

CSIS0230A Principles of Operating Systems(Class A)
Notes for Tutorial 12
Unix Security

In this tutorial, we will experiment with the user protection mechanisms provided by Linux, which is very similar to that of other Unix flavors.

1. UID and GID

When a user logs in, his username is mapped into a numeric UID and GID (as well as other information like the location of the home directory) using the `/etc/passwd` file¹. Some more group IDs, as specified by `/etc/group`, may be given to the user as well. These numbers form the user **domain** which restricts the resources that the process can gain access to. The user with UID 0, traditionally called `root`, has a special role: he can side-step most security checks and is used to administer the system. From the point of view of the OS, processes are the only things that can request for resources, in particular filesystem objects (like files, directories, devices, etc), System V Interprocess Communication (IPC) objects, and network ports.

2. Process domain

The domain of each process is defined by a set of IDs, including:

- RUID, RGID—“real” UID and GID. These are typically used to store the ID of the user who starts the program. A few system calls (most notably `signal`) check for these ids.
- EUID, EGID—“effective” UID and GID. UID/GID is used for most privilege checks (except for the filesystem). Normally they are either RUID/RGID or SUID/SGID.
- FSUID, FSGID²—“filesystem” UID and GID. They are similar to EUID and EGID, except that they are used for privilege checks for the filesystem. Most of the time EUID and FSUID are the same, and EGID and FSGID are the same.
- SUID, SGID³—“saved” UID and GID. It has no significance in privileges checking, and is only used to hold an old ID that the program would like to switch to some time later. Typically it stores the UID or GID of the owner of the `setuid` or `setgid` program currently being executed.

These IDs are stored in the PCB of each process, i.e., their `task_struct`. A program can get the real and effective IDs using `getuid()`, `geteuid()`, `getgid()` and `getegid()` functions. For example:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main() {
    printf("RUID is %d. EUID is %d.\n", getuid(), geteuid());
    return 0;
}
```

¹There are also some “system users” which have a UID/GID. E.g., most Linux system has a user called `man` which owns all the manual page files.

²FSUID and FSGID are Linux-specific.

³Introduced by System V, and later added into BSD systems

Only `root` can change these IDs arbitrarily. A process running as a normal user can only change these IDs to one of the IDs it already has.

3. Filesystem Object Attributes

The Unix filesystems store some attributes for each file stored. They are checked when a process open a file for reading or writing, or creating, renaming or unlinking a directory entry.

- Owner UID and GID - identifies the “owner” of the filesystem object. Only the owner or `root` can change the access control attributes.
- *Read, write, execute* bits for each of user (owner), group, and other. For ordinary files, read, write, and execute have their typical meanings. For directories, the “read” permission is needed to list a directory. The “execute” permission is sometimes called “search” permission, and is needed to actually change to the directory or to use its entries. The “write” permission permits adding, unlinking (i.e., removing) and renaming files in the directory. The permission values of symbolic links are never used; it’s only the values of their containing directories and the linked-to file that matter.
- *Setuid, setgid* bits. They can be used for executable files to switch privileges (see below). The *setgid* bit is also be used for directories: when it is set, new files created in the directory inherit the group id the parent directory instead of the creator’s group id. Linux (indeed, System V) also use this bit for non-executable files, which requests for mandatory file locking (see *Documentation/mandatory.txt* of the kernel directory for information).
- *Sticky* bit. This bit is historically used to hint the filesystem to keep executable files in the memory, but is nowadays with modern VM systems this is not needed any more. Instead, sticky bit is used only in directory. When it is set, non-owners of the directory cannot rename or unlink files that are not owned by himself. This is generally used in public directories like `/tmp`.

Apart from the UIDs (which can be manipulated by the `chown` command or the `chown()` system call if you are the owner of the file), all the above are represented in a numeric property of the file called the “file permission”. The read, write and execute bits are the bottom 9 bits of the file permission. The sticky, setuid and setgid bits are bit 9, 10 and 11 respectively. To change the permission of the file object, you can use the `chmod` command (or `chmod()` or `fchmod()` system call). For more information, please refer to man page.

4. Switching Privileges

It is possible to switch to a different UID if you are `root`. But what if you are not? The answer is that it is also possible, if you agrees to run a specific program owned by the target UID, and if that owner agrees to let you switch to it. These are all done by the setuid and setgid mechanism.

If you run a program (i.e., `execve()` an executable file) with the setuid bit turned on, the EUID of the process is set to that of the owner of the executable file. The setuid bit is represented with an ‘s’ when displaying the rights with the `ls` command:

```
> ls -l /bin/su
-rwsr-xr-x 1 root root 14124 Aug 18 1999 /bin/su
```

Here, when a process of a normal user runs this program, the EUID and FSUID are set to that

of `root` (i.e., 0). Needless to say, the owner of the executable file (e.g., `root` here) must think thrice and make sure the program only deliver the capability that he wants the user to be able to do. (For the `su` program, it is that the user should be able to run a shell with the UID of another user if he can produce the password of that user.) Since bugs in programs can easily allow users to obtain further capabilities, such programs must be written very carefully.

Similar to `setuid`, the `setgid` bit asks the kernel to use the file owner's GID, instead of the user's GID, as the EGID and FSGID of the process. This is usually done to grant access to a file or device only via specific programs, by granting access of a file or device to a group accessible only through a `setgid` program.

5. An Example

In this section we run an example to demonstrate the use of the `setuid` bit. First, login as `root`. Compile the program list in Section 2 by the following command:

```
gcc -o testuid testuid.c
```

Then, use the `ls -l` command to see the attributes of the compile program:

```
-rwxr-xr-x 1 root root 13980 Nov 19 16:33 testuid*
```

Now copy the file to a directory which can be access by all other users, e.g. `/usr/local/bin`, by the following command:

```
cp testuid /usr/local/bin
```

After that, run this program by `root`, you will found the following output:

```
RUID is 0. EUID is 0.
```

Try to run the same program using normal user, you will get something similar to the following:

```
RUID is 501. EUID is 501.
```

Then, use `root` to set the program's `setuid` bit by the following command:

```
chmod u+s /usr/local/bin/testuid
```

The attributes of the file will change:

```
-rwsr-xr-x 1 root root 13980 Nov 19 16:33 testuid*
```

Then, run the same program using normal user, you will get the following output:

```
RUID is 501. EUID is 0.
```

You HAVE ROOT ACCESS!!!!

6. Manipulating IDs within programs

There are two types of programs with special privilege: those like `login` are created by the `init` process (the process created during system startup) which has `root` privileges, and those `setuid` and `setgid` programs like the above which has the privileges of the owner of the program apart from the one executing the program. Typically, these programs need to drop their special

privilege after performing some specific tasks. The second type of programs sometimes need to regain their privilege later. Using the Linux-specific *setresuid()* and *setresgid()* system calls, the user id's can be arbitrarily swapped. However, if you want to write it in a more portable way, you should use some API which are well established. Unluckily, the API are quite difficult to understand due to historical reasons.

The original API consists of the *setuid()* and *setgid()* system calls, which are designed to allow programs like `login` to set the process domain after authentication. Originally it works only if the original user is `root`. When *setuid/setgid* programs are introduced (at that time there is no SUID or SGID), this call has been extended for dropping the privilege of such programs. So normal users can call the functions, but only the **effective** user-ID is set (rather than all the IDs). Such usages are not recommended for new functions, however, because it behaves differently depending on whether the program is *setuid root* or *setuid* to other users. Instead, one should use the *seteuid()* and *setegid()* functions, which never change the “real” (and “saved”) IDs.

The story doesn't end here, unluckily. Some programs which drops their privilege needs to regain its access later. The original answer given by BSD is the *setreuid()* and *setregid()* functions, which allows non-`root` users to set both the real and effective UID/GID at the same time. So an SUID program can now **swap** the RUID and the EUID using *setreuid()*, so that the normal privilege used is the one executing the program, while the RUID retains the ability to switch back to the owner of the *setuid* program if need arise. If a program wants to give up even that ability, it can call *setregid()* setting **both** the RUID and EUID to the UID of the one executing the program. This is not a really satisfactory solution, though. It makes it difficult for system administrators to know whether a program is executed by A *setuid*'ed to B, or executed by B *setuid*'ed to A. New programs should no longer use this interface.

To address the short-coming of the above scheme, the SUID and SGID are introduced. Now programs which use *seteuid()* and *setegid()* can switch back and forth between the the IDs of the owner and the real user of the program, and the user executing the program remains as the real ID. No new system call is introduced. Then how a program can make sure that it can never regain the power of its owner? This can be achieved using *setreuid()* and *setregid()*. If one set both the effective and real uid (or gid) to the same value, then the saved ID is also set to the same value, thus giving up capability to switch IDs further. (The actual rule is somewhat more complicated, see the man page of *setreuid()*).

As a final note, in Linux one can change the FSUID and FSGID using the *setfsuid()* and *setfsgid()* system calls, which are defined in `<unistd.h>` or `<sys/fsuid.h>` depending on the version of the C library you use.

7. References

<http://www.linuxfocus.org/English/January2001/article182.shtml>,
<http://www.theorygroup.com/Theory/FAQ/Secure-Programs-HOWTO-3.html>