from the corresponding effective IDs, and hence Linux differs from other UNIX implementations, only when we use two Linux-specific system calls, *setfsuid()* and *setfsgid()*, to explicitly make them different.

Why does Linux provide the file-system IDs and in what circumstances would we want the effective and file-system IDs to differ? The reasons are primarily historical. The file-system IDs first appeared in Linux 1.2. In that kernel version, one process could send a signal to another if the effective user ID of the sender matched the real or effective user ID of the target process. This affected certain programs such as the Linux NFS (Network File System) server program, which needed to be able to access files as though it had the effective IDs of the corresponding client process. However, if the NFS server changed its effective user ID, it would be vulnerable to signals from unprivileged user processes. To prevent this possibility, the separate file-system user and group IDs were devised. By leaving its effective IDs unchanged, but changing its file-system IDs, the NFS server could masquerade as another user for the purpose of accessing files without being vulnerable to signals from user processes.

From kernel 2.0 onward, Linux adopted the SUSv3-mandated rules regarding permission for sending signals, and these rules don't involve the effective user ID of the target process (refer to Section 20.5). Thus, the file-system ID feature is no longer strictly necessary (a process can nowadays achieve the desired results by making judicious use of the system calls described later in this chapter to change the value of the effective user ID to and from an unprivileged value, as required), but it remains for compatibility with existing software.

Since the file-system IDs are something of an oddity, and they normally have the same values as the corresponding effective IDs, in the remainder of this book, we'll generally describe various file permission checks, as well as the setting of the

ownership of new files, in terms of the effective IDs of a process. Even though the process's file-system IDs are really used for these purposes on Linux, in practice, their presence seldom makes an effective difference.

9.6 Supplementary Group IDs

The supplementary group IDs are a set of additional groups to which a process belongs. A new process inherits these IDs from its parent. A login shell obtains its supplementary group IDs from the system group file. As noted above, these IDs are used in conjunction with the effective and file-system IDs to determine permissions for accessing files, System V IPC objects, and other system resources.

9.7 Retrieving and Modifying Process Credentials

Linux provides a range of system calls and library functions for retrieving and changing the various user and group IDs that we have described in this chapter. Only some of these APIs are specified in SUSv3. Of the remainder, several are widely available on other UNIX implementations and a few are Linux-specific. We note portability issues as we describe each interface. Toward the end of this chapter, Table 9-1 summarizes the operation of all of the interfaces used to change process credentials.

As an alternative to using the system calls described in the following pages, the credentials of any process can be found by examining the `Uid`, `Gid`, and `Groups` lines provided in the Linux-specific `/proc/`*PID*`/status` file. The `Uid` and `Gid` lines list the identifiers in the order real, effective, saved set, and file system.

In the following sections, we use the traditional definition of a privileged process as one whose effective user ID is 0. However, Linux divides the notion of superuser privileges into distinct capabilities, as described in Chapter 39. Two capabilities are relevant for our discussion of all of the system calls used to change process user and group IDs:

- The `CAP_SETUID` capability allows a process to make arbitrary changes to its user IDs.
- The `CAP_SETGID` capability allows a process to make arbitrary changes to its group IDs.

9.7.1 Retrieving and Modifying Real, Effective, and Saved Set IDs

In the following paragraphs, we describe the system calls that retrieve and modify the real, effective, and saved set IDs. There are several system calls that perform these tasks, and in some cases their functionality overlaps, reflecting the fact that the various system calls originated on different UNIX implementations.

Retrieving real and effective IDs

The *getuid()* and *getgid()* system calls return, respectively, the real user ID and real group ID of the calling process. The *geteuid()* and *getegid()* system calls perform the corresponding tasks for the effective IDs. These system calls are always successful.

```
#include <unistd.h>

uid_t getuid(void);
                                        Returns real user ID of calling process
uid_t geteuid(void);
                                        Returns effective user ID of calling process
gid_t getgid(void);
                                        Returns real group ID of calling process
gid_t getegid(void);
                                        Returns effective group ID of calling process
```

Modifying effective IDs

The *setuid()* system call changes the effective user ID—and possibly the real user ID and the saved set-user-ID—of the calling process to the value given by the *uid* argument. The *setgid()* system call performs the analogous task for the corresponding group IDs.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
                                        Both return 0 on success, or −1 on error
```

The rules about what changes a process can make to its credentials using *setuid()* and *setgid()* depend on whether the process is privileged (i.e., has an effective user ID equal to 0). The following rules apply to *setuid()*:

1. When an unprivileged process calls *setuid()*, only the effective user ID of the process is changed. Furthermore, it can be changed only to the same value as either the real user ID or saved set-user-ID. (Attempts to violate this constraint yield the error EPERM.) This means that, for unprivileged users, this call is useful only when executing a set-user-ID program, since, for the execution of normal programs, the process's real user ID, effective user ID, and saved set-user-ID all have the same value. On some BSD-derived implementations, calls to *setuid()* or *setgid()* by an unprivileged process have different semantics from other UNIX implementations: the calls change the real, effective, and saved set IDs (to the value of the current real or effective ID).
2. When a privileged process executes *setuid()* with a nonzero argument, then the real user ID, effective user ID, and saved set-user-ID are all set to the value specified in the *uid* argument. This is a one-way trip, in that 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. If this is not desired, then either *seteuid()* or *setreuid()*, which we describe shortly, should be used instead of *setuid()*.

The rules governing the changes that may be made to group IDs using *setgid()* are similar, but with *setgid()* substituted for *setuid()* and *group* for *user*. With these

changes, rule 1 applies exactly as stated. In rule 2, since changing the group IDs doesn't cause a process to lose privileges (which are determined by the effective *user* ID), privileged programs can use *setgid()* to freely change the group IDs to any desired values.

The following call is the preferred method for a set-user-ID-*root* program whose effective user ID is currently 0 to irrevocably drop all privileges (by setting both the effective user ID and saved set-user-ID to the same value as the real user ID):

```
if (setuid(getuid()) == -1)
    errExit("setuid");
```

A set-user-ID program owned by a user other than *root* can use *setuid()* to switch the effective user ID between the values of the real user ID and saved set-user-ID for the security reasons described in Section 9.4. However, *seteuid()* is preferable for this purpose, since it has the same effect, regardless of whether the set-user-ID program is owned by *root*.

A process can use *seteuid()* to change its effective user ID (to the value specified by *euid*), and *setegid()* to change its effective group ID (to the value specified by *egid*).

```
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Both return 0 on success, or −1 on error

The following rules govern the changes that a process may make to its effective IDs using *seteuid()* and *setegid()*:

1. An unprivileged process can change an effective ID only to the same value as the corresponding real or saved set ID. (In other words, for an unprivileged process, *seteuid()* and *setegid()* have the same effect as *setuid()* and *setgid()*, respectively, except for the BSD portability issues noted earlier.)
2. A privileged process can change an effective ID to any value. If a privileged process uses *seteuid()* to change its effective user ID to a nonzero value, then it ceases to be privileged (but may be able to regain privilege via the previous rule).

Using *seteuid()* is the preferred method for set-user-ID and set-group-ID programs to temporarily drop and later regain privileges. Here's an example:

```
euid = geteuid();               /* Save initial effective user ID (which
                                   is same as saved set-user-ID) */
if (seteuid(getuid()) == -1)    /* Drop privileges */
    errExit("seteuid");
if (seteuid(euid) == -1)        /* Regain privileges */
    errExit("seteuid");
```

Originally derived from BSD, *seteuid()* and *setegid()* are now specified in SUSv3 and appear on most UNIX implementations.

In older versions of the GNU C library (*glibc* 2.0 and earlier), *seteuid(euid)* is implemented as *setreuid(−1, euid)*. In modern versions of *glibc*, *seteuid(euid)* is implemented as *setresuid(−1, euid, −1)*. (We describe *setreuid()*, *setresuid()*, and their group analogs shortly.) Both implementations permit us to specify *euid* as the same value as the current effective user ID (i.e., no change). However, SUSv3 doesn't specify this behavior for *seteuid()*, and it is not possible on some other UNIX implementations. Generally, this potential variation across implementations is not apparent, since, in normal circumstances, the effective user ID has the same value as either the real user ID or the saved set-user-ID. (The only way in which we can make the effective user ID differ from both the real user ID and the saved set-user-ID on Linux is via the use of the nonstandard *setresuid()* system call.)

In all versions of *glibc* (including modern ones), *setegid(egid)* is implemented as *setregid(−1, egid)*. As with *seteuid()*, this means that we can specify *egid* as the same value as the current effective group ID, although this behavior is not specified in SUSv3. It also means that *setegid()* changes the saved set-group-ID if the effective group ID is set to a value other than the current real group ID. (A similar remark applies for the older implementation of *seteuid()* using *setreuid()*.) Again, this behavior is not specified in SUSv3.

Modifying real and effective IDs

The *setreuid()* system call allows the calling process to independently change the values of its real and effective user IDs. The *setregid()* system call performs the analogous task for the real and effective group IDs.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Both return 0 on success, or −1 on error

The first argument to each of these system calls is the new real ID. The second argument is the new effective ID. If we want to change only one of the identifiers, then we can specify −1 for the other argument.

Originally derived from BSD, *setreuid()* and *setregid()* are now specified in SUSv3 and are available on most UNIX implementations.

As with the other system calls described in this section, rules govern the changes that we can make using *setreuid()* and *setregid()*. We describe these rules from the point of view of *setreuid()*, with the understanding that *setregid()* is similar, except as noted:

1. An unprivileged process can set the real user ID only to the current value of the real (i.e., no change) or effective user ID. The effective user ID can be set only to the current value of the real user ID, effective user ID (i.e., no change), or saved set-user-ID.

SUSv3 says that it is unspecified whether an unprivileged process can use *setreuid()* to change the value of the real user ID to the current value of the real user ID, effective user ID, or saved set-user-ID, and the details of precisely what changes can be made to the real user ID vary across implementations.

SUSv3 describes slightly different behavior for *setregid()*: an unprivileged process can set the real group ID to the current value of the saved set-group-ID or set the effective group ID to the current value of either the real group ID or the saved set-group-ID. Again, the details of precisely what changes can be made vary across implementations.

2. A privileged process can make any changes to the IDs.
3. For both privileged and unprivileged processes, the saved set-user-ID is also set to the same value as the (new) effective user ID if either of the following is true:
   a) *ruid* is not −1 (i.e., the real user ID is being set, even to the same value it already had), or
   b) the effective user ID is being set to a value other than the value of the real user ID prior to the call.

   Put conversely, if a process uses *setreuid()* only to change the effective user ID to the same value as the current real user ID, then the saved set-user-ID is left unchanged, and a later call to *setreuid()* (or *seteuid()*) can restore the effective user ID to the saved set-user-ID value. (SUSv3 doesn't specify the effect of *setreuid()* and *setregid()* on the saved set IDs, but SUSv4 specifies the behavior described here.)

The third rule provides a way for a set-user-ID program to permanently drop its privilege, using a call such as the following:

```
setreuid(getuid(), getuid());
```

A set-user-ID-*root* process that wants to change both its user and group credentials to arbitrary values should first call *setregid()* and then call *setreuid()*. If the calls are made in the opposite order, then the *setregid()* call will fail, because the program will no longer be privileged after the call to *setreuid()*. Similar remarks apply if we are using *setresuid()* and *setresgid()* (described below) for this purpose.

BSD releases up to and including 4.3BSD did not have the saved set-user-ID and saved set-group-ID (which are nowadays mandated by SUSv3). Instead, on BSD, *setreuid()* and *setregid()* permitted a process to drop and regain privilege by swapping the values of the real and effective IDs back and forth. This had the undesirable side effect of changing the real user ID in order to change the effective user ID.

Retrieving real, effective, and saved set IDs

On most UNIX implementations, a process can't directly retrieve (or update) its saved set-user-ID and saved set-group-ID. However, Linux provides two (nonstandard) system calls allowing us to do just that: *getresuid()* and *getresgid()*.

```
#define _GNU_SOURCE
#include <unistd.h>

int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
```

Both return 0 on success, or −1 on error

The *getresuid()* system call returns the current values of the calling process's real user ID, effective user ID, and saved set-user-ID in the locations pointed by its three arguments. The *getresgid()* system call does the same for the corresponding group IDs.

Modifying real, effective, and saved set IDs

The *setresuid()* system call allows the calling process to independently change the values of all three of its user IDs. The new values for each of the user IDs are specified by the three arguments to the system call. The *setresgid()* system call performs the analogous task for the group IDs.

```
#define _GNU_SOURCE
#include <unistd.h>

int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

Both return 0 on success, or −1 on error

If we don't want to change all of the identifiers, then specifying −1 for an argument leaves the corresponding identifier unchanged. For example, the following call is equivalent to *seteuid(x)*:

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

The rules about what changes may be made by *setresuid()* (*setresgid()* is similar) are as follows:

1. An unprivileged process can set any of its real user ID, effective user ID, and saved set-user-ID to any of the values currently in its current real user ID, effective user ID, or saved set-user-ID.
2. A privileged process can make arbitrary changes to its real user ID, effective user ID, and saved set-user-ID.
3. Regardless of whether the call makes any changes to other IDs, the file-system user ID is always set to the same value as the (possibly new) effective user ID.

Calls to *setresuid()* and *setresgid()* have an all-or-nothing effect. Either all of the requested identifiers are successfully changed or none are changed. (The same comment applies with respect to the other system calls described in this chapter that change multiple identifiers.)

Although *setresuid()* and *setresgid()* provide the most straightforward API for changing process credentials, we can't portably employ them in applications; they are not specified in SUSv3 and are available on only a few other UNIX implementations.

9.7.2 Retrieving and Modifying File-System IDs

All of the previously described system calls that change the process's effective user or group ID also always change the corresponding file-system ID. To change the file-system IDs independently of the effective IDs, we must employ two Linux-specific system calls: *setfsuid()* and *setfsgid()*.

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

int setfsuid(uid_t fsuid);
                                        Always returns the previous file-system user ID

int setfsgid(gid_t fsgid);
                                        Always returns the previous file-system group ID
```

The *setfsuid()* system call changes the file-system user ID of a process to the value specified by *fsuid*. The *setfsgid()* system call changes the file system group ID to the value specified by *fsgid*.

Again, there are rules about the kind of changes that can be made. The rules for *setfsgid()* are similar to the rules for *setfsuid()*, which are as follows:

1. An unprivileged process can set the file-system user ID to the current value of the real user ID, effective user ID, file-system user ID (i.e., no change), or saved set-user-ID.
2. A privileged process can set the file-system user ID to any value.

The implementation of these calls is somewhat unpolished. To begin with, there are no corresponding system calls that retrieve the current value of the file-system IDs. In addition, the system calls do no error checking; if an unprivileged process attempts to set its file-system ID to an unacceptable value, the attempt is silently ignored. The return value from each of these system calls is the previous value of the corresponding file-system ID, *whether the call succeeds or fails*. Thus, we do have a way of finding out the current values of the file-system IDs, but only at the same time as we try (either successfully or unsuccessfully) to change them.

Use of the *setfsuid()* and *setfsgid()* system calls is no longer necessary on Linux and should be avoided in applications designed to be ported to other UNIX implementations.

9.7.3 Retrieving and Modifying Supplementary Group IDs

The *getgroups()* system call returns the set of groups of which the calling process is currently a member, in the array pointed to by *grouplist*.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Returns number of group IDs placed in *grouplist* on success, or −1 on error

On Linux, as on most UNIX implementations, *getgroups()* simply returns the calling process's supplementary group IDs. However, SUSv3 also allows an implementation to include the calling process's effective group ID in the returned *grouplist*.

The calling program must allocate the *grouplist* array and specify its length in the argument *gidsetsize*. On successful completion, *getgroups()* returns the number of group IDs placed in *grouplist*.

If the number of groups of which a process is a member exceeds *gidsetsize*, *getgroups()* returns an error (EINVAL). To avoid this possibility, we can size the *grouplist* array to be one greater (to portably allow for the possible inclusion of the effective group ID) than the constant NGROUPS_MAX (defined in <limits.h>), which defines the maximum number of supplementary groups of which a process may be a member. Thus, we could declare *grouplist* as follows:

```
gid_t grouplist[NGROUPS_MAX + 1];
```

In Linux kernels prior to 2.6.4, NGROUPS_MAX has the value 32. From kernel 2.6.4 onward, NGROUPS_MAX has the value 65,536.

An application can also determine the NGROUPS_MAX limit at run time in the following ways:

- Call *sysconf(_SC_NGROUPS_MAX)*. (We explain the use of *sysconf()* in Section 11.2.)
- Read the limit from the read-only, Linux-specific /proc/sys/kernel/ngroups_max file. This file is provided since kernel 2.6.4.

Alternatively, an application can make a call to *getgroups()* specifying *gidtsetsize* as 0. In this case, *grouplist* is not modified, but the return value of the call gives the number of groups of which the process is a member.

The value obtained by any of these run-time techniques can then be used to dynamically allocate a *grouplist* array for a future *getgroups()* call.

A privileged process can use *setgroups()* and *initgroups()* to change its set of supplementary group IDs.

```
#define _BSD_SOURCE
#include <grp.h>

int setgroups(size_t gidsetsize, const gid_t *grouplist);
int initgroups(const char *user, gid_t group);
```

Both return 0 on success, or −1 on error

The *setgroups()* system call replaces the calling process's supplementary group IDs with the set given in the array *grouplist*. The *gidsetsize* argument specifies the number of group IDs in the array argument *grouplist*.

The *initgroups()* function initializes the calling process's supplementary group IDs by scanning `/etc/groups` and building a list of all groups of which the named *user* is a member. In addition, the group ID specified in *group* is also added to the process's set of supplementary group IDs.

The primary users of *initgroups()* are programs that create login sessions—for example, *login(1)*, which sets various process attributes prior to executing the user's login shell. Such programs typically obtain the value to be used for the *group* argument by reading the group ID field from the user's record in the password file. This is slightly confusing, since the group ID from the password file is not really a supplementary group, Instead, it defines the initial real user ID, effective user ID, and saved set-user-ID of the login shell. Nevertheless, this is how *initgroups()* is usually employed.

Although not part of SUSv3, *setgroups()* and *initgroups()* are available on all UNIX implementations.

9.7.4 Summary of Calls for Modifying Process Credentials

Table 9-1 summarizes the effects of the various system calls and library functions used to change process credentials.

Figure 9-1 provides a graphical overview of the same information given in Table 9-1. This diagram shows things from the perspective of the calls that change the user IDs, but the rules for changes to the group IDs are similar.

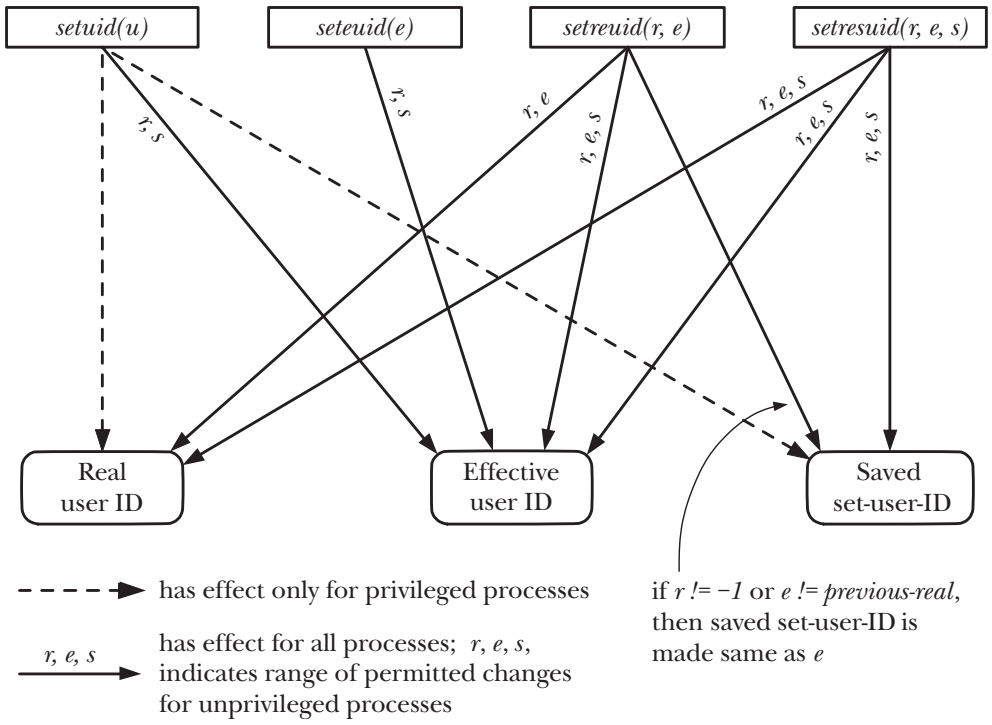

Figure 9-1: Effect of credential-changing functions on process user IDs

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

Interface	Purpose and effect within:		Portability
	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	Specified in SUSv3; BSD derivatives have different semantics
seteuid(e) *setegid(e)*	Change effective ID to the same value as current real or saved set ID	Change effective ID to any value	Specified in SUSv3
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	Specified in SUSv3, but operation varies across implementations
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	Not in SUSv3 and present on few other UNIX implementations
setfsuid(u) *setfsgid(u)*	Change file-system ID to same value as current real, effective, file system, or saved set ID	Change file-system ID to any value	Linux-specific
setgroups(n, l)	Can't be called from an unprivileged process	Set supplementary group IDs to any values	Not in SUSv3, but available on all UNIX implementations

Note the following supplementary information to Table 9-1:

- The *glibc* implementations of *seteuid()* (as *setresuid(−1, e, −1)*) and *setegid()* (as *setregid(−1, e)*) also allow the effective ID to be set to the same value it already has, but this is not specified in SUSv3. The *setegid()* implementation also changes the saved set-group-ID if the effective user ID is set to a value other than that of the current real user ID. (SUSv3 doesn't specify that *setegid()* makes changes to the saved set-group-ID.)
- For calls to *setreuid()* and *setregid()* by both privileged and unprivileged processes, if *r* is not −1, or *e* is specified as a value different from the real ID prior to the call, then the saved set-user-ID or saved set-group-ID is also set to the same value as the (new) effective ID. (SUSv3 doesn't specify that *setreuid()* and *setregid()* make changes to the saved set IDs.)
- Whenever the effective user (group) ID is changed, the Linux-specific file-system user (group) ID is changed to the same value.
- Calls to *setresuid()* always modify the file-system user ID to have the same value as the effective user ID, regardless of whether the effective user ID is changed by the call. Calls to *setresgid()* have an analogous effect on the file-system group ID.

9.7.5 Example: Displaying Process Credentials

The program in Listing 9-1 uses the system calls and library functions described in the preceding pages to retrieve all of the process's user and group IDs, and then displays them.

Listing 9-1: Display all process user and group IDs

proccred/idshow.c

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/fsuid.h>
#include <limits.h>
#include "ugid_functions.h"   /* userNameFromId() & groupNameFromId() */
#include "tlpi_hdr.h"

#define SG_SIZE (NGROUPS_MAX + 1)

int
main(int argc, char *argv[])
{
    uid_t ruid, euid, suid, fsuid;
    gid_t rgid, egid, sgid, fsgid;
    gid_t suppGroups[SG_SIZE];
    int numGroups, j;
    char *p;

    if (getresuid(&ruid, &euid, &suid) == -1)
        errExit("getresuid");
    if (getresgid(&rgid, &egid, &sgid) == -1)
        errExit("getresgid");

    /* Attempts to change the file-system IDs are always ignored
       for unprivileged processes, but even so, the following
       calls return the current file-system IDs */

    fsuid = setfsuid(0);
    fsgid = setfsgid(0);

    printf("UID: ");
    p = userNameFromId(ruid);
    printf("real=%s (%ld); ", (p == NULL) ? "???" : p, (long) ruid);
    p = userNameFromId(euid);
    printf("eff=%s (%ld); ", (p == NULL) ? "???" : p, (long) euid);
    p = userNameFromId(suid);
    printf("saved=%s (%ld); ", (p == NULL) ? "???" : p, (long) suid);
    p = userNameFromId(fsuid);
    printf("fs=%s (%ld); ", (p == NULL) ? "???" : p, (long) fsuid);
    printf("\n");

    printf("GID: ");
    p = groupNameFromId(rgid);
    printf("real=%s (%ld); ", (p == NULL) ? "???" : p, (long) rgid);
```

```
    p = groupNameFromId(egid);
    printf("eff=%s (%ld); ", (p == NULL) ? "???" : p, (long) egid);
    p = groupNameFromId(sgid);
    printf("saved=%s (%ld); ", (p == NULL) ? "???" : p, (long) sgid);
    p = groupNameFromId(fsgid);
    printf("fs=%s (%ld); ", (p == NULL) ? "???" : p, (long) fsgid);
    printf("\n");

    numGroups = getgroups(SG_SIZE, suppGroups);
    if (numGroups == -1)
        errExit("getgroups");

    printf("Supplementary groups (%d): ", numGroups);
    for (j = 0; j < numGroups; j++) {
        p = groupNameFromId(suppGroups[j]);
        printf("%s (%ld) ", (p == NULL) ? "???" : p, (long) suppGroups[j]);
    }
    printf("\n");

    exit(EXIT_SUCCESS);
}
```

proccred/idshow.c

9.8 Summary

Each process has a number of associated user and group IDs (credentials). The real IDs define the ownership of the process. On most UNIX implementations, the effective IDs are used to determine a process's permissions when accessing resources such as files. On Linux, however, the file-system IDs are used for determining permissions for accessing files, while the effective IDs are used for other permission checks. (Because the file-system IDs normally have the same values as the corresponding effective IDs, Linux behaves in the same way as other UNIX implementations when checking file permissions.) A process's supplementary group IDs are a further set of groups of which the process is considered to be a member for the purpose of permission checking. Various system calls and library functions allow a process to retrieve and change its user and group IDs.

When a set-user-ID program is run, the effective user ID of the process is set to that of the owner of the file. This mechanism allows a user to assume the identity, and thus the privileges, of another user while running a particular program. Correspondingly, set-group-ID programs change the effective group ID of the process running a program. The saved set-user-ID and saved set-group-ID allow set-user-ID and set-group-ID programs to temporarily drop and then later reassume privileges.

The user ID 0 is special. Normally, a single user account, named *root*, has this user ID. Processes with an effective user ID of 0 are privileged—that is, they are exempt from many of the permission checks normally performed when a process makes various system calls (such as those used to arbitrarily change the various process user and group IDs).

9.9 Exercises

9-1. Assume in each of the following cases that the initial set of process user IDs is *real=1000 effective=0 saved=0 file-system=0*. What would be the state of the user IDs after the following calls?

a) *setuid(2000);*

b) *setreuid(−1, 2000);*

c) *seteuid(2000);*

d) *setfsuid(2000);*

e) *setresuid(−1, 2000, 3000);*

9-2. Is a process with the following user IDs privileged? Explain your answer.

```
real=0 effective=1000 saved=1000 file-system=1000
```

9-3. Implement *initgroups()* using *setgroups()* and library functions for retrieving information from the password and group files (Section 8.4). Remember that a process must be privileged in order to be able to call *setgroups()*.

9-4. If a process whose user IDs all have the value *X* executes a set-user-ID program whose user ID, *Y*, is nonzero, then the process credentials are set as follows:

```
real=X effective=Y saved=Y
```

(We ignore the file-system user ID, since it tracks the effective user ID.) Show the *setuid()*, *seteuid()*, *setreuid()*, and *setresuid()* calls, respectively, that would be used to perform the following operations:

a) Suspend and resume the set-user-ID identity (i.e., switch the effective user ID to the value of the real user ID and then back to the saved set-user-ID).

b) Permanently drop the set-user-ID identity (i.e., ensure that the effective user ID and the saved set-user-ID are set to the value of the real user ID).

(This exercise also requires the use of *getuid()* and *geteuid()* to retrieve the process's real and effective user IDs.) Note that for certain of the system calls listed above, some of these operations can't be performed.

9-5. Repeat the previous exercise for a process executing a set-user-ID-*root* program, which has the following initial set of process credentials:

```
real=X effective=0 saved=0
```