Operating System Enhancements to Prevent the Misuse of System Calls

Massimo Bernaschi
IAC-CNR
Viale del Policlinico, 137
00161 Rome, Italy
massimo@iac.rm.cnr.it

Emanuele Gabrielli
IAC-CNR
Viale del Policlinico, 137
00161 Rome, Italy
gabrielli@iac.rm.cnr.it

Luigi V. Mancini
Dip. Scienze Informazione
Univ. di Roma "La Sapienza"
00198 Rome, Italy
lv.mancini@dsi.uniroma1.it

ABSTRACT

We propose a cost-effective mechanism, to control 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.
This method provides simple interception of both system calls and their argument values and do not require changes in the kernel data structures and algorithms. All kernel modifications are transparent to the application processes that can continue to work correctly without needing changes of the source code or re-compilation. A working prototype has been implemented inside the kernel of the Linux operating system, the prototype is able to detect and block also buffer overflow based attacks.

Keywords

access control database, buffer overflow based attacks, isolation, Linux, system calls interception

1. INTRODUCTION

Most of the conventional techniques for intrusion detection are based on some form of analysis of audit data and system log files [20]. The idea is to spot anomalies (i.e. a root login during system's administrator vacations) and to check periodically the whole system for unexpected changes in the configuration (i.e., new users with administrative privileges). Another common practice is to perform periodically a comparison between the properties (size, access mode, ...) of system files and a *reference* list. This procedure is aimed to spot *Trojan horses*, that is programs, left by an intruder, which look harmless but allow her/him to take full control of the system. The main advantage of these methods is of being *non intrusive*, i.e. they do not require changes to the Operating System (OS) or system commands. However, they cannot be considered as procedures of intrusion prevention since, most of the times, the detection is completed after the attack succeeds.
We believe that immediate detection of security rules violations can be achieved by monitoring the system calls made by processes, and blocking any malicious invocations of system calls from being completed. Hereafter we propose a mechanism for system calls interception at the OS kernel level. Our method requires minimal additions to the kernel code and no change to the syntax and semantics of the system calls. For UNIX-like OS's whose source code is available, we propose to "instrument" the system calls so that the system call itself once invoked checks whether the invoking process and the value of the arguments comply with the rules kept in a small access control database within the kernel. Common penetration techniques that involve tricking the system into running the intruder's own program in privileged mode are blocked by this approach.

A detailed analysis of the system calls can simplify the design and implementation of this OS reference monitor by identifying the system calls dangerous for the system security and by reducing the cost of system calls monitoring. As an example of this methodology, we have designed and implemented a prototype for monitoring system calls which blocks buffer overflow attacks before they can complete. Note that these are just examples of possible attacks, indeed our OS reference monitor is intended to protect against any technique that allows an attacker to *hijack the control* of a privileged process. In particular, our OS based solution is effective against buffer overflow attacks regardless of the buffer location whereas most of the proposed solutions address just the case of stack smashing (see Section 5). In addition, our solution allows several response options to be taken into consideration to handle the attack. This is an additional argument in favor of reference monitoring of system calls, indeed the basic OS cannot stop buffer overflow while this attack is just one of many attacks that our approach protects against.

The key issues addressed by this work can be summarized as follows.

1. detect illegal invocation of *critical* system calls before they complete so to prevent attackers to hijack the control of any privileged process.
2. make a case for reference monitoring of system calls;
3. allow an efficient check of the argument values of the system calls;

CCS'00, Athens, Greece

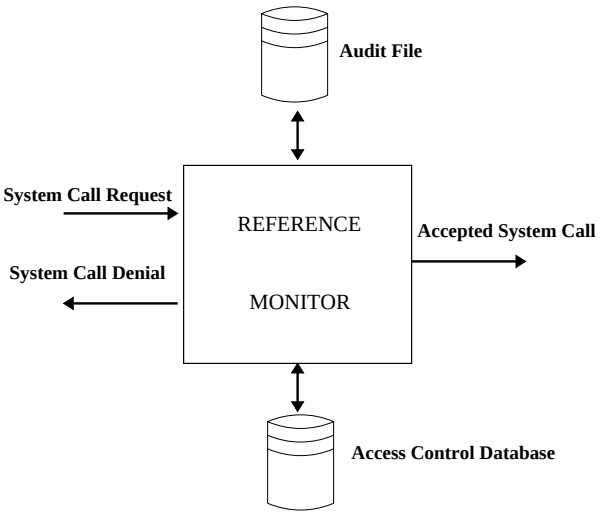

Figure 1: Reference Monitor

4. implement a secure OS by means of lightweight extensions of the kernel, in particular without requiring changes in *existing* data structures and algorithms;
5. support, thanks to the immediate detection of possible attacks, other extensions of the OS to confine and tolerate intrusive processes running together with legitimate processes. This could allow a safe analysis of the intrusive processes while the intrusion is in progress so to determine potential collusions among intruders;

The rest of the paper is organized as follows. Section 2 overviews the main ideas behind our approach. Section 3 describes the enhancements needed to detect buffer overflow attacks before they can complete. Section 4 gives details of the new operations and data structures used in the current implementation in the Linux Kernel. Section 5 reviews related approaches presented in the literature. Section 6 concludes the paper and discusses future activities.

2. OVERVIEW OF THE APPROACH

Our design is based on the *reference monitor* concept described in [2]. The only path the processes can take to access the system calls is through the reference monitor shown in Figure 1. The reference monitor consists of two main functions: the *reference function* and the *authorization function*. The reference function is used to make decisions about whether to permit or deny a system call request based on information kept in an *Access Control Database* (ACD). The access control database conceptually contains entries or *access control rules* in the form (process, system call, access mode). The access control rules in the ACD capture conditions on both the system calls and the values of their arguments. For instance, an access control rule of the `execve` system call could specify in the access mode field the list of the executable files that the invoking process can execute, that is the files that the invoking process can legally pass as argument to `execve`. Such feature prevents a privileged process not properly registered from starting "dangerous" programs like an interactive shell. Note that the tests on the file-parameter of a system call will be based on detailed information that include the i-node number of the file, and are not based simply on the file name. As explained in Section 4, this is because tests based on file name can be bypassed by renaming the file-parameter.
The ACD is not part of the reference monitor, but all modifications to such database are controlled by the reference monitor by means of its second component: the *authorization function*. This function is used to monitor changes to individual access control rules.
One of the fundamental principles of a reference monitor is its completeness, meaning that all accesses must be mediated by the monitor. Unfortunately, the implementation at system call entry point level of this principle has a high cost which is constant regardless of the system call [17]. This overhead can be reduced by identifying a subset of system calls to be monitored which gives a complete protection against attacks. In other words, the system cannot be attacked by executing unprotected system calls.
Following this approach, we have built a prototype which detects and prevents *buffer overflow* based attacks before they can complete. The prototype implements the reference monitor functions and in particular the system calls interception *inside* the kernel of the Linux OS.
By means of checks made by the OS kernel *before* the system call is completed, one should be able to prevent any possible side effect of system calls that intruders currently exploit for their attacks.
It is worth noting that the checks made inside the system calls do not require changes in *existing* kernel data structures and algorithms. Hence all the kernel modifications are transparent to the application processes that continue to work correctly requiring neither changes of the source code nor re-compilation.

3. KERNEL ENHANCEMENTS

This section describes the enhancements of the Linux kernel which block any attempt to hijack the control of a privileged process. In particular, these enhancements prevent buffer overflows attacks from being dangerous for the system security.
We address the attacks by which an intruder tries to gain immediate access to the system as privileged user (i.e. root). According to this choice, the prototype monitors those system calls invoked by root processes which may compromise the security and integrity of the operating system. Of course in a system that maintains user sensitive information, a buffer overflow attack should never read or modify user information too.
Given these assumptions, during the design phase, the complete set of system calls has been analyzed to identify the system calls that might be invoked during a dangerous buffer overflow attack. The result of the analysis, summarized in section 3.2, 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, and thus improves the performance of the present prototype.

3.1 The buffer overflow attack

The lack of array bounds checking in C makes possible to *overflow* a memory buffer beyond its boundaries. By means of widely known techniques [1, 15, 8], a malicious user may corrupt one or more memory buffers, belonging to the stack or data segment, in such a way that, typically on return from a function call, a different piece of code, "injected" by the attacker, is executed by the flawed C program. Obviously the process corresponding to the buggy program maintains its "status" including special privileges (if any). As a consequence, 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. System software components, that is commands, daemons, libraries written in C rarely perform all the necessary checks before invoking functions like `sprintf` or `strcpy` that may result in a buffer overflow. This "feature" makes them an obvious target of buffer overflow based attacks specially if these components generate processes running with administrator privileges. So we must prevent any root process from executing unexpected system calls if it undergoes a buffer overflow based attack.
For the purpose of our discussion, a root process may belong to one of the following three categories:

interactive. This is a generic process started by the system superuser. Both the User IDentifier (UID) and the Effective User IDentifier (EUID) are equal to 0. No additional threat is generated by such process since the user who starts it has already the full control of the system;
background. This is, usually, either a daemon process started at boot time or a process started periodically by the cron daemon on root behalf.
setuid. For this kind of process the pair of user identifiers (UID, EUID) has values UID>0 and EUID$=0$. So a process can be identified as setuid to root by means of the following simple macro

```
#define
IS_SETUID_TO_ROOT(proc) !((proc)->euid)&&(proc)->uid
```

On specific OS's other information may be more suitable to identify a setuid process. For example, in the AIX OS a good candidate appears to be the LOGIN UID which is assigned during the initial user authentication and never modified. Our choice is motivated by the consideration that the pair (UID,EUID) is present in all Unix flavors. This should ease the extension of our work to other Unix platforms.
In the rest of the paper we will consider, for the sake of simplicity, just the last class (the setuid processes). The extension to the background root processes should be apparent once we describe how such processes can be identified within the kernel. This is done in the following subsection.

3.1.1 Background root programs

On a typical UNIX system there are always root programs running in "background". Most of the times, the system administrator neither start them directly (i.e. during an interactive session) nor control their execution, so such "unattended" programs are a preferred target of buffer overflow based attacks.
According to [19] it is possible to group them as follows:

1. Daemons processes originated directly by the system initialization scripts. Network servers like the `inetd` superserver, Web server, mail server (e.g., `sendmail`) are often started in this way. Another example is the `syslogd` daemon.
2. Network servers started by the `inetd` super-server to fulfill requests for services like remote access (telnet), file transfer (FTP) and so on.
3. Programs executed on a regular basis by the `cron` daemon. The `cron` daemon itself is started at boot time (i.e. it belongs to group 1).
4. Programs executed just *once* in the future by means of the `at` command. Actually these programs can be considered as a special case of the previous group.
5. Programs started in background during an interactive sessions (i.e. with a & at the end of the command line). This is done mostly for testing purposes or for restarting a daemon that was terminated for some reason.

These programs in general do not have a *controlling terminal*. To tell them apart from root programs running in interactive mode, we resort to the following macro:

```
#define IS_A_ROOT_DAEMON(proc)
            !((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. Following [19] and [7] we assume that a daemon *never* needs a control terminal. As a consequence we block any attempt 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.

Table 1: System calls categories

group	*functionality*	*group*	*functionality*
I	File system, devices	V	Communication
II	Process management	VI	Time and timers
III	Module management	VII	System info
IV	Memory management	VIII	Reserved
		IX	Unimplemented

Table 2: Threat level classification

threat level	*description*
1	Allows to get full control of the system
2	Used for a denial of service attack
3	Used for subverting the invoking process
4	It is harmless

3.2 System calls analysis

The analysis presented in this section identifies which system calls may be dangerous if invoked in a buffer overflow based 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 which is defined in table 2. The details of this classification are summarized in table 3 and 4. In the present study we face the issues raised by system calls classified as *threat* level 1.
For different reasons no system call in the groups IV-IX can

Table 3: System calls with threat level 1 and 2

threat	system call	group	system call	group	system call	group
1	open	I	link	I	unlink	I
1	chmod	I	lchown	I	rename	I
1	mount	I	symlink	I	fchmod	I
1	fchown	I	chown	I	execve	II
1	setuid	II	setgid	II	setreuid	II
1	setregid	II	setgroups	II	setfsuid	II
1	setfsgid	II	setresuid	II	setresgid	II
1	create_module	III				
2	creat	I	mknod	I	umount	I
2	mkdir	I	rmdir	I	umount2	I
2	ioctl	I	dup2	I	truncate	I
2	ftruncate	I	quotactl	I	afs_syscall	I
2	flock	I	nfsservctl	I	fork	II
2	brk	II	kill	II	setrlimit	II
2	reboot	II	setpriority	II	ioperm	II
2	iopl	II	vm86old	II	clone	II
2	modify_ldt	II	adjtimex	II	sched_setparam	II
2	vfork	II	vhangup	II	sched_setscheduler	II
2	vm86	II	delete_module	III	swapon	IV
2	swapoff	IV	mlock	IV	mlockall	IV
2	stime	V	settimeofday	V	nice	V
2	socketcall	VI	ipc	VI	sethostname	VII
2	syslog	VII	setdomainname	VII	_sysctl	VII

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 (e.g. a module that is not listed in /lib/modules). 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` first.
The largest set (78) of system calls is related to process management. Among these primitives, 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.
The ten system calls related to the file system classified as threat level 1 require special attention. It is apparent that

```
chmod(``/etc/passwd'',0666),
chown(``/etc/passwd'',intruder,intruder_group)
rename(``/tmp/passwd'',``/etc/passwd'')
```

compromise 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, with a buffer overflow, it is possible to execute code that directly remove the root password from `/etc/passwd` (and/or `/etc/shadow`), add to the same file a new user with UID=0 or add a fake *.rhosts* file to the root home directory.
An intruder having access to a CD-ROM connected to the "victim" system may build a filesystem which contains a fake /bin with Trojan Horse programs. Then, by means of the following invocation

```
mount(``/dev/hdb'',``/bin'',``ext2'',MS_MSG_VAL,NULL)
```

the intruder may cover the legitimate /bin directory with his fake /bin.
Although less common, these exploits are, by no means, less dangerous than those based on the "classic" *shell-code*. 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 [12] 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` a number of different situations must be considered. Several setuid programs or root daemons may open, for good reasons, critical files. For the time being we are going to monitor just the access to files in write-mode. At times it may be necessary to let root programs write any file in specific directories defined in the ACD. This is the case of directories like */var/mail* or */var/spool/mqueue* used by `sendmail`. For directories like */etc* a fine-grain control is required. So, a file belonging to */etc* may be modified just in case its identifiers (inode and device number) are explicitly inserted in the ACD.
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 the `strace` command which intercepts and records the system calls invoked by a process. Table 5 summarizes the results of the analysis.

Table 4: System calls with threat level 3 and 4

threat	system call	group	system call	group	system call	group
3	read	I	write	I	close	I
3	chdir	I	lseek	I	dup	I
3	fcntl	I	umask	I	chroot	I
3	select	I	fsync	I	fchdir	I
3	_llseek	I	_newselect	I	readv	I
3	writev	I	poll	I	pread	I
3	pwrite	I	sendfile	I	putpmsg	I
3	utime	I	exit	II	waitpid	II
3	ptrace	II	signal	II	setpgid	II
3	setsid	II	sigaction	II	ssetmask	II
3	sigsuspend	II	sigpending	II	uselib	II
3	wait4	II	sigreturn	II	sigprocmask	II
3	personality	II	rt_sigreturn	II	rt_sigaction	II
3	rt_sigprocmask	II	rt_sigpending	II	rt_sigtimedwait	II
3	rt_sigqueueinfo	II	rt_sigsuspend	II	capset	II
3	sched_yield	II	prctl	II	mmap	IV
3	munmap	IV	mprotect	IV	msync	IV
3	munlock	IV	munlockall	IV	mremap	IV
3	pause	V	setitimer	V	nanosleep	V
4	oldstat	I	oldfstat	I	access	I
4	sync	I	pipe	I	ustat	I
4	oldlstat	I	readlink	I	readdir	I
4	statfs	I	fstatfs	I	stat	I
4	lstat	I	fstat	I	olduname	I
4	bdflush	I	sysfs	I	getdents	I
4	fdatasync	I	getpmsg	I	getpid	II
4	getuid	II	getgid	II	geteuid	II
4	getegid	II	acct	II	getppid	II
4	getpgrp	II	sgetmask	II	getrlimit	II
4	getrusage	II	getgroups	II	getpriority	II
4	sched_getscheduler	II	getsid	II	getcwd	II
4	sched_getparam	II	capget	II	getpgid	II
4	sched_get_priority_max	II	getresgid	II	getresuid	II
4	sched_get_priority_min	II	get_kernel_syms	III	init_module	III
4	sched_rr_get_interval	II	times	V	time	V
4	query_module	III	gettimeofday	V	getitimer	V
4	sysinfo	VII	uname	VII	idle	VIII
4	break	IX	ftime	IX	mpx	IX
4	stty	IX	prof	IX	ulimit	IX
4	gtty	IX	lock	IX	profil	IX

Table 5: Threat level 1 system calls

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`	modules not in /lib/modules

```
static char rpasswd[LEN_PWD]; /* setuid_acd */

/* 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 2: The layout of some of the access control data structures

4. IMPLEMENTATION

This section presents the current implementation of our control mechanism. The description includes the authorization functions, the data structures added to the OS kernel which implement the Access Control Database (ACD); the new system call to read, write and update the ACD and the reference functions. Some practical information about the installation, the usage and the performance are included as well.

4.1 The authorization functions

The Access Control Database contains a section for each system call under the control of the Reference Monitor. 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 such data structures is shown in Figure 2. The `setuid_acd` contains just the string `rpasswd` which keeps in the kernel memory the encrypted root password. This is used for a stronger authentication of a setuid process that invokes the `setuid` system call.
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 setuid program needs to execute F via an `execve`. The information stored in the entry is the list of all setuid 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 3 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. The failure data structure which is not shown in the figure, is similar to the admitted data structure, but it grows dynamically as long as the kernel intercepts unauthorized invocations by setuid programs. Currently it is kept just for auditing purposes.

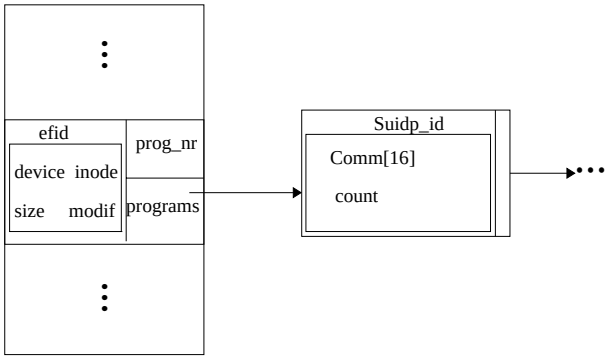

Figure 3: The layout of the *admitted* data structure

Each element of the admitted data structure contains three fields: `efid, prog_nr` and `programs`.
`efid:`
this field identifies the executable file F. The information stored in `efid` are: the device number of the file system to which file F belongs (`device`), the inode number of file F (`inode`), the length in byte of file F (`size`), the last modification time of file F (`modif`).
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.
`prog_nr:`
this field indicates the length of the programs list, that is the number of setuid programs that can invoke file F;
`programs:`
is a pointer to the list of setuid programs which can execute file F. Each element, named `suidp_id`, of the list contains two fields: `comm` and `count`. The string of characters `comm` keeps a copy of the name of the setuid program as it appears in the `comm` field of the process table. The field count is used for statistics and indicates the number of invocations by process `comm` of file F.

In the rest of this section, we describe `sys_setuid_aclm`, a new system call that 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 purpose of `sys_setuid_aclm` is to allow interactions with the kernel for reading and modifying the information kept in the ACD.
Since distinct root processes can access the ACD, conflicts can arise. Hence our new primitive enforces a concurrency control mechanism among the conflicting processes. In particular, a lock variable called `write_pid` is used to implement the mutual exclusion. To avoid race conditions on `write_pid` itself, this variable must be checked and updated atomically. This has been achieved by means of a kernel function called `atomic_access` which resorts to Intel Architecture instruction for atomic exchange: *xchg*.
Actually, the `sys_setuid_aclm` system call is a common front end for six different operations which are briefly described below:

`PUT (exe_file, suid_prog, list)` adds the pair (`exe_file, suid_prog`) in the specified list of the ACD.
`GET (exe_file, suid_prog, param)` reads the pair (`param` → `file_nr, param` → `prog_nr`) from `param` → `list`.
`PUTHEADERACL(header_acl)` stores the header of the ACD in kernel memory space. It is useful when the system is started the first time after the patch installation.
`GETHEADERACL(local_header_acl)` retrieves the header of the ACD from kernel space and stores it in the local variable `local_header_acl`.
`DELETE(exe_file, suid_prog, list)` removes the pair `(exe_file, suid_prog)` from the specified list. If `exe_file` is NULL then it removes all pairs of the form `(*, suid_prog)`, that is it denies the process `suid_prog` from executing any other file;
`PUT_PWD(param)` writes the password `param`→`passwd` in kernel memory space. It deletes temporary copies of the password both in the kernel space and in the user space.
The system administrator can manage the ACD resorting to the `sys_setuid_aclm` system call through a new command, named `aclmng`.

4.2 The reference functions

This section presents some examples of the extensions introduced to control the system calls defined in section 3.2 as threat level 1.

`execve:`

the reference function is invoked at the beginning of this system call after the file has been opened. The `check_rootproc()` function authenticates the root process that invokes `execve` 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.

In 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 assign UID=0 to itself, is enforced to type the root password. The password keyed is then compared with the encrypted copy kept in the Access Control Database. In case of a password mismatch the `setuid(0)` invocation is denied. An example of program we expect to monitor with this mechanism is `su`, a setuid program which runs a shell with substitute user (and group) ID.

`chmod:`

An additional check is performed on the `filename` argument if `chmod` is invoked by a background or setuid process. 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. Important programs, like the *X* server (which is a backgroud root process) would not work without this distinction.

`chown:`

Similar considerations made for `chmod` apply to `chown`, where a check has been added to prevent a background or setuid process from modifying the owner of any regular file or directory.

4.3 Installation and day-by-day operations

The software prototype is available from [5] and it is composed of three parts:

1. 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.
2. the new command `aclmng` (which has been described in section 4.1).
3. a modified version of the `chmod` command. The only difference with the original program is that `chmod` accepts a new option *-p*. When `chmod` is used to add the setuid functionality (mode *+s*) to a program, say *foo*, owned by root, the *- -p* option allows to specify the list of the programs that *foo* will be allowed to *exec*. For instance:

```
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 re-compilation nor code inspection is required.

4.4 Performance

We expect a very limited degradation of the global performance for a system running our enhanced kernel. There are a number of reasons for this forecast:

1. 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.
2. Very few primitives include the additional checks (approximately 10% of the total number of system calls).
3. Only a limited subset of the processes execute all the checks. A generic user process which invokes a threat level 1 system call undergoes just the following controls within the instrumented system calls:

```
if( IS_SETUID_TO_ROOT(current) ||
    IS_A_ROOT_DAEMON(current) ) { ...}
```

So if the process is neither a root daemon nor setuid to root, it does no more than a couple of logical tests (note that the two conditions are evaluated separately just for the sake of code readability).
4. 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.
5. The checks do not require any access to "out of core" data, all the info is resident in the kernel memory.

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:` eight 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.

Table 6 reports the average execution time (in seconds) and the standard deviation of 40 runs. It is apparent looking at the results that the difference between the average execution times is comparable with the standard deviation of the multiple runs. This suggests that the actual impact of the *patches* on the global system performance is, for all practical purposes, negligible.

5. RELATED WORK

Buffer overflow based attacks have been around, at least, since 80's (the *Internet Worm* exploited a buffer overflow in the `fingerd` daemon) and many solutions have been proposed in the past to solve the problem.
Some UNIX distributions (BSDI, Open-BSD) have modified the linker to produce warning messages when a program uses dangerous functions. This approach implies many "false positive" alarms since the use of dangerous functions are not all incorrect whereas overflows occur not only in standard libraries.
Marking both *data* and *stack* regions as "non-executable" would catch most "cut and paste" exploits. A non-executable stack is readily implemented [18] 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. 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 [22].
There are both commercial [13] and public domain [14] solutions which add array bounds checking capabilities to C programs. These packages can be considered as good debugging tools but, in a production environment, their use is not feasible since the performance penalty is barely acceptable (programs undergo a slowdown of about one order of magnitude).
Recently, two compiler techniques have been proposed for introducing in the executable code "lightweight" checks on the integrity of functions' return address.
StackGuard [10] 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.
To be effective, the attacker must not be able to "spoof" the canary word by embedding the value for the canary word in the attack string. StackGuard offers a range of techniques to prevent canary spoofing:

`Random canaries:` the canary word value is chosen at random at the time the program executes. Thus the attacker cannot learn the canary value prior to the program start by searching the executable image.
`Null canary:` the canary word is "null", i.e. 0x00000000. Since most string operations that are exploited by stack smashing attacks terminate on null, the attacker cannot easily spoof a series of nulls into the middle of the string.
`Terminator canary:` not all string operations are terminated by null, e.g. gets() terminates on new line or end-of-file (represented as -1). The terminator canary is a combination of Null, CR, LF, and -1 (0xFF) which should terminate most string operations.

StackGuard is implemented as a small patch to the gcc code generator, specifically the function_prolog() and function_epilog() routines. function_prolog() has been enhanced to lay down canaries on the stack when functions start, and function_epilog() checks canary integrity when the function exits. Any attempt at corrupting the return address is thus detected before the function returns.
The Stack Shield [21] protection system copies the return address in an unoverflowable location (the beginning of the data segment) on function prolog and checks if the two values are different before the function returns. If the two values are different the return address has been modified so Stack Shield terminates the program or tries to let the program run ignoring the attack (risking at maximum a program crash). Stack Shield works as an assembler file processor and is supported by gcc/g++ front ends to automatize the compilation. No code change or other special operations are required.
StackGuard and StackShield offer many nice features: minimum performance penalty, no change in existing code, no constraint is imposed on new code. The major limitation is that they protect against buffer overflows *in the stack*. Unfortunately, heap overflows are less common but, by no means, less dangerous than stack overflows [8]. The solution we propose is effective regardless of buffer location. 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 [6]
Last, but not least, it is necessary to take into account that:

1. gcc is NOT the only C compiler available.
2. Checks introduced by a compiler are not selective: each function of any process is affected. Even if benchmarks of single applications do not show significant performance degradation, it is unclear what is the impact on the performance if the entire system software (kernel, libraries, utilities,...) is compiled with StackGuard (or StackShield).

Other groups in the past have proposed to address security issues by means of special controls on the values of system calls arguments. In [12] a user–level tracing mechanism to restrict the execution environment of untrusted helper applications is described. Our solution is based on a similar

Table 6: Results from performance tests (in seconds)

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	10.16 ± 0.8	10.36 ± 0.6
X server	0.101 ± 0.001	0.102 ± 0.002

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 [3].
More recently, a high level specification language called Auditing Specification Language has been introduced [17] 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 to us about their "System Call Detection Engine".

6. FUTURE ACTIVITIES

We have described how *simple* enhancements (supported by a detailed analysis of the system calls) of an existing kernel code can make harmless well known threats for the system security like buffer overflow based attacks. The prototype kernel has been in use for six months in two organizations and 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 for intrusion tolerance. Simple systems like the *cage* [4] 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. Such intrusion handling subsystem seems adequate to analyze possible collusions among distinct intruders, and for planting trip-wires.
In the medium term, we would like to port the *patches* we developed for Linux to other Unix flavors whose source code is available (xxBSD, Solaris).

Acknowledgements

The work of Luigi V. Mancini was partially supported by the Italian MURST and the Italian CNR under project "ADESSO".

7. REFERENCES

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

[2] Ames S.R., Gasser M., Schell, R.R., "Security Kernel Design and Implementation: An Introduction", IEEE Computer, Vol. 16, N. 7, 14-22, 1983.

[3] Badger L. *et al.*, "A Domain and Type Enforcement UNIX Prototype", Proceedings of the 5^{th} USENIX UNIX Security Symposium, Salt Lake City, UT, June 1995.

[4] Bellovin S. M., Cheswick W. R, "Firewalls and Internet Security", Addison Wesley, 1994.

[5] Bernaschi M., Gabrielli E., Mancini L.V.,"A Reference Monitor Patch to Linux Kernel", ftp://ftp.iac.rm.cnr.it/pub/BufOverP/

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

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

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

[9] Cowan C., *et al.*, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", http://www.cse.ogi.edu/DISC/projects/immunix.

[10] Cowan C. *et al.*, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", 7^{th} USENIX UNIX Security Symposium, San Antonio, TX, Januar 1998. http://www.cse.ogi.edu/DISC/projects/immunix/

[11] GNU Software, "Patch utility", http://prep.ai.mit.edu/software/patch/patch.html

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

[13] Hastings R. and Joyce B., "Purify: Fast Detection of Memory Leaks and Access Errors", Proceedings of the Winter USENIX Conference, 1992, http://www.rational.com/support/techpapers/fast_detection

[14] Jones R. and Kelly P., "Bounds Checking for C", http://www_ala.doc.ic.ac.uk/phjk/BoundsChecking.html

[15] Mudge, "How to write Buffer Overflows", http://www.l0pht.com/advisories/bufero.html

[16] OpenBSD Team, "OpenBSD Operating System", http://www.openbsd.org

[17] Sekar R., Bowen T. and Segal M., "On Preventing Intrusions by Process Behavior Monitoring", 1^{st}

Usenix Workshop on Intrusion Detection and Network Monitoring (ID), Santa Clara, CA, April 1999.

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

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

[20] Sundaram A.,"An introduction to Intrusion Detection", http://www.acm.org/crossroads/xrds2-4/intrus.html

[21] Vendicator, "Stack Shield: A Stack Smashing Tecnique protection tool for Linux", http://www.angelfire.com/sk/stackshield

[22] Wojtczuk R., "Defeating Solar Designer Non-Executable Stack Patch". Bugtraq mailing list: January 30 1998, http://www.securityfocus.com/bugtraq