DAN TSAFRIR, DILMA DA SILVA, AND DAVID WAGNER

# the murky issue of changing process identity: revising "setuid demystified"

Dan Tsafrir is a postdoctoral researcher at the IBM T.J. Watson Research Center in New York. He is a member of the advanced operating systems group and is interested in various aspects of operating systems.

*dants@us.ibm.com*

Dilma da Silva is a researcher at the IBM T.J. Watson Research Center in New York. She manages the Advanced Operating Systems group. Prior to joining IBM, she was an Assistant Professor at University of Sao Paulo, Brazil. Her research in operating systems addresses the need for scalable and customizable system software.

*dilmasilva@us.ibm.com*

David Wagner is a professor in the computer science division at the University of California at Berkeley. He studies computer security, cryptography, and electronic voting.

*daw@cs.berkeley.edu*

DROPPING UNNEEDED PROCESS PRIVILEGES promotes security but is notoriously error-prone because of confusing set*id system calls with unclear semantics and subtle portability issues. To make things worse, existing recipes to accomplish the task are lacking, related manuals can be misleading, and the associated kernel subsystem might contain bugs. We therefore proclaim the system as untrustworthy when it comes to the subject matter, and we suggest a defensive, easy-to-use solution that addresses all concerns.

Whenever you run a program, it assumes your identity and you lend it all your power: Whatever you're allowed to do, it too is allowed. This includes deleting your files, killing your other programs, changing your password, and retrieving your mail, for example. Occasionally, you need to write programs that enhance the power of others. Consider, for example, a Mahjongg game that maintains a high-score file. Of course, making the file writeable by all is not a very good idea if you want to ensure that no one cheats, so Mahjongg must somehow convey to players the ability to update the file in a controlled manner. In UNIX systems this is done as follows: When a game ends, if the score is high enough, Mahjongg temporarily assumes the identity of the file's owner, makes the appropriate modifications, and switches back to the identity of the original player.

Many standard utilities work this way, including passwd and chsh (which update /etc/passwd), xterm (which updates utmp usage information), su (which changes user), sudo (which acts as root), and X (which accesses interactive devices). The common feature of these tools is that they know their real identity is of a nonprivileged user, but they have the ability to assume a privileged identity when required. (Note that "privileged" doesn't necessarily mean root; it merely means some other identity that has the power to do what the real user can't.) Such executables are collectively referred as "setuid programs," because (1) they must be explicitly associated with a "setuid bit" (through the `chmod` command) and (2) they pull off the identity juggling trick through the use of set*id system calls (`setuid(2)`, `setreuid(2)`, and all their friends).

There's another, often overlooked, type of program that can do identity juggling but does *not* have an associated setuid bit. These start off as root processes and use set*id system calls to change their identity to that of an ordinary nonprivileged user. Examples include the login program, the cron daemon (which runs user tasks at a specified time), daemons providing service to remote users by assuming their identity (sshd, telnetd, nfs, etc.), and various mail server components.

Both types of programs share a similar philosophy: To reduce the chances of their extra powers being abused, they attempt to obey the principle of least privilege, which states that "every program and every user of the system should operate using the least set of privileges necessary to complete the job" [16]. For setuid programs this translates to:

1. minimizing the number and duration of the time periods at which the program temporarily assumes the privileged identity, to reduce the negative effect that programming mistakes might have (e.g., mistakenly removing a file as root can have far greater negative implications than doing it when the nonprivileged identity is in effect), and
2. permanently giving up the ability to assume the privileged identity as soon as it's no longer needed, so that if an attacker gains control (e.g., through a buffer overflow vulnerability), the attacker can't exploit those privileges.

The principle of least privilege is a simple and sensible rule. But when it comes to identity-changing programs (in the immortal words of The Essex [7] or anybody who ever tried to lose weight [14]) it's easier said than done. Here are a few quotes that may explain why it's at least as hard as doing a diet: Chen et al. said that "for historical reasons, the uid-setting system calls are poorly designed, insufficiently documented, and widely misunderstood" and that the associated manuals "are often incomplete or even wrong" [2]. Dean and Hu observed that "the setuid family of system calls is its own rat's nest; on different UNIX and UNIX-like systems, system calls of the same name and arguments can have different semantics, including the possibility of silent failures" [3]. Torek and Dik concluded that "many years after the inception of setuid programs, how to write them is still not well understood by the majority of people who write them" [17]. All these deficiencies have made the setuid mechanism the source of many security vulnerabilities.

It has been more than 30 years since Dennis Ritchie introduced the setuid mechanism [15] and more than 20 years since people started publishing papers about how to correctly write setuid programs [1]. The fact that this article has something new to say serves as an unfortunate testament that the topic is not yet resolved. Our goal in this paper is to provide the equivalent of a magical diet pill that effortlessly makes you slim (or at least lays the foundation for this magic). Specifically, we design and implement an intuitive change-identity algorithm that abstracts away the many pitfalls, confusing details, operating-system-specific behavior, and portability issues. We build on and extend the algorithm proposed by Chen et al. [2], which neglected to factor in the role that supplementary groups play in forming an identity. Our code is publicly available [18]. It was extensively tested on Linux 2.6.22, FreeBSD 7.0-STABLE, OpenSolaris, and AIX 5.3. We warn that, given the history of subtle pitfalls in the set*id syscalls, it may be prudent for developers to avoid relying upon our algorithm until it has been subject to careful review by others.

## User Identity vs. Process Identity

Before attempting to securely switch identities, we need to define what the term "identity" means. In this context, we found it productive to make a distinction between two types of identities: that of a user and that of a process. The user's credentials include the user ID (uid), the user's primary group (gid), and an additional array of supplementary groups (sups). Collectively, they determine which system resources the user can access. In particular, a zero uid is associated with the superuser (root) who can access all resources. We define the `ucred_t` type to represent a user by aggregating these three fields, as follows:

```
typedef struct supplementary_groups {
   gid_t *list;     // sorted ascending, no duplicates
   int       size;   // number of entries in 'list'
} sups_t;

typedef struct user_credentials {
   uid_t    uid;
   gid_t    gid;
   sups_t sups;
} ucred_t;
```

Things are a bit more complicated when it comes to the corresponding process credentials. Each process has three user IDs: real (ruid), effective (euid), and saved (suid). The real uid identifies the "owner" of the process, which is typically the executable's invoker. The effective uid represents the identity in effect, namely, the one used by the OS (operating system) for most access decisions. The saved uid stores some previous user ID, so that it can be restored (copied to the euid) at some later time with the help of set*uid system calls. Similarly, a process has three group IDs: rgid, egid, and sgid. We define the `pcred_t` type to encapsulate the credentials of a process:

```
typedef struct user_ids     { uid_t  r, e, s; }     uids_t;
typedef struct group_ids   { gid_t  r, e, s; }     gids_t;

typedef struct process_credentials {
   uids_t  uids;        // uids.r = ruid, uids.e  = euid, uids.s  = suid
   gids_t  gids;        // gids.r  = rgid, gids.e  = egid, gids.s  = sgid
   sups_t sups;
} pcred_t;
```

Supplementary groups can be queried with the help of the `getgroups` system call. The ruid, euid, rgid, and egid of a process can be retrieved with `getuid`, `geteuid`, `getgid`, and `getegid`, respectively. The ways to find out the values of suid and sgid are OS-specific.

In Linux, each process also has an fsuid and an fsgid, which are used for access control to the file system. Normally, these are equal to the euid and egid, respectively, unless they are explicitly changed [11]. As this rarely used feature is Linux-specific, it is not included in the aforementioned data structures. To ensure correctness, our algorithm never manipulates the fsuid or fsgid, ensuring that (if programs rely only upon our interface for manipulating privileges) the fsuid and fsgid will always match the euid and egid.

The benefit of differentiating between user and process identities is that the former is more convenient to work with, easier to understand, better captures the perception of programmers regarding identity, and typically is all that is needed for programmers to specify what kind of an identity they require. In other words, the notions of real, effective, and saved IDs are not important in their own right; rather, they are simply the technical means by which identity change is made possible. Note, however, that "user" isn't an abstraction that is represented by any kernel primitive: The kernel doesn't deal with users; it deals with processes. It is therefore the job of our algorithm to internally use `pcred_t` and provide the appropriate mappings.

## Rules of Identity Juggling

### IDENTITY PROPAGATION AND SPLIT PERSONALITIES

The second thing one has to consider when attempting to correctly switch identities is the manner by which processes initially get their identity. When a user *rik* logs in, the login program forks a process *P* and sets things up such that (1) *P*'s three uids hold *rik*'s uid, (2) *P*'s three gids hold *rik*'s primary group, and (3) *P*'s supplementary array is populated with the gids of the groups to which *rik* belongs. The process credentials are then inherited across `fork`. They are also inherited across `exec`, unless the corresponding executable *E* has its setuid bit set, in which case the effective and saved uids are set to be that of *E*'s owner (but the real uid remains unchanged). Likewise, if *E* is setgid, then the saved and effective groups of the new process are assigned with *E*'s group.

Conversely, the supplementary array is *always* inherited as is, even if *E*'s setuid/setgid bits are set. Notice that this can lead to a bizarre situation where *E* is running with a split personality: The effective user and group are of *E*'s owner, whereas the supplementary groups are of *E*'s invoker. This isn't necessarily bad (and in fact constitutes the typical case), but it's important to understand that this is what goes on.

### USER ID JUGGLING

Since access control is based on the effective user ID, a process gains privilege by assigning a privileged user ID to its euid, and drops privilege by removing it. To drop privilege temporarily, a process removes the privileged user ID from its euid but stores it in its saved ID; later, the process may restore privilege by copying this value back to the euid. To drop privilege permanently, a process removes the privileged user ID from all three uids. Thereafter, the process can never restore privilege.

Roughly speaking, there typically exists some technical way for a process to copy the value from one of its three uids to another, and thus perform the uid juggling as was just described. If the process is nonroot (uid $\neq 0$), then that's all it can do (juggle back and forth between the real and saved uids). Root, however, can assume any identity.

### PRIMARY GROUP JUGGLING

The rules of changing gids are identical, with the exception that egid=0 doesn't convey any special privileges: Only if euid=0 can the process set arbitrary gids.

### SUPPLEMENTARY GROUPS JUGGLING

The rules for changing supplementary groups are much simpler: If a process has euid=0, it can change them however it likes through the `setgroups` system call. Otherwise, the process is forbidden from using `setgroups` and is stuck with the current setting. The implications for setuid programs are interesting. If the setuid program drops privileges (assuming the identity of its invoker), then the supplementary groups will already be set appropriately. However, until that happens, the program will have a split personality. A setuid-root program can set the supplementary groups to match its privileged identity, if it chooses. However, nonroot setuid programs cannot: They will suffer from a split personality for as long as they maintain their privileged identity, and there's simply no way around it. As a result, nonroot setuid programs might run with extra privileges that their creators did not anticipate.

## MESSINESS OF SETUID SYSTEM CALLS

Several standard set*id system calls allow programmers to manipulate the real, effective, and saved IDs, in various ways. To demonstrate their problematic semantics, we focus on only `setuid(2)` through an example of a vulnerability found in a mainstream program. Googling the word "setuid" with "vulnerability" or "bug" immediately brings up many examples that are suitable for this purpose. But to also demonstrate the prevalence of the problem, we attempted to find a new vulnerability. Indeed, the first program we examined contained one.

Exim is a popular mail server that is used by default in many systems [5]. Figure 1 shows the function exim uses to drop privileges permanently, taken from the latest version available at the time of this writing [6]. It implicitly assumes that calling `setuid` will update all three uids, so that all privileges are permanently relinquished. This assumption indeed holds for some OSes (e.g., FreeBSD). But if the effective ID is nonzero (which may be the case according to the associated documentation) then the assumption doesn't hold for Linux, Solaris, and AIX, as the semantics of `setuid` under these circumstances dictate that only the euid will be updated, leaving the ruid and suid unchanged. Consequently, if exim is compromised, the attacker can restore exim's special privileges and, for example, obtain uncontrolled access to all mail in the system.

Although this particular vulnerability isn't nearly as dangerous as some previously discovered setuid bugs, it does successfully highlight the problematic system call behavior, which differs not only between OSes but also according to the current identity.

```
/*
 * This function sets a new uid and gid permanently, optionally calling
 * initgroups() to set auxiliary groups. There are some special cases when
 * running Exim in unprivileged modes. In these situations the effective
 * uid will not be root; [...]
 */
void exim_setugid(uid_t uid, gid_t gid, BOOL igflag, uschar *msg)
{
   uid_t euid  =   geteuid();
   gid_t egid  =   getegid();

   if (euid == root_uid || euid != uid || egid != gid || igflag) {

      if (igflag) {
         /* do some supplementary groups handling here */ ...
      }

      if (setgid(gid) < 0 || setuid(uid) < 0) {
         /* PANIC! */ ...
      }
   }
}
```

**FIGURE 1: EXIM'S CODE TO PERMANENTLY CHANGE IDENTITY CONTAINS A VULNERABILITY.**

## Safely Dropping Privileges

Equipped with a good understanding of the subject, we go on to develop an algorithm to safely drop privileges permanently. We do so in a top-down manner, making use of the ucred_t and pcred_t types previously defined. Figure 2 (facing page) shows the algorithm. Its input parameter specifies the target identity; the algorithm guarantees to permanently switch to the target identity or clearly indicate failure. The algorithm works by first changing the supplementary groups, then changing the gids and changing the uids (in that order), and, finally, checking that the current identity matches the target identity.

### ERROR HANDLING

There are two ways to indicate failure, depending on how the macros DO_CHK and DO_SYS are defined:

```
#ifdef LIVING_ON_THE_EDGE
#   define DO_SYS(call)    if( (call) == -1  )  return -1     /* do system call     */
#   define DO_CHK(expr)    if( ! (expr)      )  return -1     /* do boolean check */
#else
#   define DO_SYS(call)    if( (call) == -1  )  abort()       /* do system call     */
#   define DO_CHK(expr)    if( ! (expr       )  abort()       /* do boolean check */
#endif
```

But although reporting failure through return values is possible, we advise against it, as it might leave the identity in an inconsistent state. Thus, when an identity change fails in the middle, programmers should either abort or really know what they're doing.

### INPUT CHECK

The ucred_is_sane function checks the validity of the input parameter. It is implemented as follows:

```
long nm = sysconf(_SC_NGROUPS_MAX);
return (nm >= 0) && (nm >= uc->sups.size) && (uc->sups.size >= 0) &&
       uc->uid != (uid_t)-1 &&
       uc->gid != (gid_t)-1;
```

The maximal size of the supplementary groups may differ between systems, but it can be queried in a standard way. We also check that the user and group IDs aren't -1, because this has special meaning for several set*id system calls ("ignore").

### VERIFICATION

The first chunk of code in Figure 2 is responsible for setting the supplementary groups to uc->sups, the three gids to g, and the three uids to u. Setting the uids last is important, because afterward the process might lose its privilege to change its groups. Setting supplementary groups before primary groups is also important, for reasons to become clear later on. The remainder of the function verifies that all of these operations successfully changed our credentials to the desired identity. This policy is required in order to prevent mistakes in the face of the poorly designed set*id interface (e.g., this policy would have prevented the exim vulnerability), to protect against possible

```
int drop_privileges_permanently(const ucred_t *uc /*target identity*/)
{
   uid_t u    =   uc->uid;
   gid_t g    =   uc->gid;
   pcred_t   pc;

   DO_CHK(  ucred_is_sane(uc)                                   );
   DO_SYS(  set_sups( &uc->sups )                               );
   DO_SYS(  set_gids(  g/*real*/, g/*effective*/, g/*saved*/ )  );
   DO_SYS(  set_uids(  u/*real*/, u/*effective*/, u/*saved*/ )  );

   DO_SYS(  get_pcred(  &pc )                                              );
   DO_CHK(  eql_sups  (  &pc.sups , &uc->sups )                            );
   DO_CHK(  g == pc.gids.r   && g == pc.gids.e   && g == pc.gids.s  );
   DO_CHK(  u == pc.uids.r   && u == pc.uids.e   && u == pc.uids.s  );
   free( pc.sups.list );

#if defined(__linux__)
   DO_SYS(  get_fs_ids( &u, &g)                    );
   DO_CHK(  u == uc->uid && g == uc->gid );
#endif

   return 0;    /* success */
}
```

**FIGURE 2: PERMANENTLY SWITCHING IDENTITY AND VERIFYING THE CORRECTNESS OF THE SWITCH.**

related kernel bugs [2] or noncompliant behavior (see below) and to defend against possible future kernel changes. These reasons, combined with the fact that having the correct identity is crucial in terms of security, provide good motivation for our untrusting approach.

## QUERYING PROCESS IDENTITY

The `get_pcred` function we implement fills the memory pointed to by the `pcred_t` pointer it gets. We get the ruid, rgid, euid, and egid with the help of the standard system calls `getuid`, `getgid`, `geteuid`, and `getegid`, respectively. Unfortunately, there's no standard way to retrieve saved IDs, so we use whatever facility the OS makes available, as shown in Figure 3 on the next page. The `getresuid` and `getresgid` nonstandard system calls are the easiest to use and the most popular among OSes. AIX's `getuidx` and `getgidx` also have easy semantics, whereas with Solaris the programmer must resort to using Solaris's /proc interface [10].

The supplementary groups are retrieved with the help of the standard `getgroups` system call. To allow for easy comparison of supplementary arrays, we normalize the array by sorting it and by removing duplicate entries, if any exist. The array is `malloc`ed, and it should therefore be `free`d later on.

## LINUX FILESYSTEM IDS

In Linux, the fsuid is supposed to mirror the euid, as long as `setfsuid` isn't explicitly used [11], and the same goes for fsgid and egid. However, there has been at least one kernel bug that violated this invariant [2]. Therefore, in accordance with our defensive approach, the algorithm in Figure 2 explicitly

```
int get_saved_ids(uid_t *suid, gid_t *sgid)
{
#if defined(__linux__)       ||defined(__HPUX__)       ||\
    defined(__FreeBSD__) ||defined(__OpenBSD__) ||defined(__DragonFly__)
    uid_t ruid, euid;
    gid_t rgid, egid;
    DO_SYS( getresuid(&ruid, &euid, suid) );
    DO_SYS( getresgid(&rgid, &egid, sgid) );

#elif defined(_AIX)
    DO_SYS( *suid = getuidx(ID_SAVED) );
    DO_SYS( *sgid = getgidx(ID_SAVED) );

#elif defined(__sun__) || defined(__sun)
    prcred_t p;   /* prcred_t is defined by Solaris */
    int fd;
    DO_SYS( fd = open( "/proc/self/cred", O_RDONLY)   );
    DO_CHK( read(fd, &p, sizeof(p)) == sizeof(p)       );
    DO_SYS( close(fd)                                  );
    *suid = p.pr_suid;
    *sgid = p.pr_sgid;

#else
#   error "need to implement, notably: __NetBSD__, __APPLE__, __CYGWIN__"

#endif
    return 0;
}
```

**FIGURE 3: GETTING THE SAVED UID AND GID IS AN OS-DEPENDENT OPERATION.**

verifies that the fs-invariant indeed holds. As there is no getfsuid or getfsgid, our implementation of get_fs_ids is the C equivalent of

```
grep Uid /proc/self/status | awk '{print $5}'   # prints fsuid
grep Gid /proc/self/status | awk '{print $5}'   # prints fsgid
```

## SETTING UIDS AND GIDS

The POSIX-standard interfaces for setting IDs are tricky, OS-dependent, and offer no way to directly set the saved IDs. Consequently, *nonstandard* interfaces are preferable, if they offer superior semantics. This is the design principle underlying our implementation of set_uids and set_gids. The implementation is similar in spirit to the code in Figure 3, but it is complicated by the fact that nonprivileged processes are sometimes not allowed to use the preferable interface, in which case we fall back on whatever is available.

Specifically, all OSes that support getresuid (see Figure 3) also support setresuid and setresgid. These offer the clearest and most consistent semantics and can be used by privileged and nonprivileged processes alike. (Of course the usual restrictions for nonprivileged processes still apply, namely, each of the three parameters must be equal to one of the three IDs of the process.) In Solaris, only root can use the /proc interface for setting IDs [10], so with nonroot processes we naively use seteuid and setreuid (and their gid counterparts) and hope for the best: The verification part in Figure 2 will catch any discrepancies. In AIX, setuidx and setgidx are the clearest and most expressive, and they can be used by both root and nonroot processes [13]. However, AIX is very restrictive: a nonroot process can only change its effective IDs, so dropping privileges permanently is impossible for nonroot processes; also, root processes are allowed to set euid, euid/ruid, or euid/ruid/suid, but only to the same value.

## SUPPLEMENTARY GROUPS CAVEATS

Recall that nonroot processes are not allowed to call setgroups. Therefore, to avoid unnecessary failure, setgroups is only invoked if the current and target supplementary sets are unequal, as shown in Figure 4. (Disregard the FreeBSD chunk of code for the moment.) Additionally, recall that after setting the supplementary groups in Figure 2, we verify that this succeeded by querying the current set of supplementary groups and checking that it matches the desired value. In both cases the current and target supplementary sets must be compared. But, unfortunately, this isn't as easy as one would expect.

```
int set_sups(const sups_t *target_sups)
{
    sups_t targetsups = *target_sups;

#ifdef __FreeBSD__
    gid_t arr[ targetsups.size + 1 ];
    memcpy(arr+1, targetsups.list, targetsups.size * sizeof(gid_t) );
    targetsups.size       =   targetsups.size + 1;
    targetsups.list       =   arr;
    targetsups.list[0]    =   getegid();
#endif

    if( geteuid() == 0 ) { // allowed to setgroups, let's not take any chances
        DO_SYS( setgroups(targetsups.size, targetsups.list) );
    }
    else {
        sups_t cursups;
        DO_SYS( get_sups( &cursups) );
        if( ! eql_sups( &cursups, &targetsups) ) // this will probably fail... :(
            DO_SYS( setgroups(targetsups.size, targetsups.list) );
        free( cursups.list );
    }

    return 0;
}
```

**FIGURE 4: SETTING SUPPLEMENTARY GROUPS, WHILE TRYING TO AVOID FAILURE OF NONROOT PROCESSES, AND ACCOMMODATING NONCOMPLIANT BEHAVIOR OF FREEBSD.**

The POSIX standard specifies that "it is implementation-defined whether getgroups also returns the effective group ID in the grouplist array" [9]. This seemingly harmless statement means that if the egid is in fact found in the list returned by getgroups, there's no way to tell whether this group is actually a member of the supplementary group list. In particular, there is no reliable, portable way to get the current list of supplementary groups. As a result, our code for comparing the current and target supplementary sets (see eql_sups in Figure 5, which is used in Figure 2 and Figure 4) assumes that they match even if the current supplementary set contains the egid and the target supplementary set doesn't. This isn't completely safe, but it's the best we can do, and it's certainly better than not comparing at all.

```
bool eql_sups(const sups_t *cursups, const sups_t *targetsups)
{
  int     i, j, n  =   targetsups->size;
  int     diff     =   cursups->size - targetsups->size;
  gid_t   egid     =   getegid();

  if( diff > 1 || diff < 0 ) return false;

  for(i=0, j=0; i < n; i++, j++)
    if( cursups->list[j] != targetsups->list[i] ) {
      if( cursups->list[j] == egid )   i--; // skipping j
      else                             return false;
    }

  // If reached here, we're sure i==targetsups->size. Now, either
  // j==cursups->size (skipped the egid or it wasn't there), or we didn't
  // get to the egid yet because it's the last entry in cursups
  return j    ==    cursups->size ||
    (j+1      ==    cursups->size && cursups->list[j] == egid);
}
```

**FIGURE 5: WHEN COMPARING THE CURRENT SUPPLEMENTARY ARRAY TO THE TARGET ARRAY, WE IGNORE THE EGID IF IT'S INCLUDED IN THE FORMER.**

## NONCOMPLIANT FREEBSD BEHAVIOR

Kernel designers might be tempted to internally represent the egid as just another entry in the supplementary array, as this can somewhat simplify the checking of file permissions. Indeed, instead of separately comparing the file's group against (1) the egid of the process and (2) its supplementary array, only the latter check is required. The aforementioned POSIX rule that allows `getgroups` to also return the egid reflects this fact. But POSIX also explicitly states that "set[*]gid function[s] shall not affect the supplementary group list in any way" [12]. And, likewise, `setgroups` shouldn't affect the egid. So such a design decision, if made, must be implemented with care.

The FreeBSD kernel has taken this decision and designated the first entry of the supplementary array to the egid of the process. But the implementers weren't careful enough, or didn't care about POSIX semantics [4]. When trying to understand why the verification code in Figure 2 sometimes fails in FreeBSD, we realized that the kernel ignores the aforementioned POSIX rules and makes no attempt to mask the internal connection between egid and the supplementary array. Thus, when changing the array through `setgroups`, the egid becomes whatever happens to be the first entry of the array. Likewise, when setting the egid (e.g., through `setegid`), the first entry of the array changes accordingly, in clear violation of POSIX. The code in the beginning of Figure 4 accommodates this noncompliant behavior. Additionally, whenever we need to set the egid, we always make sure to do it after setting the supplementary groups, not before (see Figure 2).

## TEMPORARILY DROPPING AND RESTORING PRIVILEGES

Our implementation also includes functions to temporarily drop privileges and to restore them. They are similar to Figure 2 in that they accept a "target identity" `ucred_t` argument, they treat supplementary groups identically, and they verify that the required change has indeed occurred. When dropping privileges temporarily, we change only the euid/egid if we can help it (namely, if the values before the change are present in the real or saved IDs, which means restoration of privileges will be possible). Otherwise we attempt to copy the current values to the saved IDs before making the change. (Unfortunately, this will fail on AIX for nonroot processes.) The algorithm that restores privileges performs operations in the reverse order: first restoring uids, and only then restoring groups; saved and real IDs are unaffected.

### CAUTION!

Identity is typically shared among threads of the same application. Consequently, our code is not safe in the presence of any kind of multithreading: Concurrent threads should be suspended, or else they run the risk of executing with an inconsistent identity. Likewise, signals should be blocked or else the corresponding handlers might suffer from the same deficiency.

The algorithms described in this article do not take into account any capabilities system the OS might have (e.g., "POSIX capabilities" in Linux [8]). Capabilities systems, if used, should be handled separately.

## Conclusion

Correctly changing identity is an elusive, OS-dependent, error-prone, and laborious task. We therefore feel that it is unreasonable and counterproductive to require every programmer to invent his or her own algorithm to do so, or to expect programmers to become experts on these pitfalls. We suggest that the interests of the community would be better served by a unified solution for managing process privileges, and we propose the approach outlined in this article as one possible basis for such a solution. Our code is publicly available [18]. We welcome suggestions, bug reports, and extensions.

### REFERENCES

[1] M. Bishop, "How to Write a Setuid Program," *;login* 12(1) (Jan./Feb. 1987).

[2] H. Chen, D. Wagner, and D. Dean, "Setuid Demystified," in *11th USENIX Security Symp.*, pp. 171–190 (Aug. 2002).

[3] D. Dean and A.J. Hu, "Fixing Races for Fun and Profit: How to Use Access(2)," in *13th USENIX Security Symp.*, pp. 195–206 (Aug. 2004).

[4] R. Ermilov, R. Watson, and B. Evans, [CFR] ucred.cr_gid, thread from the FreeBSD-current mailing list: http://www.mail-archive.com/freebsd-current@freebsd.org/msg28642.html (June 2001) (accessed March 2008).

[5] Exim Internet mailer: http://www.exim.org/ (accessed March 2008).

[6] Exim-4.69/src/exim.c, source code of exim 4.69: ftp://ftp.exim.org/pub/exim/exim4/exim-4.69.tar.gz (accessed March 2008).

[7] W. Linton and L. Huff, "Easier Said Than Done," performed by The Essex (July 1963): http://www.youtube.com/watch?v=tgJ1ssTJtnA (accessed March 2008).

[8] Man capabilities(7)—Linux man page—overview of Linux capabilities: http://linux.die.net/man/7/capabilities (accessed Mar 2008).

[9] Man getgroups(2)—the Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 edition: http://www.opengroup.org/onlinepubs/000095399/functions/getgroups.html (accessed March 2008).

[10] Man proc(4)—Solaris 10 reference manual collection: http://docs.sun.com/app/docs/doc/816-5174/proc-4?l=en&a=view (accessed March 2008).

[11] Man setfsuid(2)—Linux man page: http://linux.die.net/man/2/setfsuid (accessed March 2008).

[12] Man setgid(2)—the Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 edition: http://www.opengroup.org/onlinepubs/000095399/functions/setgid.html (accessed Jan. 2008).

[13] Man setuidx—AIX Technical Reference: Base Operating System and Extensions, Volume 2: http://publib.boulder.ibm.com/infocenter/systems/topic/com.ibm.aix.basetechre f/doc/basetrf2/setuid.htm (accessed March 2008).

[14] Nerd Gurl, "Why Can't I Ever Achieve My Goals?" Yahoo! Answers (Jan. 2008): http://answers.yahoo.com/question/index?qid=20080101143342AAQ1jbO (accessed March 2008).

[15] D.M. Ritchie, Protection of Data File Contents, Patent No. 4135240 (July 1973): http://www.google.com/patents?vid=USPAT4135240 (accessed March 2008).

[16] J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems, *Proc. of the IEEE* 63(9), 1278–1308 (Sept. 1975).

[17] C. Torek and C.H. Dik, Setuid mess (Sept. 1995): http://yarchive.net/comp/setuid_mess.html (accessed March 2008).

[18] D. Tsafrir, D. Da Silva, and D. Wagner, "Change Process Identity": http://www.research.ibm.com/change-process-identity or http://code.google.com/p/change-process-identity.