USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

Enhancements to the Linux Kernel for Blocking Buffer Overflow Based Attacks

Massimo Bernaschi[1] Emanuele Gabrielli[2] Luigi V. Mancini[2]

[1] Istituto Applicazioni del Calcolo, CNR
Viale del Policlinico 137, 00161 Roma, Italy
massimo@iac.rm.cnr.it

[2]Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza", 00198 Roma, Italy
gabriell@dsi.uniroma1.it, lv.mancini@dsi.uniroma1.it

Abstract

We present the design and implementation of a cost-effective mechanism which controls the invocation of *critical*, from the security viewpoint, system calls.

The integration into existing UNIX operating systems is carried out by instrumenting the code of the system calls so that the system call itself once invoked checks to see whether the invoking process and the argument values passed comply with the rules held in an access control database.

A working prototype able to detect and block buffer overflow attacks is available as a small set of "patches" to the Linux operating system kernel source.

1 Introduction

We propose an approach to the control of system calls which requires minimal additions to the kernel code and neither changes to the syntax and semantics of the system calls nor modifications of existing kernel data structures and algorithms. All kernel enhancements are transparent to existing or new programs. No change in the source code or special compiling procedure is required.

Basically when *controlled* system calls are invoked, our mechanism checks whether the invoking process and the value of the arguments comply with the rules kept in an Access Control Database (ACD) placed within the kernel.
To reduce the cost of the checks, a detailed analysis of privileged applications and system calls is carried out. This allows to identify the set of primitives which may be dangerous for the system security.

As an example of this methodology, we have designed and implemented a prototype which prevents privileged processes from compromising the security and integrity of the Linux operating system when they are subverted by means of techniques like *buffer overflow*.

Buffer overflow is a widely known technique [AlephOne] which allows to force a process generated by a buggy program in executing "fake" instructions injected by the attacker. If the technique is successfully applied to a privileged process and the fake code is used, for instance, to start the execution of an interactive shell, the attacker gains the access to a privileged shell.

During the design phase, the complete set of Linux system calls has been analyzed. The result of the analysis shows that by adding access control tests to a small number of system calls, the protection against buffer overflow is complete and can not be bypassed by executing unprotected system calls. This reduces the cost of system call interception since the invocation of most system calls is not checked.

Any process running with root privileges is a potential target of a buffer overflow attack. Our control mechanism prevents these processes from executing unexpected system calls if they undergo an attack.

2 Problem analysis

2.1 Privileged processes

For the purpose of our discussion, a privileged process may belong to one of the following three categories:

interactive: This is a generic process started by the system administrator. Both the User IDentifier (UID) and the Effective User IDentifier (EUID) are equal to 0. It does not make much sense to monitor such processes, since any user able to start them has the full control of the system. However we must prevent a privileged process from migrating to this category if it was started in a different one.

background: This is, usually, either a daemon process started at boot time or a process started periodically by the cron daemon on root behalf. Following [Stevens] and [Comer] we assume that such processes *never* need a control terminal. To distinguish them within the kernel, we resort to the following check:

```
!((proc)->euid)&&((proc)->tty==NULL)
```

Here, the first logical clause checks whether the process runs with root privileges (EUID=0) whereas the second one checks whether the process has a controlling terminal. We block any attempt made by these processes to re-acquire a control terminal. Note that a daemon can still open a terminal device (e.g. `/dev/tty` or `/dev/console`) to log error messages.

setuid: When a program with *setuid* access mode is executed, the effective UID of the process is set equal to the UID of the program file owner. As a consequence, the access to files and system resources is carried on with the identity of the owner of the program file. This is the standard UNIX mechanism to grant ordinary users with special privileges on a temporary basis. A process can be identified as setuid to root (EUID=0) by means of the following simple check:

```
!((proc)->euid)&&(proc)->uid
```

Note that a setuid process started by the user *root* has $UID = 0$. For this special case the same considerations made for an interactive root process apply.

2.2 System calls analysis

A drastic technique to prevent the damages produced by buffer overflow attacks would be to abort the execution of all system calls invoked by non-interactive root processes. However, the abort of all system calls is not possible (root processes can legally invoke some system calls), and is not needed (many system calls are not dangerous even when invoked via a buffer overflow attack).

The system calls available in Linux 2.2 have been grouped in categories according to their functionality as reported in table 1. In addition, each primitive has been classified according to the level of threat that it may introduce following the definition reported in table 2.

Currently we address the issues raised by the system calls classified as *threat* level 1 which are reported in table 3.

For different reasons no system call in the groups IV-IX can be used to gain the control of the system. For instance system calls in group IX return immediately the error `ENOSYS`, whereas primitives in group VIII can not be invoked by a generic user process (even if it is a privileged process).

As to the system calls in group III, a subverted process may use them for loading a *malicious* module. Our investigation shows that `create_module` is the only primitive which reaches the threat level 1 since no module can be activated without invoking `create_module`. The term *malicious* in table 3 has a very broad meaning. Basically we block any module which is not listed in the ACD.

Among the primitives related to process management, ten reach the highest level of danger for the system security. The `execve` can be used to start a root shell. The other nine primitives set user and group identifiers. It is worth noting that `capset` allows a process only to restrict its capabilities, so it belongs to class 3.

I	File system and devices	V	Communication
II	Process management	VI	Time and timers
III	Module management	VII	System information
IV	Memory management	VIII	Reserved
		IX	Not implemented

Table 1: System calls categories

threat level	threat description
1	May be used to get full control of the system
2	May be used for a denial of service attack
3	May be used for subverting the behavior of the invoking process
4	It is harmless

Table 2: Threat level classification

system calls	dangerous parameter
`chmod, fchmod`	a system file or a directory
`chown, fchown, lchown`	a system file or a directory
`execve`	an executable file
`mount`	on a system directory
`rename, open`	a system file
`link, symlink, unlink`	a system file
`setuid, setresuid, setfsuid, setreuid`	UID set to zero
`setgroups, setgid, setfsgid, setresgid, setregid`	GID set to zero
`create_module`	a malicious module

Table 3: Threat level 1 system calls

The ten system calls related to the file system classified as threat level 1 require special attention. It is apparent that the execution of

```
chmod(''/etc/passwd'',0666)
```

compromises the OS authentication mechanisms. However, it is necessary to consider *chains* of system calls as well. For instance, `chown` and `chmod` primitives can be used in a two-steps procedure to create a setuid shell, whereas the following sequence:

```
unlink(''/etc/passwd'')
link(''/tmp/passwd'',''/etc/passwd'')
```

produces the same result of the `rename` primitive.

Moreover, by means of a buffer overflow, it is possible to execute code that adds to the `/etc/passwd` (and/or `/etc/shadow`) file a new user with UID=0 or adds a fake *.rhosts* file to the root home directory. Although less common, these exploits are, by no means, less dangerous than those based on the classic *shellcode*. Most attacks that involve a file require at least two system calls. A first one to open the file and a second one to modify it. In this case, in accordance with [Goldberg] we assume that it is necessary to monitor just the `open` primitive. That is the reason why the `write` system call is not considered threat level 1.

In building an ACD for the `open` primitive a number of different situations must be considered. Several setuid programs or root daemons may open, for good reasons, critical files (e.g., `/etc/hosts.allow`). For the time being, we assume that opening such files in read-only mode is harmless. We understand that it may be possible to steal precious information like the encrypted passwords or the names of "trusted" hosts but we are going to address such issues in a second time. We recall that the checks are enforced on setuid or daemon programs just in case they invoke critical system calls with root privileges (that is with $EUID = 0$). Since most of the times these programs give up to their privileges (the EUID is set equal to the real UID) before opening files in write mode, the access to user files does not need to be authorized. A detailed study of setuid and daemon programs has been carried out to define which directories and files must be included in the ACD. This has been realized by both source code inspection and analysis of the results produced by tools like the `strace` command which intercepts and records the system calls invoked by a process.

```
/* execve_acd */

typedef struct setuid_proc_id {
        char comm[16];
        unsigned long count;
} suidpid_t;

typedef struct  setuid_program {
        suidpid_t suidp_id;
        suidp_t   * next;  /* next program */
} suidp_t;

typedef struct exe_file_id {
        __kernel_dev_t   device; /* device number     */
        unsigned long    inode;  /* inode number      */
        __kernel_off_t   size;   /* size              */
        __kernel_time_t  modif;  /* modification time */
} efid_t;

typedef  struct executable_file {
        efid_t   efid; /* info for file identification */
        int prog_nr;
        /* number of programs that can invoke this exe */
        suidp_t  *programs; /* list authorized programs */
} efile_t;

typedef struct executable_file_list  {
        efile_t  lst[NR_EXE];
        unsigned int total;
        /* total number of exe in the list */
} eflst_t;
```

Figure 1: The layout of the `execve_acd` data structure

3 The Access Control Database

The Access Control Database contains a section for each system call kept under control. For instance, the working prototype maintains a `setuid_acd` data structure to check the access to the `setuid` system call, and an `execve_acd` data structure to check the access to the `execve` system call. The layout of `execve_acd` data structure is shown in Figure 1:

The execve_acd is composed by two arrays of `eflst_t` structures:

admitted: an executable file F has an entry in this structure if, at least, one privileged program needs to execute F via an `execve`. The information stored in the entry is the list of all

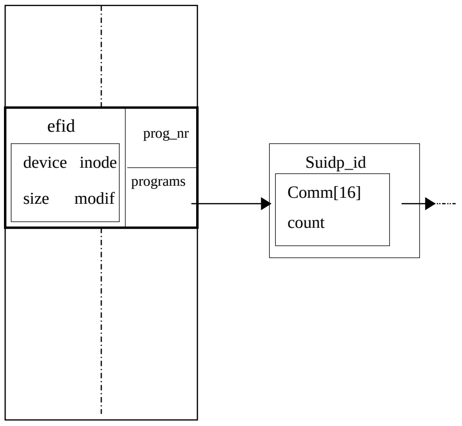

Figure 2: The layout of the *admitted* data structure

privileged programs which may invoke F.

failure: this list keeps a log of the unauthorized attempts (that is, not explicitly allowed by the `admitted` data structure) of invoking `execve` by any setuid process.

Figure 2 shows the `admitted` data structure which is an array where each element refers to an executable file and points to a list of setuid programs that can execute that file.

Each element of the admitted data structure contains these fields:

`efid`: identifies the executable file F. The information stored in efid are:
`device` that is the device number of the file system to which file F belongs;
`inode` that is the inode number of file F;
`size` that is the length in byte of file F;
`modif` which keeps the last modification time of file F.
The pair of information `device` and `inode` identifies file F in a unique way within the system whereas the information `size` and `modif` allow to detect unauthorized modifications of the file contents.

`programs`: is a pointer to the list of privileged programs which can execute file F.

We have introduced a new system call `sys_setuid_aclm` for reading and modifying the information kept in the ACD.

This system call can be invoked only by interactive root processes with EUID=0 and UID=0. These constraints are required to prevent a subverted setuid or root daemon from tampering the ACD.

The system administrator (root user) can manage the ACD resorting to the `sys_setuid_aclm` system call through a new command, named `aclmng` which offers the following options:

-l lists the contents of the Access Control Database kept in kernel space;

-L loads in kernel space the Access Control Database from file `/etc/bop/acd`, mostly used during booting;

-w writes the Access Control Database from kernel space into file `/etc/bop/acd`, mostly used during system shutdown;

default with no options, -l is assumed.

4 The reference functions

This section presents some examples of the extensions introduced to control the system calls defined in section 2.2 as threat level 1.

execve As shown in Figure 3, the new fragment of code is added at the beginning of this system call right after the file has been opened. The `check_rootproc()` function authenticates the privileged process that invokes the `execve` system call and checks in the Access Control Database whether the calling process has the right to execute the program whose name is passed as first parameter. The system call execution is denied when `check_rootproc` returns one of the two following values:

EXENA: the calling process is not authorized to execute the requested program. That is, the program name is not present at all in the Access Control Database or the calling program is not listed in the `programs` field of the admitted list in the Access Control Database.

EFNA: the calling process is authorized to execute the requested program, but the file is not authenticated, e.g. the modification time or the size do not match.

```
/*
 * sys_execve() executes a new program.
 */
int do_execve
    (char * filename, char ** argv, char ** envp, struct pt_regs * regs){

    ...
    dentry = open_namei(filename, 0, 0);
    retval = PTR_ERR(dentry);
    if (IS_ERR(dentry))
      return retval;
    ...
    retval = prepare_binprm(&bprm);

    /*********** BUFF_OVERFLOW PATCH  **************************/
    rc=check_rootproc(bprm.dentry->d_inode);
    if ((rc==EXENA) || (rc==EFNA)) {
      printk(BOP_LEVEL"BOP kernel:do_execve psuid %s no
             authorized to exec file %s\n",current->comm,filename);
      printk(BOP_LEVEL" by euid %d uid %d\n",
             current->euid,current->uid);
      if (rc==EXENA)
        printk(BOP_LEVEL" EXE NO AUTHORIZED\n");
      else  printk(BOP_LEVEL" EXE NO AUTHENTICATED\n");
      return rc;
    }
    /************************************************************/
    ...
    if (retval >= 0)
      retval = search_binary_handler(&bprm,regs);
    if (retval >= 0)
       /* execve success */
        return retval;
    ...
}
```

Figure 3: The “patch” to the execve system call

In the appendix we show the details of the `check_rootproc` function. If the calling process does not run with root privileges (EUID=0) then no further check is performed and the execve proceeds normally. Otherwise, the service is provided if and only if the permission is explicitly contained in the Access Control Database.

setuid For the `setuid` system call, the authentication of the root processes is the same as in the `execve` case. A user running a setuid program which attempts to invoke `setuid(0)` to set the (real) UID equal to 0, is enforced to type the root password. The password keyed is compared with the encrypted copy kept in the Access Control Database. In case of a password mismatch the `setuid(0)` invocation is denied. So far only the program `su` (a setuid program which runs a shell with substitute user and group ID) needs to be monitored with this mechanism.

chmod The major difference with respect to the `setuid` primitive is that no root password is required whereas an additional check is performed on the `filename` argument. If `filename` refers to a regular file or a directory, the operation is denied. This means that the operation is allowed if `filename` refers to a device registered in the ACD. As usual if `chmod` is invoked by an ordinary user process or an interactive root process no additional check is done.

5 Installation and Performance

The software prototype (for availability see section 8) is composed of three parts:

- A kernel patch. The patch has been developed and tested with the version 2.2.12-20 of the Linux kernel. Since it is basically a set of additions to the existing code for the system calls (there are neither changes nor deletions), we do not expect major problems in porting it to other (newer) versions of the kernel.
- The new command `aclmng`
- A modified version of the `chmod` command. The only difference with the original program is that `chmod` accepts new options to add entries in the ACD. For instance, when `chmod` is used to add the setuid functionality (mode *+s*) to the *foo* program, owned by root, the *-p* option allows to specify the list of the programs that *foo* will be allowed to *exec*.

```
chmod +s -p /program1:... :/programN foo
```

  allows the setuid program *foo* to execute any of *program1,... programN* (the execution of any other program is, by default, forbidden). What the modified `chmod` does is to invoke the `sys_setuid_aclm` system call to add the necessary information in the ACD.

The system administrator's duties are limited to run the new version of the `chmod` command. Neither recompilation nor code inspection is required. Messages sent to the syslog by the modified commands and by the system calls, start with the "BOP" prefix to spot them easily.

A very limited degradation of the global performance is expected for a system running our patched kernel. There are a number of reasons for this forecast:

- When a process runs in *user* mode, there is no difference at all with a standard system since all new checks are confined in the kernel.
- Very few primitives include new checks (approximately 10% of the total number of system calls).
- Only a limited subset of the processes execute all the checks.
- With the exception of the `open` primitive, it is unlikely that a setuid or daemon process invokes any of the instrumented system calls more than once during its lifetime.
- The checks do not require any access to "out of core" data, all the info is resident in the kernel memory.
- There are no large data structures, so the lookup is fast without requiring complex algorithms. For instance, the number of setuid programs which need to *exec* other programs is less than five in a typical Linux configuration.

To assess these considerations, a set of experiments has been executed. We have selected four applications and ran them on the same system (a 330 MHz

Pentium II with 128 MB of RAM) with a standard Linux kernel (version 2.2.12) and the same kernel "patched" to include the additional checks. Each test has been repeated 40 times. The applications have been used as follows: `sendmail:` by means of a simple shell script three messages of different size (1 KB, 30 KB and 1 MB) have been sent to a local user;
`lpr:` 8 files of different size (from 1 KB to 10 MB) have been sent to a local printer;
`rsync:` a directory with 1440 files (total size about 10 MB) has been synchronized with a different path (on the same system);
`X server:` by means of the `x11perf` program a 300×300 trapezoid is filled with a 8×8 stipple.

It is apparent looking at the results reported in table 4 that the difference between the average execution times is comparable with the standard deviation of the multiple runs. This confirms that the actual impact of the *patches* on the global system performance is, for all practical purposes, negligible.

6 Related work

Buffer overflow based attacks have been around, at least, since 80's and many solutions have been proposed in the past to solve the problem *definitely.* We are not going to offer here an exhaustive review of all possible approaches. We list just some solutions recently proposed.

Marking both *data* and *stack* regions as non executable may catch most "cut and paste" exploits. A non executable stack is readily implemented [Solar] since it introduces just minor side effects in most UNIX variations (e.g., Linux places code for signal delivery onto the process's stack). Note that there is no performance penalty and existing programs require neither changes nor re-compilation (unless they use exotic features like gcc *trampolines*). The situation is not so simple for the data region. It is not possible to mark it as non-executable without introducing major compatibility problems. Even if this could be solved, there is still the problem of attacks which instead of introducing *new* code, corrupt code pointers. This technique allows to execute dangerous instructions which are already part either of the program or of its libraries [Wojtczuk].

Compiler techniques have been proposed for introducing in an executable code "lightweight" checks on the integrity of functions' return address.

StackGuard [StackGuard] detects and defeats stack smashing attacks by protecting the return address on the stack from being altered. StackGuard places a "canary" word next to the return address when a function is called. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitting an intruder alert into syslog, and then halts.

The major limitation of StackGuard is that it protects against buffer overflows *in the stack.* Unfortunately, heap overflows are less common but, by no means, less dangerous than stack overflows [Conover]. Moreover it has shown very recently that it is possible to exploit buffer overflow vulnerabilities in the stack even in programs compiled with StackGuard or StackShield [Bulba], [Bouchareine]

Other groups in the past have proposed to address security issues by means of special controls on the values of system calls arguments. In [Goldberg] a user–level tracing mechanism to restrict the execution environment of untrusted helper applications is described. Our solution is based on a similar (but independent) analysis of the potential problems associated with *some* system primitives, but we control a different set of programs (i.e. root daemons and setuid programs instead of helper applications). We add our additional checks to the system calls code mainly for performance reasons but the impact on existing kernel code is reduced to the bare minimum (*no* change just additions).

The Domain and Type Enforcement (DTE) is an access control technology which associates a *domain* with each running process and a *type* with each object (e.g, file, network packet). At run time a kernel-level DTE subsystem compares a process's domain with the type of any file or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process's domain does not include a right to the requested access mode for that type. DTE is a very general approach to mandatory access control, however it requires deep kernel modifications (about 17,000 lines of kernel resident code) and 20 new system calls for DTE-aware applications [Badger].

More recently, a high level specification language called Auditing Specification Language has been in-

Table 4: Results from performance tests. We report the average execution time (in seconds) and the standard deviation of 40 runs

Application	elapsed time (standard kernel)	elapsed time (*patched* kernel)
sendmail	1.32 ± 0.05	1.33 ± 0.04
lpr	2.08 ± 0.1	2.1 ± 0.15
rsync	10.36 ± 0.8	10.56 ± 0.6
X server	0.101 ± 0.001	0.102 ± 0.002

troduced [Sekar] for specifying normal and abnormal behaviors of processes as logical assertions on the sequence of system calls and system call argument values invoked by the process. Unfortunately, not enough information are available up to now about their "System Call Detection Engine".

7 Future activities

We have described how *simple* enhancements of an existing kernel code can make harmless a well known threat for the system security like buffer overflow based attacks. Our prototype kernel has been in production for eight months in our organizations and so far no fault due to our patches has been reported by the users.

In the short term, we expect to add "reaction" capabilities to our attack detection mechanism. The starting point is to develop a kernel subsystem to manage intrusion attempts. Simple systems have been already used in the past to analyze the intruders' activities in progress without let them notice it. However those systems were not activated on the fly during the intrusion attempt. The real-time intrusion handling mechanism we have in mind requires the migration of the offending process to a distinct system designed to reproduce the original environment as faithful as possible. We are currently investigating which is the best way to implement this technique.

8 Availability

The prototype and an updated version of this document is available from:
`http://www.iac.rm.cnr.it/newweb/tecno/indexsecurity.htm`

References

[AlephOne] Aleph One,*Smashing The Stack For Fun And Profit*, Phrack Mag., V. 7, N. 49, 1996.

[Cowan] Cowan, C. *et al.*, *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, to appear as an invited talk at SANS 2000,
`http://www.cse.ogi.edu/DISC/projects/immunix`.

[Conover] Conover, M. and the *w00w00* Security Team, *w00w00 on Heap Overflows*,
`http://www.w00w00.org/articles.html`

[Solar] Solar Designer, *Non-Executable User Stack*
`http://www.openwall.com/linux`

[Wojtczuk] Wojtczuk R., *Defeating Solar Designer Non-Executable Stack Patch*, Bugtraq mailing list: January 30 1998.

[StackGuard] StackGuard:
`http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard`

[Stevens] W.R. Stevens, *Unix Network Programming*, II edition, Prentice Hall 1998.

[Comer] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Volume III*, Prentice Hall 1998.

[Goldberg] I. Goldberg, *et al.* "A Secure Environment for Untrusted Helper Applications", Proceedings of the USENIX 6^{th} UNIX Security Symposium (1996).

[Badger] L. Badger *et al.*, "A Domain and Type Enforcement UNIX Prototype", Proceedings of the USENIX 5^{th} UNIX Security Symposium (1995).

[Bulba] Bulba and Kil3R, *Bypassing StackGuard and Stackshield*, Phrack Mag., V. 10, N. 56, 2000.

[Bouchareine] P. Bouchareine, *Format bugs*, Bugtraq mailing list: July 18 2000.

[Sekar] R. Sekar, T. Bowen and M. Segal, "On Preventing Intrusions by Process Behavior Monitoring", Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99).

Appendix

The check_rootproc function

```
int check_rootproc(struct inode *ino) {
   int cont=0,iproc=0,error=0;
   suidp_t * suidproc;
   efile_t f;
   suidp_t p;

    if ((IS_SETUID_TO_ROOT(current))||(IS_A_ROOT_DAEMON(current)))  {
      for (;cont<permitted.total;cont++) {
        if((permitted.lst[cont].efid.device==ino->i_dev)&&
           (permitted.lst[cont].efid.inode==ino->i_ino)) {
            if((permitted.lst[cont].efid.size==ino->i_size)&&
               (permitted.lst[cont].efid.modif==ino->i_mtime)) {
                suidproc=permitted.lst[cont].processes;
                for (iproc=1;iproc<=permitted.lst[cont].proc_nr;iproc++)  {
                     if (!strcmp(suidproc->suidp_id.comm,current->comm)) {
                        suidproc->suidp_id.count++;
                        return PSA;
                     }
                     if (iproc<permitted.lst[cont].proc_nr) {
                        suidproc=suidproc->next;
                    }
                }
            } else {
               error=EFNA;
               goto file_exe_unauthorized;
            }
        }
      }
      error=EXENA; /* EXE is not in the database */
      goto file_exe_unauthorized;
    }

    return PNS; /* the process is not setuid to root or root daemon */

    file_exe_unauthorized:
                   f.efid.device=ino->i_dev;
                   f.efid.inode=ino->i_ino;
                   f.efid.size=ino->i_size;
                   f.efid.modif=ino->i_mtime;
                   strncpy(p.suidp_id.comm,current->comm,
                           sizeof(p.suidp_id.comm));
                   p.suidp_id.count=1;
                   do {
                     while (writer_pid!=0){
                       cli(); /* interrupt disabled */
                       if (writer_pid!=0)
                         interruptible_sleep_on(&pid_queue);
                       sti();
                     }
                   } while (!atomic_access(&writer_pid,current->pid));
                   /* start of critical section */
                   do_setuid_put(&(f.efid),&(p.suidp_id),FAILURE);
                   writer_pid=0; /* end of critical section */
                   atomic_access(&writer_pid,0); /* release of the lock */
                   return error;
}
```