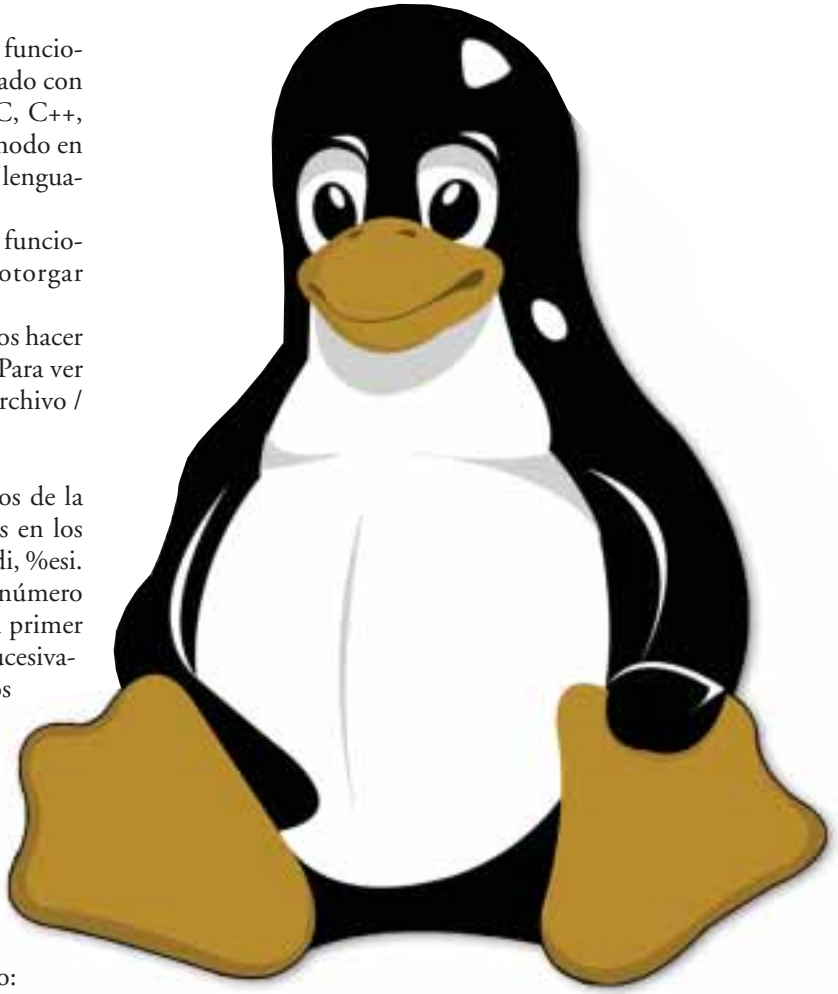
Ensamblador básico I: Syscalls bajo Linux

Los syscalls bajo Linux se utilizan para programar funciones básicas a través del kernell. Si estás familiarizado con algún lenguaje de programación de alto nivel como C, C++, PASCAL, etcétera, entonces seguramente conoces el modo en que las *funciones* (procedimientos) trabajan bajo estos lenguajes.

Del mismo modo trabajan las *syscalls*, porque son funciones del sistema operativo a las cuales debemos otorgar parámetros (argumentos) antes de ejecutarlas.

Bajo lenguajes de alto nivel lo único que necesitamos hacer es llamar a la función con los parámetros apropiados. Para ver una lista de los syscalls de tu OS puedes consultar el archivo /usr/incluye/asm/uniste.h



MODO DE EMPLEO

Te estarás preguntando cómo pasar los argumentos de la función. Dichos argumentos deberemos almacenarlos en los registros de uso general: %eax, %ebx, %ecx, %edx, %edi, %esi.

El registro %eax está reservado para introducir el número de syscall que vayamos a usar; el %ebx corresponde al primer argumento de la función; %ecx al segundo, y así sucesivamente. Puede haber syscalls que empleen uno o varios argumentos, todo dependerá del syscall que vayamos a usar.

Una vez que hemos pasado los argumentos apropiadamente a la función sólo falta ejecutarla, y esto lo logramos mediante el interruptor int $0x80.

Vamos a tomar por ejemplo la syscall "write" que tiene el número 4 reservado. Al consultar man 2 write podemos observar el prototipo de la función en lenguaje C y una breve descripción de su funcionamiento:

```
NAME
       write(1, 2) - write(1,2) to a file(1,n) descriptor

SYNOPSIS
       #include <unistd.h>

.......ssize_t write(1,2)(int fd, const void *buf, size_t count);

DESCRIPTION
       write(1,2)  writes  up  to count  bytes  to  the  file(1,n) referenced by the file(1,n)
       descriptor fd from the buffer starting at buf.  POSIX  requires  that  a
       read(2,n,1 builtins)()  wich  can  be  proved  to ocurr  after a write(1,2)() has returned
       returns the new data.  Note that not  all file(1,n) systems  are  POSIX  conforming.
```

PARA SABER MÁS

Puedes consultar las páginas man ejecutando man 2 NOMBRE_FUNCION

El primer argumento, int fd, corresponde al archivo donde vamos a imprimir. Normalmente necesitaremos mostrar los resultados en pantalla, por lo que "/dev/stdout" es el candidato obvio. Las salidas estándar del sistema están numeradas, si ejecutamos "ls –al /dev/stdout" podemos ver que es un link que apunta a un proceso dentro de /proc (dependiendo de tu OS) que generalmente es "1". En mi sistema puedo comprobarlo mediante:

```
benn $ echo "STDOUT – Imprime en pantalla" > /proc/self/fd/1
STDOUT – Imprime en pantalla
benn $
```

Así que deberíamos pasar $1 a %ebx para así poder imprimir en pantalla.

Ahora debemos decidir qué es lo que queremos imprimir. Siguiendo el método anterior podemos deducir que la variable "const void * buf" corresponde a %ecx.

Supongamos que deseamos imprimir la cadena "hola", lo que debemos hacer es empujar tal cadena al top del stack (%esp) y luego pasarle la dirección de memoria a %ecx para que apunte al top del stack (%esp).

Esto nos limita a cuatro caracteres, si deseamos usar más debemos alojar más memoria declarando una cadena, pero esto no corresponde a este artículo, será tratado en entregas posteriores, por lo que ahora usaremos sólo cuatro caracteres.

Sabemos de antemano los valores de cada letra:

h=0x68
o=0x6f
l=0x6c
a=0x61

Procedemos a empujarlos en forma inversa (LIFO: last in, first out), y luego apuntar %ecx al top del stack:

push $0x616c6f68
mov %esp,%ecx

Ahora, el último argumento que nos queda es size_t num que corresponde a %edx y es el número de bytes que ocupa el buffer size_t num (sizeof(buf)), en este caso son cuatro bytes, por lo que la instrucción apropiada luce así:

mov $4,%edx

Y, finalmente, para imprimir en pantalla llamamos a la interrupción para que ejecute la syscall:

int $0x80

En resumen, los argumentos de nuestra función write() son:

```
%eax = $4                    #Referencia a syscall write()
%ebx = $1                    #Recuerda, stdout = 1
%ecx = 0x616c6f68            #Cadena "hola"
%ecx = $4                    #Número de caracteres
```

Aquí no termina esto, ya que para evitar una violación de segmento debida a los file descriptors aún abiertos debemos llamar a la syscall exit () para finalizar el programa correctamente. Esta syscall exit() tiene número reservado "1", y su prototipo es:

```
NAME
       _exit, _Exit – terminate the current process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);

DESCRIPTION
    The function _exit terminates the calling process
"immediately". Any open file descriptors belonging to the
process are closed; any children of the process are inherited
by process 1, init, and the process's parent is sent a
SIGCHLD signal.

    The value status is returned to the parent process
as the process's exit status, and can be collected
using one of the wait family of calls.
```

Código fuente
Equivalente en C

Equivalente en ASM

```
.globl main
main:
mov $4,%eax                  # write()
mov $1,%ebx                  # stdout
push $0x616c6f68             # "hola"
mov %esp,%ecx
mov $4,%edx                  # 4 bytes
int $0x80                    # interruptor
mov $1,%eax                  # exit
mov $0,%ebx                  # statur = OK
int $0x80                    # interruptor
```

Puedes compilarlo con gcc write.s –o write ; /write

En la próxima entrega ampliaremos estos conocimientos y los utilizaremos para crear shellcodes que podremos incluir en nuestros futuros exploits. ¡Hasta la próxima!

End of file.