

# Operating System Support for Process Confinement

José R. Herrero  
Computer Architecture Department

David Benlliure  
Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya\*  
Jordi Girona 1-3, Mòdul D6, 08034 Barcelona, Spain

**Abstract** *Execution of untrusted software can compromise a whole system. Tools for restricting access of software to system resources are essential for security maintenance. Operating systems should offer functionality for building tools which could run in user mode with no special privileges while providing full access control. Thus, they could be made available to any user in the system.*

*In this paper we show ways of extending an operating system in order to provide such functionality. We present the changes introduced in the Linux kernel to offer the minimum functionality necessary for building such a tool. Using the new functionality we were able to code a program for controlling execution of untrusted software through system call interposition.*

**Keywords:** Process confinement, system call interposition, user mode monitor.

## 1 Introduction

There is a strong need for means to secure systems against all sort of attacks. Many efforts have been devoted to preventing them, mainly in the system administrator land. There is, however, a simple and common problem against which administrators cannot do much: the execution of all kind of software by trusted users in the system. This software can, for instance, be received from friends through E-mail or downloaded from internet. The problem comes when this programs contain malicious code,

like Trojan horses or viruses. Their execution can compromise the user account. Thus, there is a need for tools which control and restrict execution of user processes. There must be a way to distinguish between normal user processes and user processes which must run under restrictions.

In the last decade some research has been conducted towards the creation of secure environments for the execution of untrusted software [1,2,11,12,14]. They aim at confining processes in such a way that only a reduced functionality of the system is available to them. Any potentially dangerous operation is not allowed. This environments are commonly referred to as *Sandboxes*, a term introduced in [8] in a slightly different context.

There has also been a lot of work on restricting damage caused by malicious software [6,7] or subverted accounts, specially root [4, 5, 10]. This shows the importance of preventing attacks.

We are looking for a general solution that allows for the secure execution of any program, regardless of the programming language it has been written with, or the accessibility to source code. For such a tool to be made available to users, it must run in user mode with no special privileges. Since all processes access resources via system calls, controlling and restricting those calls is the most general way to tackle this problem. Any effort at creating secure execution environments in user space needs some support from the operating system. It must provide some system calls which allow for a user process to control and restrict the execution of another process.

In this paper we show how we extended the Linux kernel to provide the necessary functionality

\* This work was supported by the Ministerio de Ciencia y Tecnología of Spain and the EU FEDER funds (TIC2001-0995-C02-01)

to be able to create a secure execution environment in user space. We analyze our approach and compare it to other approaches.

## 2 Goals

Our work on process confinement strives after getting maximum security at low cost. We propose some changes in the Linux kernel which are then used to build a user program, the *security monitor*, henceforward referred to as monitor, which can be used to control execution of a process or hierarchy of processes. We will call a traced process a *tracee*.

We will have all tracee system calls controlled by the monitor. The monitor will allow the tracee to continue execution as long as it issues permitted system calls. On the other hand, it will kill the tracee as soon as it requests a forbidden system call or a restricted system call using forbidden parameters.

## 3 Design

The monitor control over the tracee is achieved via system call interposition. This means that the monitor is informed each time a system call entrance or exit is performed by the tracee. The latter remains stopped until the monitor allows it to continue. The monitor is also notified each time the tracee is blocked or terminated.

We considered using the */proc* file system. Its functionality in Linux, however, is very limited compared to that of System V compliant systems. It only provides information on processes and system resources, with very restricted control of the latter. This means we cannot use the */proc* interface for process control under Linux.

Therefore, in order to get this functionality the *ptrace()* system call has been used. This call has traditionally been used for debugging. It is present in most Unix systems, although the way param-

eters can be obtained varies substantially from one system to another.

Linux *ptrace()* functionality, however, lacks two extra requirements which are needed for secure execution of processes:

- The possibility to deny execution of a system call or kill the program before the system call is actually performed
- Inheritance of trace flags across *fork()* calls

If the first requirement is not present, once a system call is issued it cannot be stopped. If the second requirement is not present, a tracee can create a child and then the child may run away from the monitor control, avoiding its security control.

Although this extra requirements are present in System V Unix systems, they are not found in BSD4.4 nor Linux. Consequently we had to modify the Linux kernel to introduce those two extra requirements. These changes though, are very easy to understand and maintain.

### 3.1 Changes to the Linux kernel

We added the possibility to use two extra flags in the process *task\_struct* (the data structure which keeps information about a process):

- *PF\_TRACEINHERIT*  
Used to request inheritance of trace flags
- *PF\_DESTROY*  
Used to request an immediate termination of the traced process

Also, we added two extra commands to *ptrace()*:

- *PTRACE\_TRACEUS*  
Which is similar to *PTRACE\_TRACEME* but sets the *PF\_TRACEINHERIT* flag in addition to the *PF\_TRACED* flag so that a whole hierarchy of subprocesses can be controlled

- **PTRACE\_DESTROY**

Used to request immediate destruction of the tracee. This differs from PTRACE\_KILL in that the latter sends SIGKILL to the tracee. Since signals are only attended at the end of system calls, before returning to user mode, the latter is not useful for aborting the execution of the offending system call

This new commands and flags are then used in *ptrace()* and *fork()* for implementing the desired behavior for the system.

The only change produced in *fork()* is the addition of a conditional that checks whether PF\_TRACEINHERIT is set in order to pass trace flags to the new child. This minor change affects all processes, but is only an extra comparison operation for a call that is executed sparingly along a process' life. Thus, we consider that it produces no impact on performance.

The changes produced in *ptrace()* set and use the PF\_TRACEINHERIT flag: it is set when a PTRACE\_TRACEUS is requested and used to correctly attach a child process when PTRACE\_ATTACH is used. The PF\_DESTROY flag is set in this routine when a PTRACE\_DESTROY command is treated. Thus, it can be later used in *syscall\_trace()*, the routine executed at system call entry and exit time when the process is being traced, to have the process call *sys\_exit()* immediately, before doing the system call proper, avoiding in this way the execution of a forbidden system call.

### 3.2 User mode security monitor

This functionality can then used by a user-level process to control the tracee using the *ptrace()* and *wait4()* calls repeatedly.

The monitor can also limit resources used by tracees, like, for instance, the number of files opened at a given time or the amount of memory allocated. Limits on resources are easily set via the *setrlimit()* system call.

We have managed to build a security monitor using this modifications in the Linux 2.2.17 kernel. Its behavior can be configured via files: lists of limits on resources and prohibited system calls. Only allowed system calls can proceed, while offending ones are forbidden.

## 4 Related Work

There is work on process confinement by means of system call interposition on Solaris systems: *janus* [1], *MAPbox* [2] and *consh* [12]. Actually, both MAPbox and consh are based on janus: they use it as their sandbox while extending its functionality. MAPbox categorizes applications into classes and defines default restrictions for each of them. Consh provides transparent access to local and remote resources. Consequently, janus is the reference software when one considers process confinement.

When janus is executed, a user process is created which acts as a sort of debugger which controls the execution of another process. All the work in janus is done in user space thanks to the process control functionality provided (in some Unix systems) by the */proc* file system [13]. User-level design is highly desirable. However, it requires operating system support. When the operating system doesn't provide it, there is no way to build secure tools in user mode. For the time being, Linux does not provide enough functionality for creating user-level only secure process confinement. There exists a */proc* file system but its functionality is limited to providing information on processes and system resources, with very restricted control of the latter.

We have learned that janus has been ported to Linux. The janus application works at the user-level. However a kernel module was built to provide the required functionality for janus to work properly. Implementing this kernel module required over 3000 lines of code in 15 files, while the user-level program was coded in about 5600 lines. This means a great effort had to be driven into providing the kernel support.

The janus project is closely related to ours. Actually, their purpose is the same: to provide a sandbox for secure execution of untrusted software. They both limit resources and control system calls. Easy configuration is possible for both. There are differences in the design, however. While their design is based on an extension of the kernel functionality through a kernel module, our security monitor is based on an extension of the kernel via a few new commands in the *ptrace()* system call and a subtle change in the *fork()* system call.

Execution speed of traced processes in our security monitor is affected by some restrictions imposed by *ptrace()*. First, the system call arguments and the address space of the child have to be read word by word. Second, selective tracing of system calls is not possible. Instead, all system calls have to be traced. Although the first restriction could be eased with new *ptrace()* functionality, the second one is far more tricky and would require a completely different approach when entering the kernel due to system calls. Instead of basing the decision of system call tracing in the value of a single bit in the process flags, a call by call distinction would have to be done. Entries in the process system call table for traceable system calls should be substituted by a tracing routine, while the others should be left unchanged. Thus, for all the system calls that the user did not want to trace, no performance penalty would be paid. This is the approach taken by the kernel module provided by janus developers.

## 5 Conclusions

Means to control execution of untrusted software are necessary in all systems. In order to make that possible, the operating system must provide some functionality for secure process tracing. We have proved that extending an operating system in such direction is fairly easy while effective.

- Security is enforced
- The addition (to the standard Unix behavior) of trace flags inheritance and the possibility to kill processes at system call entry time suffices for the creation of effective secure environments
- Execution of processes not traced is not affected by our changes
- Implementation of this new kernel functionality in other systems can be very easy.

We believe that all operating systems should, at least, offer this basic functionality so that tools for secure execution of untrusted software can be built.

In case good performance of traced processes is required, however, a greater extension of the operating system functionality is needed. Actually, we are currently extending the Linux /proc virtual file system to allow for process control.

## References

- [1] I. Goldberg, D. Wagner, R. Thomas and E. Brewer, "A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker", In *Proc. of the 1996 USENIX Security Symposium*, San Jose CA, July 1996.
- [2] A. Acharya and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications", In *Proc. of the 9th USENIX Security Symposium*, Denver, Colorado, August 2000.
- [3] L. Badger, D.F. Sterne, D.L. Sherman, K.M. Walker and S.A. Haight, "A Domain and Type Enforcement UNIX Prototype", In *Proc. of the 5th USENIX Security Symposium*, Salt Lake City, Utah, June 1995.
- [4] L. Badger, D.F. Sterne, D.L. Sherman, K.M. Walker and S.A. Haight, "Practical Domain and Type Enforcement for UNIX", 1995 *IEEE Symposium on Security and Privacy*, Oakland CA, May 1995.

[5] K.M. Walker, D.F. Sterne, L. Badger, M.J. Petkac, D.L. Sherman and K.A. Oostendorp, "Confining Root Programs with Domain and Type Enforcement (DTE)", In *Proc. of the 6th USENIX Security Symposium*, San Jose CA, July, 1996.

[6] N. Lai and T. Gray, "Strengthening discretionary access controls to inhibit trojan horses and computer viruses", In *Proc. of the 1988 USENIX Summer Conference*, pp. 275-286, 1988.

[7] P. Karger, "Limiting the damage potential of the discretionary trojan horse", In *Proc. of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[8] R. Wahbe, S. Lucco, T.E. Anderson and S.L. Graham, "Efficient software-based fault isolation", In *Proc. of the Symposium on Operating System Principles*, 1993.

[9] T. Jaeger, A.D. Rubin and A. Prakash, "Building Systems that Flexibly Control Downloaded Executable Content", In *Proc. of the 1996 USENIX Security Symposium*, San Jose CA, July 1996.

[10] S.E. Hallyn and P. Kearns, "Domain and Type Enforcement for Linux", In *Proc. of the 4th Annual Linux Showcase & Conference*, Atlanta, Georgia, October 2000.

[11] L.D. Stein, "SBOX: Put CGI Scripts in a Box", In *Proc. of the 1999 USENIX Technical Conference*, San Jose CA, July 1996.

[12] Albert Alexandrov, Paul Kmiec, and Klaus E. Schauser, "Consh: A Confined Execution Environment for Internet Computations", available at <http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps>, Dec. 1998.

[13] R. Faulkner and R. Gomes, "The Process File System and Process Model in UNIX System V", In *Proc. of the 1991 USENIX Winter Conference*, 1991.

[14] D. Wallach, D. Balfanz, D. Dean and E. Felten, "Extensible Security Architecture for JAVA", In *Proc. of the Sixteenth ACM Symposium on Operating System Principles*, 1997.