Syscall Interception in Xen Hypervisor

Frédéric Beck, Olivier Festor

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

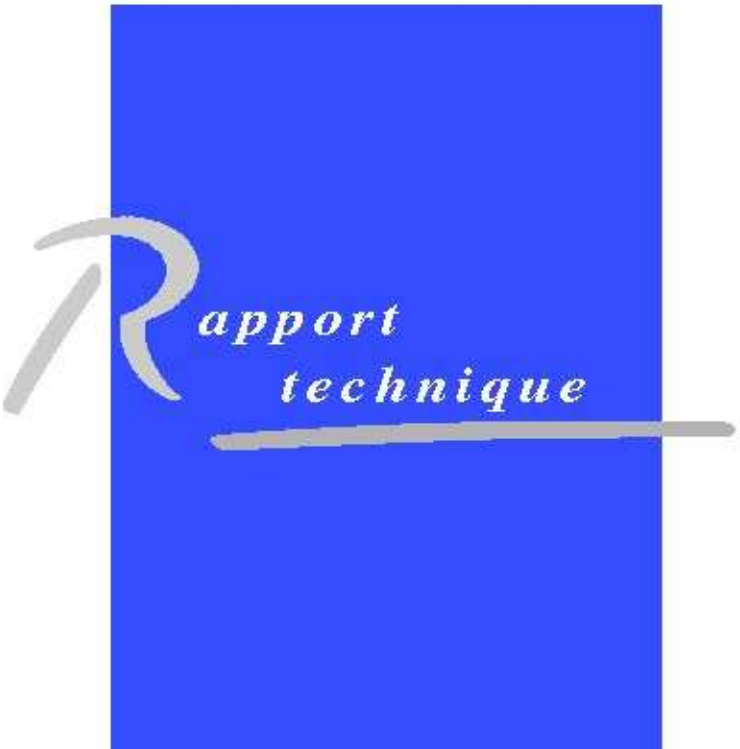
Syscall Interception in Xen Hypervisor

Frederic Beck and Olivier Festor

N° 9999

November 2009

Thème COM



ISSN 0249-0803

Syscall Interception in Xen Hypervisor

Frederic Beck and Olivier Festor

Thème COM — Systèmes communicants
Projet MADYNES

Rapport technique n° 9999 — November 2009 — 19 pages

Abstract: In the scope of the ANR-08-VERS-017 project Vampire, as part of Task 4 on close loop fuzzing, we performed a feasibility study on system calls interception within the Xen virtualization system. In this report, we present the work done and the results obtained during this study.

Key-words: network, security, virtualization

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,

Interception des Appels Systèmes dans l'Hyperviseur Xen

Résumé : Dans le cadre du projet ANR-08-VERS-017 project Vampire, pour la tâche 4 sur le "close loop fuzzing", nous avons réalisé une étude de faisabilité pour l'interception des appels système dans la plate-forme de virtualization Xen. Dans ce rapport, nous présentons cette étude ainsi que les résultats obtenus.

Mots-clés : réseau, sécurité, virtualisation

Contents

List of Figures

1 Introduction

In the scope of the ANR-08-VERS-017 project Vampire, as part of Task 4 on close loop fuzzing, we performed a feasibility study on system calls interception within the Xen virtualization system.

The objective is to fuzz a system or application (e.g. a SIP soft phone) running in a Virtual Machine (VM) while intercepting the system calls (syscalls) issued by that system or application. We aim at understanding what impact the fuzzing has on the target (e.g. detect buffer overflows). This feedback will be used to tune the fuzzing and make it more efficient and target the weaknesses of the application.

During this study, we focused on the Xen [1] virtualization system, and we are aiming at intercepting the syscalls issued by the guests within the hypervisor itself.

In this report, we will first present the Xen hypervisor and the global architecture of the virtualization system. Then, we will present the syscalls mechanisms and how we can intercept them. Finally, we will detail the implementation and modifications brought to the hypervisor before concluding.

2 Xen Hypervisor

In this section, we will give an overview of the Xen virtualization system.

2.1 Virtual Machine Monitor

The core of the Xen Virtualization System is the Hypervisor [2]. It consists on a micro-kernel that translates to extremely low overhead and near-native performance for guests. Xen re-uses existing device drivers (both closed and open source) from Linux, making device management easy. Moreover Xen is robust to device driver failure and protects both guests and the hypervisor from faulty or malicious drivers.

The hypervisor is referred as Dom0 and VMs as DomU or guests. Xen ensures VMs isolation and permits live migration of the guests. It makes possible to run unmodified Operating Systems (OS) thanks to CPU hardware virtualization support (VT for Intel or Pacifica for AMD CPUs).

Within the Xen Virtualization System, we can find two types of virtualization:

- ParaVirtualization (PV guest)
- Full Virtualization (Hardware Virtual Machine - HVM guest)

Figure 1 presents an overview of Xen's architecture.

[1] `http://www.xen.org/`
[2] `http://www.xen.org/products/xenhyp.html`

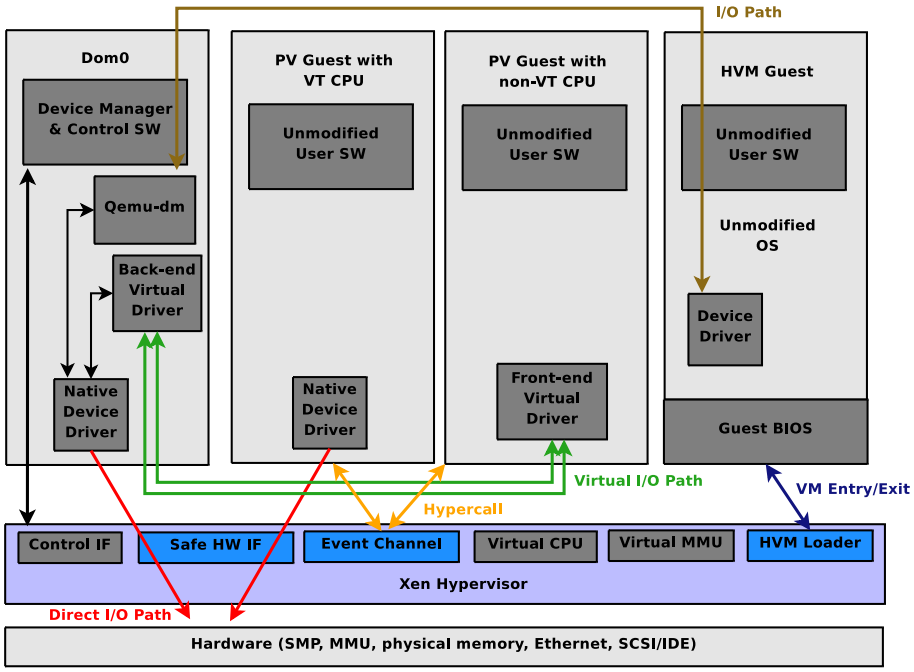


Figure 1: Xen Architecture Overview

2.2 ParaVirtualization

A VM using ParaVirtualization is called a PV guest. OS running as PV guests are typically Linux based systems. ParaVirtualization offers a reduced virtualization overhead and high-level abstractions. the Hypervisor provides hooks (hypercalls) to the guests to perform expensive operations (e.g. privileged-sensitive operations, interrupts, page faults...). This implies that the guests are running a modified kernel to support these hooks.

Xen also presents to the guests a software interface similar but not identical to the underlying hardware if it runs on a non-VT CPU. Otherwise, guests have direct access to devices via native device drivers

2.3 Full Virtualization

A VM using Full Virtualization is called a HVM guest (Hardware Virtual Machine). Full Virtualization permits to run any unchanged OS, typically Windows or other proprietary systems. An HVM guest uses a full virtualization system using help from the hardware capabilities (e.g. VT CPU). Such, it uses a virtual BIOS to boot. Network and disk access are performed with the help of a daemon called Qemu-dm located in the Dom0.

As opposed to the PV guests, HVM guests use hardware emulation. This generates many traps and a high CPU overhead, making their performances worse.

3 System Calls

In this section, we will present the System Calls (syscalls), their mechanisms, and within the Xen hypervisor, how they can be intercepted.

3.1 What are syscalls?

A syscall aims at providing userland processes a way to request services from the kernel [3]. This includes access storage operations, memory or network access, process management... Typical syscalls are *open*, *close*, *read*, *write*...

Each syscall is identified by a number and stored in the syscall table within the kernel. These identifiers are OS dependent (e.g. on Linux, syscalls are listed in the file *unistd.h*). It is possible to add, remove or modify a syscall dynamically by modifying the function pointer in the syscall table with kernel modules.

3.2 What happens in a syscall?

In computer science, hierarchical protection domains, often called protection rings, are a mechanism to protect data and functionality from faults (fault tolerance) and malicious

[3] `http://manugarg.googlepages.com/systemcallinlinux2_6.html`

behavior (computer security). This approach is diametrically opposite to that of capability-based security.

Computer operating systems provide different levels of access to resources. A protection ring is one of two or more hierarchical levels or layers of privilege within the architecture of a computer system. This is generally hardware-enforced by some CPU architectures that provide different CPU modes at the firmware level. Rings are arranged in a hierarchy from most privileged (most trusted, usually numbered zero) to least privileged (least trusted, usually with the highest ring number). On most operating systems, Ring 0 is the level with the most privileges and interacts most directly with the physical hardware such as the CPU and memory.

Special gates between rings are provided to allow an outer ring to access an inner ring's resources in a predefined manner, as opposed to allowing arbitrary usage. Figure 2 shows the example of x86 architectures. [4].

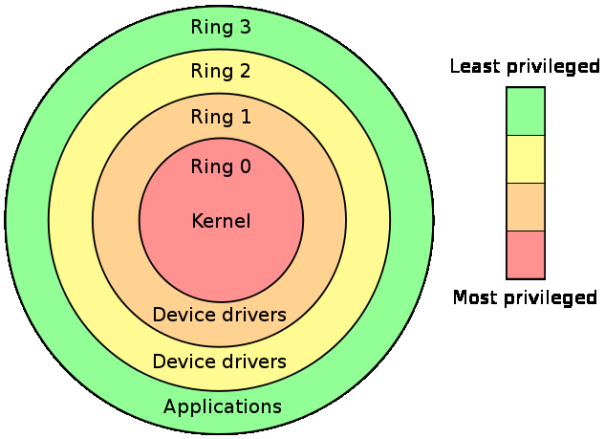


Figure 2: Privilege rings for the x86 available in protected mode

Typically, a syscall is issued when a userland process from ring 3 wants to execute a kernel code snippet located in ring 0. A syscall uses a routine that permits to call ring 0 code from ring 3, and then executes some kernel code to service the request. Thus, a syscall can be seen as:

$$syscall = routine + kernel\ code$$

3.3 2 ways of doing it

There are 2 mechanisms to perform syscall:

[4]source: `http://en.wikipedia.org/wiki/Ring_(computer_security)`

- Software Interrupts
- Fast System Call

3.3.1 Software Interrupts

One way to perform a syscall is to use software interrupts:

1. User process copies the syscall number to execute in the CPU register %EAX
2. Execute *int 0x80* to generate interrupt 0x80
3. *System calls handling* routine executed in ring 0
   - Save the current state
   - Call the appropriate system call handler based on %EAX
4. Return the result to the user process

3.3.2 Fast System Call

Software interrupts method appear to be much slower for Pentium IV CPUs. Thus, a new mechanism has appeared, using the *SYSENTER/SYSEXIT instructions*: One way to perform a syscall is to use software interrupts:

1. Set up *SYSENTER_CS_MSR*, *SYSENTER_ESP_MSR* and *SYSENTER_EIP_MSR* kernel registers
2. Kernel sets up system call entry/exit points for user processes
3. Creates a memory page attached to all processes' address space containing the actual implementation of the system call entry/exit mechanism
4. Call *__kernel_vsyscall* to execute system calls
5. Execution pretty similar to the interrupts method
6. Result pushed to userland via *SYSENTER_RESULT*

3.4 System Call Interception

The objective is to trace the syscalls executed in a VM by a given process in the most transparent possible way. There are several possible mechanisms to do so:

- ptrace: set up system call interception and modification at the user level by modifying the application
- strace: runs a command, logs syscall and signals

- Linux Kernel Audit: use auditing facilities within the kernel
- Modify the syscall table and replace syscalls with wrappers, requires a wrapper per syscall...
- Many others (debuggers...)

The problem common to all these mechanisms regarding our objective is that they are not transparent enough for the application, as it requires a special way of starting/executing it, or modifications within the application itself.

The solution we chose is to intercept syscalls within the Xen Hypervisor itself. Doing so does not require any modification in the application or the guest OS. We will try to obtain a generic solution for PV and HVM guests.

In the next subsections, we will present the thoughts and directions we will follow to perform the syscall interception in the hypervisor for each syscall mechanism.

3.4.1 Software Interrupts

In this case, syscalls use the *0x80 Interrupt* to trigger their execution. The first thing to do in the hypervisor is to disable the *Direct Trap* optimization that allows the syscalls to bypass the hypervisor. Thus, syscalls are forced through the hypervisor and we will be able to catch them.

Then, when *int 0x80* is issued, the syscall will be intercepted and the information we are interested in will be logged.

Finally, we will keep o,n with the regular syscall execution. It will be possible later on to capture the result of the syscall as well.

3.4.2 Fast System Call

If possible, this mechanism will be deactivated, forcing the use of software interrupts. Therefore, the interception will be performed as described in the previous section.

Otherwise, we will track the *SYSENTER/SYSEXIT* instructions. By default, syscall for HVM guests do not trap through the hypervisor. Thus we will need to follow the whole mechanism and place hooks and breakpoints at the right places.

When *SYSENTER_EIP_MSR* is set, alter it to induce a page fault at a known address. The resulting page fault triggers further processing, such as extracting system call number arguments... This mechanism has been successfully implemented in older version of Xen in Ether [5].

[5] `http://ether.gtisc.gatech.edu/`

4 Implementation

The implementation has been realized on a testbed composed of Xen version 3.3.1 running on Debian GNU/Linux. The guests used were one Linux PV guest with kernel 2.6.18-6-xen-vserver-686, and one Windows XP SP2 HVM guest.

In this section, we will detail the implementation for the PV and HVM guests.

4.1 PV Guest

By default, syscalls trap directly through the hypervisor. To perform the interception of these syscalls, they must pass through the hypervisor. To do so, we must disable the Fast System Calls handling optimization. This is done by commenting the following lines in *xen/arch/x86/traps.c:do_set_trap_table()*:

```
if ( cur.vector == 0x80 )
    init_int80_direct_trap(curr);
```

4.1.1 Getting the information to log

Thus, the syscalls trap through the Xen Hypervisor, and more specifically through *xen/arch/x86/traps.c:do_guest_trap()*. We define a structure that will hold the information we want to log. this structure contains:

- Syscall number
- PID of the process that issued the syscall
- Domain ID of the guest
- Guest CPU ESP register (stack pointer)
- Guest CPU EIP register (program counter)
- Guest CPU extended flags

The syscall number is retrieved in the function *xen/arch/x86/traps.c:do_guest_trap()* as follows:

```
struct cpu_user_regs *user_regs;
user_regs = guest_cpu_user_regs();
number = (uint32_t) user_regs->eax;
```

The other informations are also found within that same function as follows:

```
domain = vcpu->domain->domain_id;
esp = vcpu->arch.guest_context.kernel_sp;
eip = user_regs->eip;
eflags = user_regs->eflags;
```

The difficulty here was to find the correct value for the ESP kernel register. Depending on the context, the value stands for the register as seen by the guest, the hypervisor... We identified empirically the correct value by comparing the one in the guest and the ones within the hypervisor.

Once we have the correct value in the ESP register, we can determine the process PID. Each process has a *thread_info* structure that identifies it. To get the pointer to that structure, we use:

```
thread_info_addr = vcpu->arch.guest_context.
  kernel_sp & 0xFFFFE000;
```

The first member in that structure is a *task_struct* which contains the PID:

```
tmp = (unsigned long *)(thread_info_addr );
get_user(task_struct_addr,tmp);
```

To get the PID:

```
uint32_t pid_offset = 0;
integer_param("pid_offset", pid_offset );
tmp = (unsigned long *)(task_struct_addr
  + pid_offset);
get_user(pid,tmp);
```

The problem here is that the PID offset depends on the kernel version and the OS. There is no generic value that would work in all cases. We modified the hypervisor so that this offset can be passed as a parameter via GRUB:

```
kernel  /boot/xen-3.3.1.gz pid_offset=168
```

That permits to avoid recompiling the hypervisor when we want to trace the syscalls from different PV guests. We will only have to modify the GRUB entry and reboot the computer.

4.1.2 Logging the information

Now that we know how to get all the information, we need to log them out of the hypervisor. To do so, we decided to use the *xentrace* facility. To do so, we need to add a new trace record in *xen/include/public/trace.h*:

```
#define TRC_PV_GUEST_SYSCALL (TRC_PV + 13)
#define TRC_HVM_GUEST_SYSCALL (TRC_PV + 14)
```

Then, in *xen/include/public/trace.h*, we define a structure of at most 28 bytes that will contain the information to log:

```
typedef struct syscall_info {
uint32_t number;
uint32_t pid;
uint32_t tid;
uint32_t domain;
uint32_t esp;
uint32_t eip;
uint32_t eflags;
} syscall_info_t;
```

This structure is filled as described in the previous section. Then, we call *trace_var* to log the structure to Dom0:

```
trace_var(TRC_PV_GUEST_SYSCALL, 1, sizeof(syscall_info_t),
          (unsigned char *)info);
```

In Dom0, we need to capture the exported structure. This is done with the *xentrace* utility. We need to specify the record we want to capture and an output file. Thus, the structures that are exported are saved in a binary file. To capture TRC_PV_GUEST_SYSCALL events:

```
/usr/bin/xentrace -e 0x0020f00d output.trace
```

This output file file can be formated with *xentrace_format*. To do so, we must define a new trace format for the structure we exported:

```
0x0020f00d  CPU%(cpu)d  %(tsc)d (+%(reltsc)8d)  pv_guest_syscall
  [ syscall = %(1)d pid = %(2)d domain = %(4)d esp = 0x%(5)08x
    eip = 0x%(6)08x eflags = 0x%(7)08x ]
```

As output, we obtain for example:

```
CPU0  182759133126366 (+       0) pv_guest_syscall
  [ syscall = 78 pid = 1488 domain = 1 esp = 0xdc0b5ff8
    eip = 0xb6229a41 eflags = 0x00210202 ]
```

In order to make it more human readable, we defined a script *linux_traces.pl*, that uses unistd.h from kernel headers to identify syscall name and parses and outputs the EFLAGS. It is presented in Appendix A. Usage:

```
linux_traces.pl output.trace formats
  /usr/src/linux/include/asm-i386/unistd.h
```

If we keep the previous example, we obtain the following output:

```
CPU0    2570224191823341 (+ 11709)
  SYSCALL:gettimeofday[78] PID:1488
  DOMID:8  ESP:0xdc0b5ff8
  EIP:0xb6229a41 EFLAGS:0x00210202=IF|VIP
```

The result is a text file composed of such entries. By using grep over the PID and DOMID, we can easily filter it and obtain a list of syscalls issued by the monitored process.

4.2 HVM Guest

Syscalls do not trap through the hypervisor, only *SYSENTER/SYSEXIT* instructions do. When *SYSENTER_EIP_MSR* points are set, we must alter them to generate a page fault at a known address. When the page fault takes place, we need to extract the wanted information. all this must be done transparently to the OS.

This solution has been implemented in Ether in Xen 3.1.0. However, due to the lack of documentation and comments in Xen's code, the implementation in Xen 3.3.1 is tough and would be time consuming. Moreover, this implementation would be once again OS and version dependent.

As it would not lead to a generic solution, we did not address this problem at the moment.

5 Conclusion

During this study, we identified 2 mechanisms that would permit to intercept syscalls within the Xen hypervisor. It has been implemented and is working for PV guests.

However, none of these solutions is generic. Both of them are OS and kernel version dependent. Thus, we decided to focus on other solutions for the moment (e.g. other virtualisation systems such as Qemu, using strace/ptrace within the guest directly...).

A linux_traces.pl

```
#!/usr/bin/perl

# Format the ouput of xentrace
#
# Input:
# * binary trace file from xentrace
# * xentrace_format formats
# * header file with the syscall definition
#   e.g. /usr/src/linux-headers-2.6.18-6-xen-vserver-686/include/asm-i386/unistd.h
#
# Ouput: text file with the traces formated

# Variables
$traces_file = $ARGV[0];
$formats = $ARGV[1];
$header = $ARGV[2];

# the file where xentrace_format will save the output
# assuming the traces file has .trace extension
$traces_formated = "/tmp/$traces_file";
$traces_formated =~ s/.trace/.txt/ ;

# the output of the script
$output = $traces_file;
$output =~ s/.trace/_formated.txt/;

# Open the header file, parse it and return a list of syscall names indexed by number
sub parse_header
{
$file = $_[0];

# Open the file
open(HEADER, "<$file") || die("Could not open unistd.h file: $file!");
@lines = <HEADER>;
close(HEADER);

# the array to be returned
@ret = ();
# stores the last syscall number found
$i = 0;
```

```
foreach $line (@lines)
{
# remove trailing
chomp($line);

# only work with #define __NR_* lines
$define = substr($line,0,13);
next if( $define ne "#define __NR_");

# remove #define
$syscall_str = substr($line, 8, length($line)-8);

# split the line
@values = split(' ', $syscall_str);

# if the number an integer or another __NR_* + 1 ?
$first_char = substr($values[1],0,1);


# perl converts automatically string to integer, which is why we add 0
if( $first_char eq "(" )
{
$i++;
}
else
{
$i = $values[1] + 0;
}
$ret[$i] = substr($values[0],5,length($values[0])-5);
}

return @ret;

}

# Based on the EFLAGS value, return a string with the flags set
sub eflags_to_str
{
$reg = oct($_[0]);
$ret = "";
```

```
if( $reg == 0 )
{
return $ret;
}

if( $reg & 0x00000001)
{
$ret .= "CF|";
}
if( $reg & 0x00000004)
{
$ret .= "PF|";
}

if( $reg & 0x00000010)
{
$ret .= "AF|";
}
if( $reg & 0x00000040)
{
$ret .= "ZF|";
}
if( $reg & 0x00000080)
{
$ret .= "SF|";
}

if( $reg & 0x00000100)
{
$ret .= "TF|";
}
if( $reg & 0x00000200)
{
$ret .= "IF|";
}
if( $reg & 0x00000400)
{
$ret .= "DF|";
}
if( $reg & 0x00000800)
{
$ret .= "OF|";
```

```
}

if( $reg & 0x00003000)
{
$ret .= "IOPL|";
}
if( $reg & 0x00004000)
{
$ret .= "NT|";
}

if( $reg & 0x00001000)
{
$ret .= "RF|";
}
if( $reg & 0x00002000)
{
$ret .= "VM|";
}
if( $reg & 0x00004000)
{
$ret .= "AC|";
}
if( $reg & 0x00008000)
{
$ret .= "VIF|";
}

if( $reg & 0x00010000)
{
$ret .= "VIP|";
}
if( $reg & 0x00020000)
{
$ret .= "ID|";
}

$ret_str = substr($ret,0,length($ret)-1);

# remove last '|'
return $ret_str;
```

```
}

# the array with correspondance syscall number to name
@syscalls = parse_header($header);

# format the traces
‘cat $traces_file | xentrace_format $formats > $traces_formated‘;

# Open the traces file
open(FILE, "<$traces_formated") || die("Could not open traces file: $traces_file!");
@lines = <FILE>;
close(FILE);

# Open the output file
open DATAOUT, ">$output" or die "can’t open $output!";

# Parse the traces file and print out ot DATAOUT newly formatted lines
foreach $line(@lines)
{
chomp($line);

# each line is a set of values separated by spaces
@values = split(’ ’,$line);

# if the line concerns a cpu_change, wrap_buffer... skip it
next if( ($values[4] ne "pv_guest_syscall") and ($values[3] ne "pv_guest_syscall") );

if($values[4] eq "pv_guest_syscall")
{
$cpu = $values[0];
$timestamp = $values[1];
$real_timestamp = substr($values[3],0,length($values[3]-1)); # real_timestamp)
$syscall_nb = $values[8];
$i = $syscall_nb + 0;
$syscall_name = $syscalls[$i];
$pid = $values[11];
$domain = $values[14];
$esp = $values[17];
$eip = $values[20];
$eflags = $values[23];
}
else
```

```
{
$cpu = $values[0];
$timestamp = $values[1];
$real_timestamp = substr($values[2],2,length($values[2])-3); #(+real_timestamp)
$syscall_nb = $values[7];
$i = $syscall_nb + 0;
$syscall_name = $syscalls[$i];
$pid = $values[10];
$domain = $values[13];
$esp = $values[16];
$eip = $values[19];
$eflags = $values[22];
}

$eflags_str = eflags_to_str($eflags);

if( length("$syscall_name \[$syscall_nb\]") > 14)
{
print DATAOUT "$cpu\t$timestamp (+ $real_timestamp)\tSYSCALL: $syscall_name
                                \[$syscall_nb\]\tPID: $pid\tDOMID: $domain\tESP: $esp\t
                                EIP: $eip\tEFLAGS: $eflags = $eflags_str\n";
}
else
{
print DATAOUT "$cpu\t$timestamp (+ $real_timestamp)\tSYSCALL: $syscall_name
                                \[$syscall_nb\]\t\tPID: $pid\tDOMID: $domain\tESP: $esp\t
                                EIP: $eip\tEFLAGS: $eflags = $eflags_str\n";
}
}

# close the output file
close(DATAOUT);
```

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803