Linux Kernel Programming

Lab 08: System Calls

AUTHOR
Redha Gouicem and Julien Sopena

Task 1: My first system call

System calls are the communication interfaces provided by the kernel to enable applications to access kernel resources. A system call is defined by its *number* and *parameters*. The kernel maintains a *system call table* that contains a list of function pointers, where the function at index *n* is the handler for *system call number n*.

Most of the time, system calls are executed by applications through APIs provided by system libraries like the C standard library. These libraries provide wrappers for most system calls, e.g., the `write()` function from the libc. When no such wrapper is provided, the standard C library still provides a generic `syscall()` function that takes an arbitrary system call number and parameters. This functions places the system call number and parameters in the proper registers, triggers the system call, and then passes back the return value to the calling program and sets the `errno` variable accordingly. More details in the manual page: `man 2 syscall`.

System calls heavily depend on the underlying architecture. The system call table for *x86_64* (the architecture you are most likely using) is defined in arch/x86/entry/syscalls/syscall_64.tbl while the aarch64 table (for those of you using a recent Apple device) is defined in include/uapi/asm-generic/unistd.h. These tables define, for each system call number, the associated function that handles the system call.

Implementing a system call is done through macros defined in include/linux/syscalls.h:

```
#define SYSCALL_DEFINE0(name)
#define SYSCALL_DEFINE1(name, argtype1, argname1)
#define SYSCALL_DEFINE2(name, argtype1, argname1, argtype2, argname2)
/* up to SYSCALL_DEFINE6 */
```

When expanded, these macros will define a function named `sys_name` with the proper number of parameters, properly typed, with a return value of type long. By convention, a negative return value is a negative error code.

For example, a system call that takes no parameter such as `sync()` will be defined in this way:

```
SYSCALL_DEFINE0(sync)
{
    /* ... */
    return 0;
}
```

A system call with two parameters like `chmod` will be defined in this way:

```
SYSCALL_DEFINE2(chmod, char __user *, filename, umode_t, mode)
{
    /* ... */
    return 0;
}
```

In that case, parameters that are pointers must be manipulated carefully as they point to user space data. They should be handled with functions like `copy_from_user()` or `copy_to_user()`.

Question 1

Open the manual page of the function `syscall`. What is the exact role of this function? Is it a system call? What are its parameters and return value?

Question 2

Write a small user space C program that calls the `kill` system call through the `syscall()` function from the *libc*. You can find the system call number in the `unistd.h` header file on your system or in the kernel system call table.

Question 3

Add a new system call, hello, to your kernel and test it with a small C program. When this system call is executed, it should print `"Hello World!"` into the system logs.

Question 4

Make your user test program and your system call print the PID of the current process. What do you notice?

Question 5

We now want to dynamically change the string printed in the syslog by the system call by adding a parameter `who`. For example, passing the string `"Torvalds"` as a parameter to the system call should make the following syslog print: `"Hello Torvalds!"`.

Modify your system call to enable this feature and test it.

Question 6

As a great uncle once said, *"With great power comes great responsibility"*. Your system call might have just introduced a security breach. To check this out, try to pass as a parameter to your system call the address obtained by loading the one_addr.ko module provided in this exercise archive.

What's happening? How can this breach be removed?

Question 7

Now, instead of printing the message in the syslog, we want to return the string to the calling user space program. Modify your system call to take an additional parameter pointing to a user space buffer where the string will be copied. Your system call should also return the size of the string, or a negative value in case of an error (with the error code properly encoded).

Task 2: Re-routing a signal (rootkit)

In this exercise, we will attempt to intercept signals sent to hidden processes using the previous lab practical. The techniques used will allow us to study: the operation of a system call in Linux, the role of the system call table, and the kernel symbol table.

Question 1

After launching a command `sleep 9000 &`, send it a `SIGCONT` signal and then repeat the operation after masking it. By comparing your observations to the result of sending the same signal to a non-existent process, propose a method to list hidden processes.

Question 2:

Implement a small shell script that displays the *pid* of all hidden processes whose *pid* is between two numbers passed as parameters.

Question 3:

To counter this detection, we will intercept the signal at the system call level. Run the strace command on a kill. Which system calls should be intercepted?

Question 4:

All system calls go through interrupt int 0x80. To differentiate them, they are associated with a number (*system call number*). Once switched to kernel mode, this number is used to search for the system call address in the system call table. Intercepting a system call involves replacing the original address in the system call table with that of your function. This method requires knowing the start of the table marked by the symbol sys_call_table.

Unfortunately, since version 2.6, this symbol is no longer exported. To find it, we will use the `System.map` via the `/proc/kallsyms` file (which is not always present) to find the address of the system call table. What is this address? In which segment is it located?

> **Caution**
>
> If you only see null values, it means you are not root. Indeed, this file is part of a pseudo file system and is calculated differently if the read is done by the administrator or by a regular user.

Question 5

Now that you have the address of the table, try replacing the address of the first system call with NULL in a module that takes the address of the table as a parameter. What happens?

Question 6

In fact, the system call table is located on a write-protected page. Our code, running in kernel mode, should still be able to modify it, but Intel has added a protection mechanism based on the wp bit of the control register (*cr0*). Check for the presence of this mechanism by consulting `/proc/cpuinfo`.

Question 7

Knowing that the wp bit is bit 16 of the *cr0* register, use the `read_cr0` and `write_cr0` functions to regain your rights and test your new powers. Make sure to restore the register after modifying the system call table.

Question 8

You can now modify the table to intercept the `sys_kill` with a function that returns `-ESRCH` if the pid matches the process to hide.

> **Caution**
>
> Your module must maintain the normal operation of signals for other processes. Additionally, you must restore the original behavior upon unloading the module.