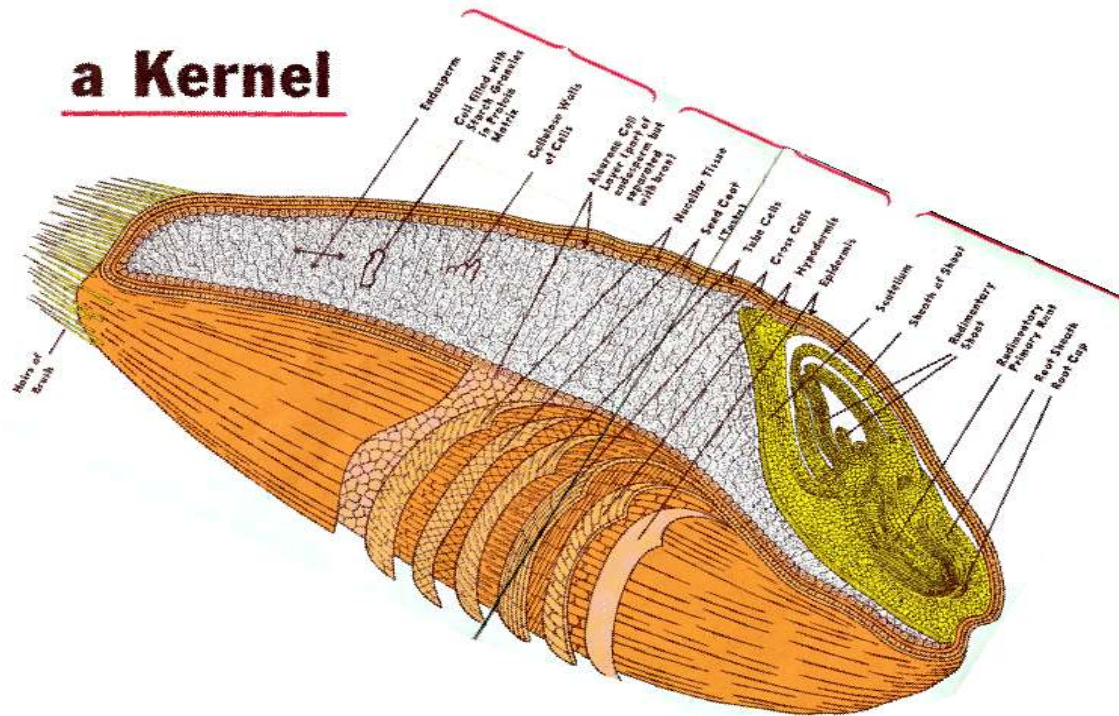


Linux 2.6 Kernel Hacking



Copyright,
all rights reversed

txipi@sindominio.net
hackAndalus'2004

¿De qué vamos a hablar?

- Algunas generalidades sobre Linux.
- Conceptos básicos de un Sistema Operativo
- Programación de LKMs en Linux 2.4
- LKMs en Linux 2.6
 - Problemas
 - Soluciones
- Técnicas alternativas: Adore

Algunas generalidades

Arquitecturas:

- Linus Torvalds lo inició pensando exclusivamente en el 80386 a principios de los 90.
- Actualmente soporta 17 arquitecturas diferentes: Intel, Motorola, SPARC, Alpha, IBM s/390...

```
$ ls /usr/src/linux/arch/  
alpha  cris  ia64  mips  
parisc  ppc64  s390x  sparc  
x86_64  arm   i386  m68k  
mips64  ppc   s390  sh  
sparc64
```

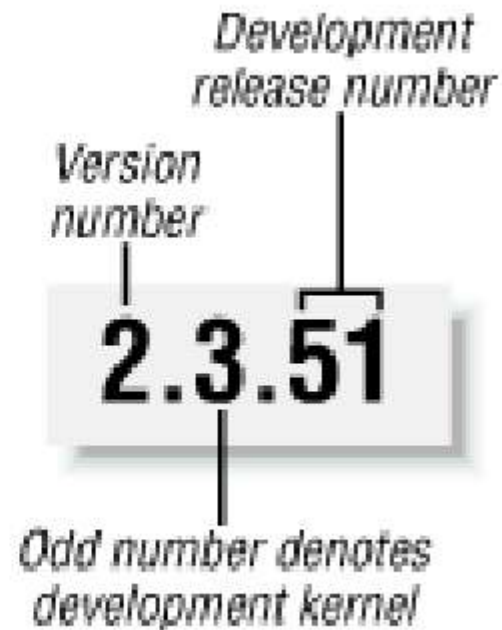
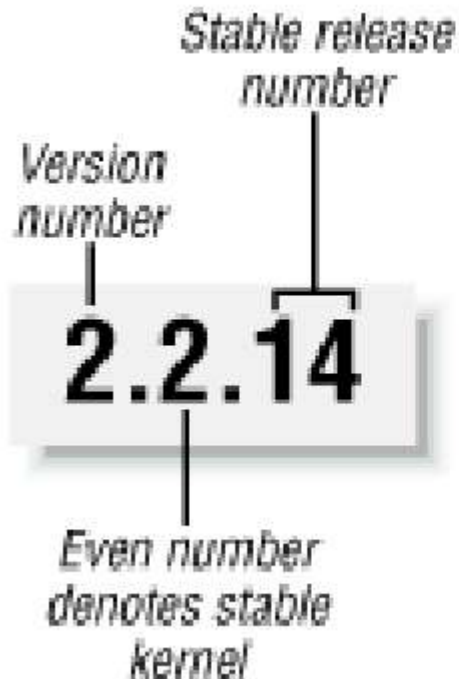
Algunas generalidades

Cifras:

- 12.000 ficheros, 5.000.000 de líneas de código.
 - drivers: 3.200 ficheros, 2.300.000 líneas de código.
 - arch: 2.900 ficheros, 990,000 líneas de código.
 - filesystems: 660 ficheros, 330.000 líneas de código.
 - networking: 450 ficheros, 230.000 líneas de código.
 - core: 100 ficheros, 46.000 líneas de código.

Algunas generalidades

Versiones:



Algunas generalidades

- Linux es un kernel monolítico
 - Al igual que la mayoría de UNIX (Mach y derivados son notables excepciones).
 - Compilado estáticamente
 - Aunque puede dar soporte para módulos cargables (LKMs), muy útiles para drivers.
- Multithreading, LightWeight Processes, **clone()**, hyperthreading...
- SMP: hasta 64 procesadores por ahora.
- No pensado para RT funciona bastante bien en entornos soft-RT. Existen parches.

Algunas generalidades

- El kernel está escrito en GNU C y inline assembler (el código dependiente de la arquitectura).
- Utiliza un espacio de direccionamiento global, sin protección de memoria, si estás en el kernel, puedes destruirlo todo

```
int *foo = NULL;  
*foo = 1; /* BOOM! */
```

Algunas generalidades

- Dentro del kernel no podemos usar las llamadas a libC, sólo unas pocas están disponibles (malloc(), strtoul()...).

- No se puede usar coma flotante o MMX:

```
int x = foo * 2.5; /* BOOM! */
```

- Espacio limitado en la pila (en algunas arquitecturas es compartido con interrupciones)

```
char buffer[4096]; /* BOOM! */
```

Algunas generalidades

- Independencia de plataforma:
 - 64-bit clean
 - endian-clean
 - char-signed clean

```
int foo = (int)&ptr; /* BOOM! */  
char *firstbyte = (char *)&foo; /* BOOM! */  
if (*firstbyte < 0) /* BOOM! */
```

Conceptos básicos de un S.O.

Modo real vs. modo protegido:

- DOS: acceso directo al hardware, memoria “real”, pocas protecciones, <32-bit.
- Linux: protección de memoria, de procesos, separación UserLand/KernelLand, >=32-bit

Conceptos básicos de un S.O.

UserLand vs. Kernelland:

- En S.O. “decente” los procesos del núcleo no se ejecutan en el mismo entorno que los procesos de usuario.
- Si falla un proceso de usuario, puede afectar a otros procesos de usuario, pero no al núcleo.

Bootstrap, el génesis

Bootstrap:

- Proceso de inicialización del Sistema Operativo.
- Es una tarea larga y tediosa: todos los periféricos están en un estado inicial impredecible.
- Es una tarea MUY dependiente de la arquitectura.

Bootstrap

La prehistoria: BIOS

BIOS: Basic Input/Output System.

- RESET a la CPU → **EIP=0xfffffffff0**
- BIOS (ROM)
 - POST (Power On Self Test)
 - Devices, PCI + IRQs
 - Buscar un Sistema Operativo
 - Cargar el primer sector en RAM (**0x000fc00**)
- Boot loader

Bootstrap

La grexia clásica: Boot loader

- Si estamos arrancando desde diskette:
 - El kernel de Linux está preparado para esto, y guarda **arch/i386/boot/bootsect.S** en los primeros bytes de su código.
- Si estamos arrancando desde disco duro:
 - Gestor de arranque → LILO
 - En el MBR (Master Boot Record)
 - En el sector de arranque de una partición (activa)

Bootstrap

LILO (Linux LOader)

- Está separado en dos partes (no cabe en el primer sector):
 - Un pequeño cargador que reemplaza a un boot loader sencillo.
 - Un programa que lee un mapa de las particiones que contienen SOs y muestra un prompt interactivo.
- Una vez elegido el SO, se carga el sector de arranque de dicho SO en memoria y se ejecuta.

Bootstrap

Boot loader:

- Se mueve a sí mismo desde **0x00007c00** a **0x00009000**.
- Define la pila del Modo Real (**0x0003ff4**)
- Usa la BIOS para sacar el mensaje “Loading...”
- Carga el kernel en memoria.
- Salta a **setup()**

Bootstrap

La Edad Media: **setup ()**

- Pregunta a la BIOS la cantidad de RAM de la que dispone el sistema.
- Reinicializa dispositivos (teclado, vga, discos, etc.), no se fía de la BIOS.
- Define una IDT (Interruption Descriptor Table) y una GDT (Global Descriptor Table) provisional.
- Salta a Modo Protegido (PE en `cr0 = 1`).
- Ejecuta la función **startup_32 ()**.

Bootstrap

El renacimiento: **startup_32()**

- En realidad hay dos funciones **startup_32()**:
 - La primera descomprime el kernel (“Uncompressing Linux...”, “OK booting the kernel”).
 - La segunda inicializa la IDT, los registros `gdtr` y `ldtr`, la pila del kernel y salta a la función **start_kernel()**.

Bootstrap

La Edad Moderna: `start_kernel()`

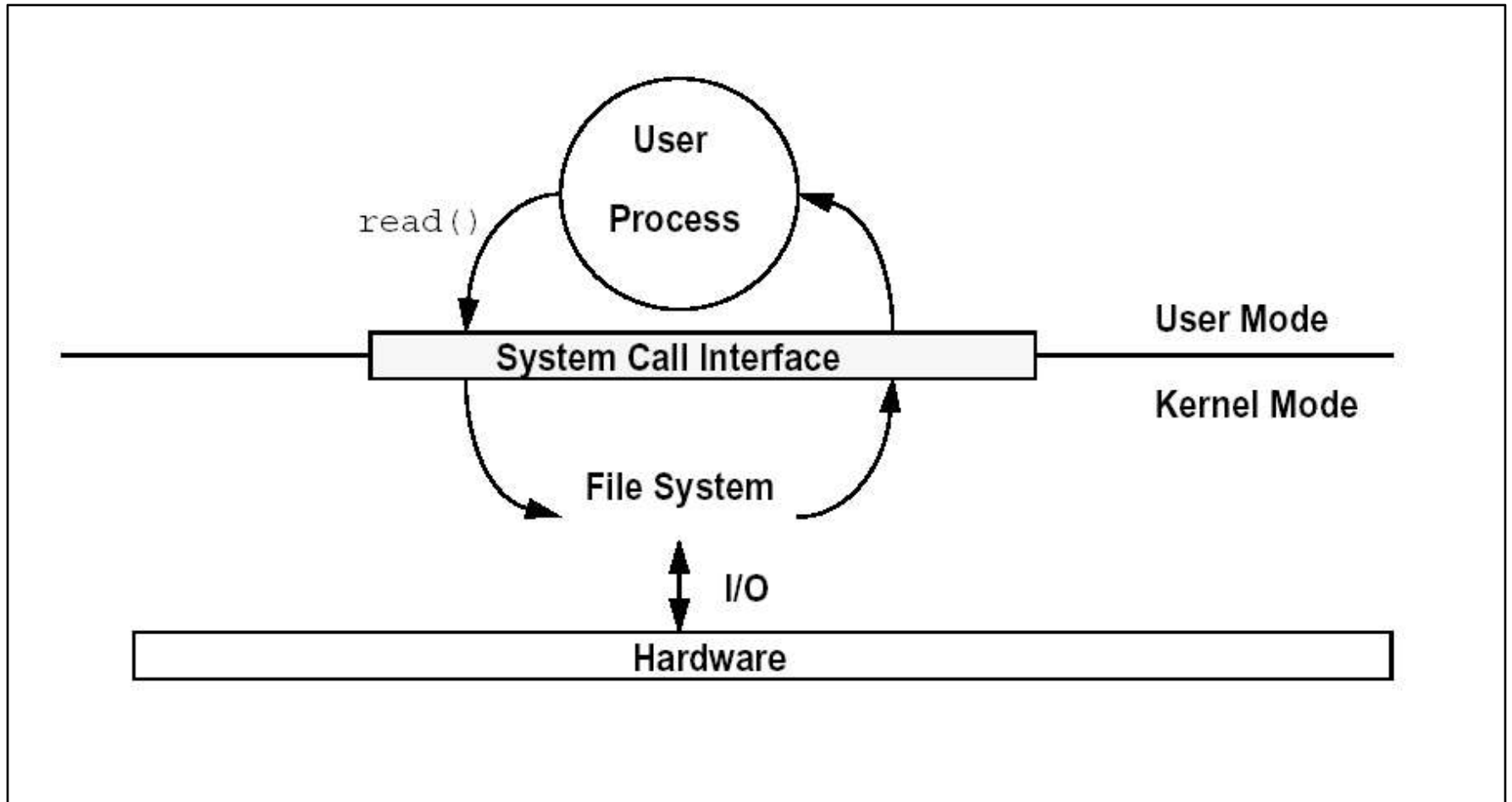
- En esta función se inicializa prácticamente todo lo que tiene que ver con el kernel:
 - `paging_init()`
 - `init_IRQ()`
 - `kmem_cache_init()`
 - ...
- Se crea el kernel thread para el proceso 1, y éste crea el resto de kernel threads.
- Se carga para su ejecución `/sbin/init`.

Syscalls

Syscall:

- Petición al Sistema:
 - Petición: ningún proceso de usuario puede obligar al kernel (Sistema).
- Similares a un desvío o “trap”, pero de forma sincrónica:
 - Una page_fault no sabes cuándo se producirá.
 - Una llamada a write, sí.

Syscalls

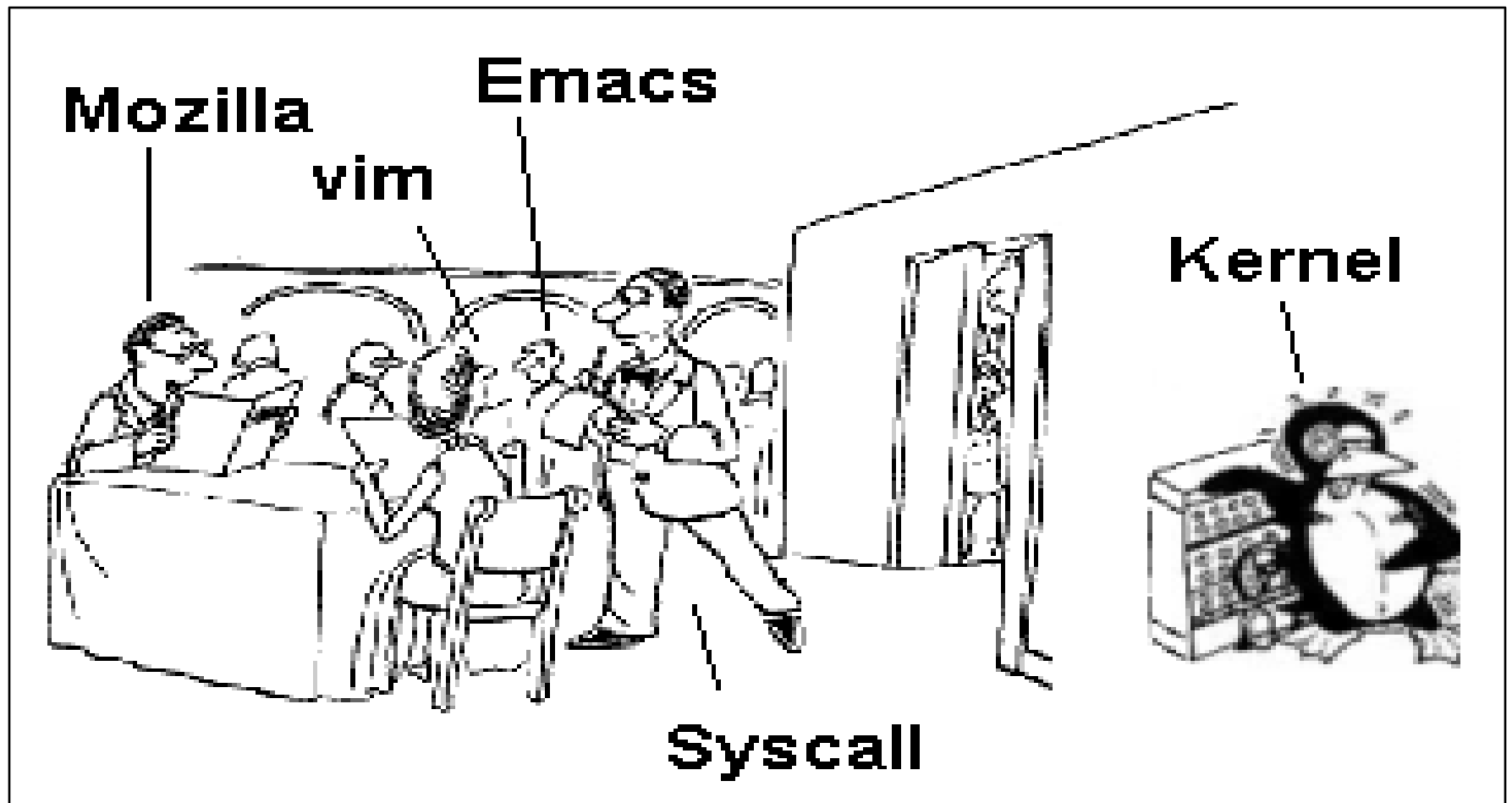


Syscalls

Símil del restaurante:

- Comensales: procesos.
- Camarero: mecanismo de syscalls.
- Cocina: kernel.
- El comensal pide al camarero algo interesante de la cocina:
 - “Excelente elección”.
 - “Lo siento, hoy no tenemos lenguado”.
- El proceso pide al mecanismo de syscalls algo interesante del kernel.

Syscalls



Syscalls

¿Syscalls = API? No:

- Una API call puede ser sólo UserLand:
 - Una función matemática: `abs(-23556)`;
- Una API call puede involucrar a varias syscalls o incluso funcionar como wrapper:
 - `malloc()`, `calloc()`, `free()` → `brk`
- `strace` vs `ltrace`

Programación de LKMs

¿Qué es un LKM?

- Un LKM es un Loadable Kernel Modules, es decir, un Módulo (des)Cargable del Núcleo.
- Útiles para:
 - No recompilar cada vez que queremos cambiar algo en el núcleo (flexibilidad).
 - Optimizar el uso de la memoria del núcleo.
 - Modificar el comportamiento del sistema (estamos en RING-0 :-9'").

Programación de LKMs

Hello, kernel!

```
#define __KERNEL__
#define MODULE
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
int init_module() {
    printk("Hello, kernel!\n");
    return 0;
}
void cleanup_module() {
    printk("Bye, bye, RING-0!\n");
}
```

Programación de LKMs

Compilación:

```
gcc -O2 -c -I/usr/src/linux/include hello.c
```

Carga y descarga:

```
insmod hello.o
```

```
lsmod | grep hello
```

```
rmmod hello
```

Programación de LKMs

Paso de parámetros:

```
char letra; short intcorto; int entero;
int array[4]; long intlargo; char *string;
MODULE_PARM(letra, "b");
MODULE_PARM(intcorto, "h");
MODULE_PARM(entero, "i");
MODULE_PARM(array, "1-4i");
MODULE_PARM(intlargo, "l");
MODULE_PARM(string, "s");
```

Programación de LKMs

Macros:

```
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("txipi");
```

```
MODULE_DESCRIPTION("Modulo de ejemplo");
```

```
int i;
```

```
MODULE_PARM(entero, "i");
```

```
MODULE_PARM_DESC(i, "Entero que leo");
```

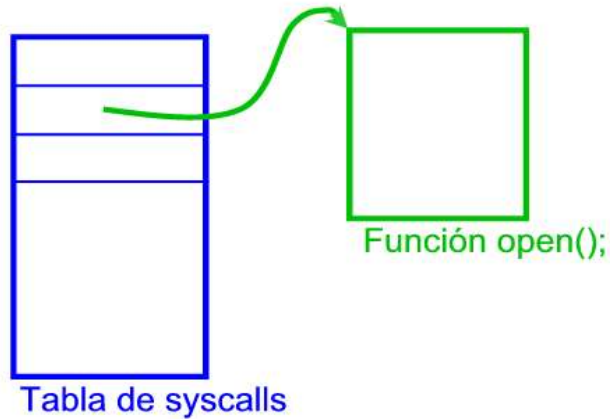
Programación de LKMs

La tabla de syscalls (Linux 2.4):

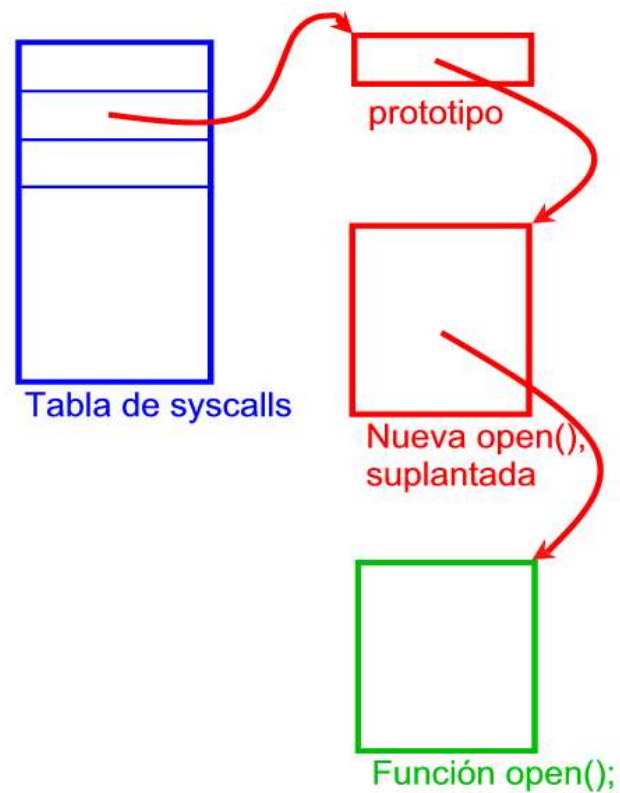
```
#include <asm/unistd.h> /* __NR_syscall */  
extern void *sys_call_table[];  
  
int (*old_getuid)();  
  
old_getuid = sys_call_table[__NR_getuid];  
sys_call_table[__NR_getuid] = my_getuid;
```

Suplantación de syscalls

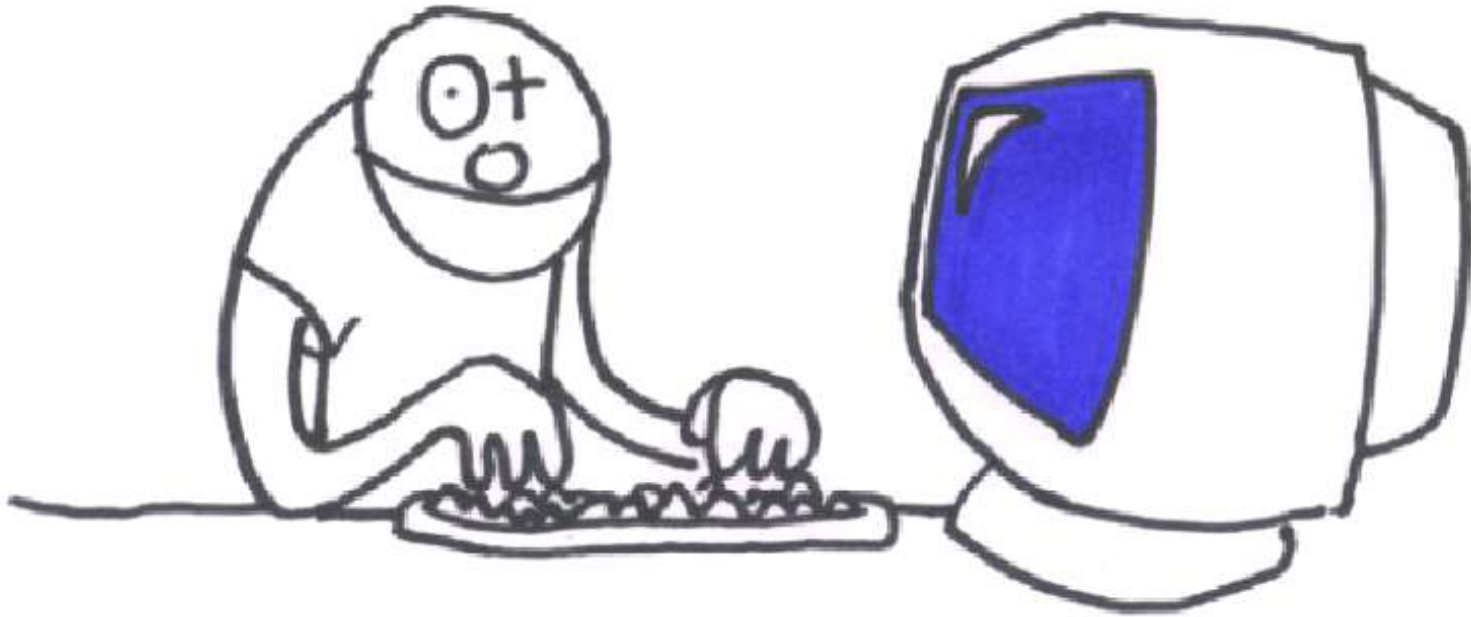
Situación inicial...



Situación final...



Programación de LKMs



Ejemplos:

- Suplantación de syscalls

LKMs en Linux 2.6

Mejoras en cuanto a seguridad:

- Perfiles de seguridad
- ¡Ya no se exporta la `sys_call_table`!
 - Todos nuestros ejemplos no valen para nada :-)
 - Vamos a repasar como se implementan las syscalls para ver si encontramos algún hueco ;-)

Syscalls, implementación

Inicialización:

- **trap_init()** inicializa la entrada en la IDT al arrancar el kernel:

```
set_system_gate(0x80, &system_call);
```

(apunta la entrada 128 (0x80) a la función

```
system_call() )
```

Syscalls, implementación

La función `system_call()`:

```
system_call:
    pushl %eax
    SAVE_ALL
    movl %esp, %ebx
    andl $0xffffe000, %ebx ; ebx = current
    cmpl $(NR_syscalls), %eax
    jnb nobadsys
    movl $(-ENOSYS), 24(%esp)
    jmp ret_from_sys_call
nobadsys:
```

Syscalls, implementación

La función `system_call()`: (II)

```
¿PF_TRACESYS? syscall_trace()
```

```
call *sys_call_table(0, %eax, 4)
```

```
ret_from_sys_call( ):
```

```
    movl %eax, 24(%esp)
```

```
    jmp ret_from_sys_call
```

LKMs en Linux 2.6

Obtener la `sys_call_table`:

- Seguimos todo el proceso que hace el sistema hasta llegar a `system_call()`.
- Dentro de `system_call()` buscamos la

```
call *sys_call_table(0, %eax, 4)
```

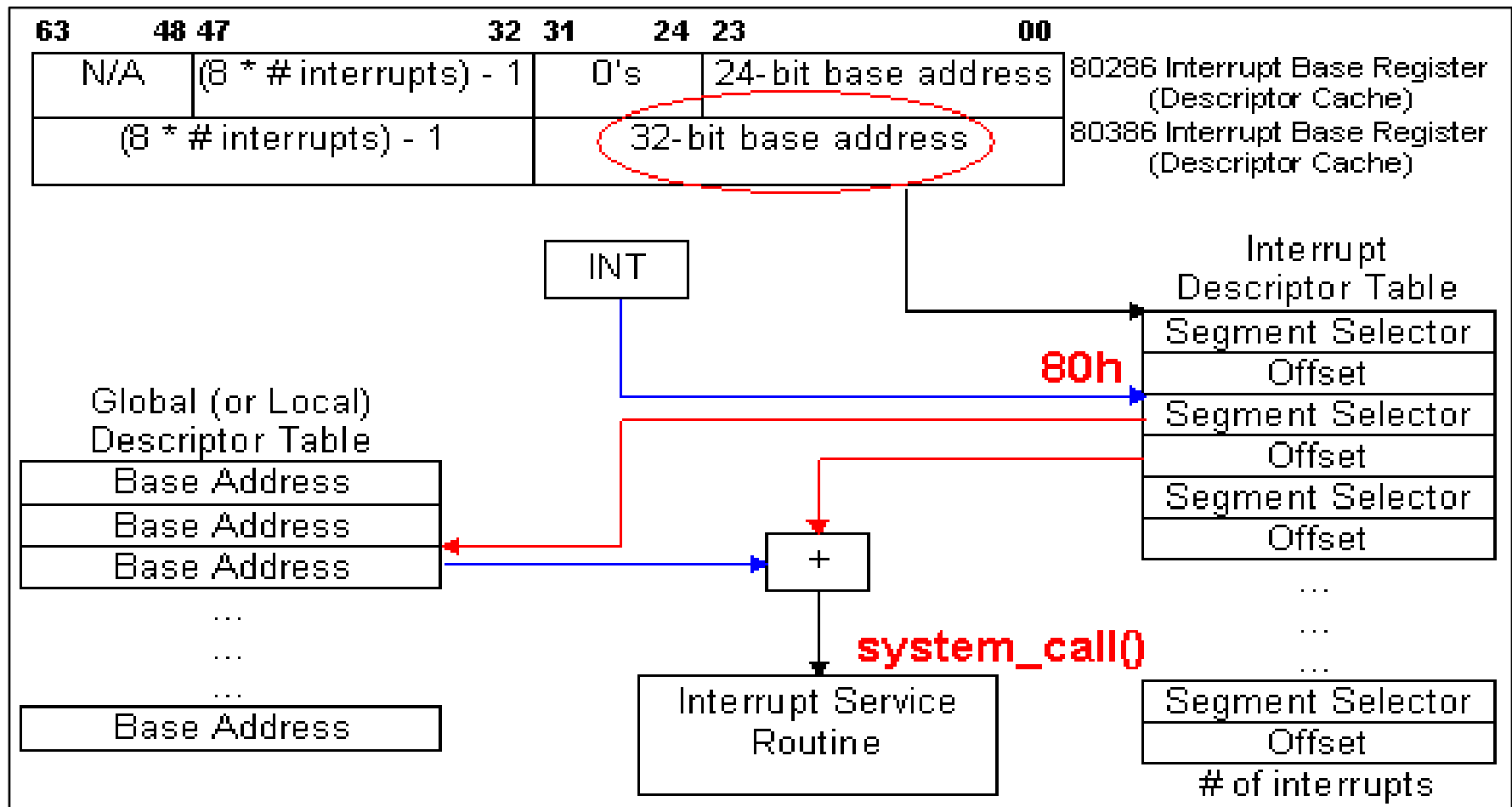
- Así conseguimos la dirección base de la `sys_call_table`.

LKMs en Linux 2.6

Obtener la `sys_call_table`:

- Obtenemos el valor del IDTR
- Con la parte de la base del IDTR, tenemos la base de la IDT
- `IDT[80h] = system_call()`
- Dentro de `system_call()` buscamos una call (`0x8514FF...`).
- Lo siguiente a esa call es -presumiblemente- la dirección de la `sys_call_table`.

LKMs en Linux 2.6



LKMs en Linux 2.6

Modificar los ejemplos para que funcionen en Linux 2.6:

- Creamos una función `get_sys_call_table()` que haga todo lo anterior.
- El puntero a la `sys_call_table` ya no es un símbolo externo.
- Usamos nuestro puntero como antes.
- Utilizamos la nueva sintaxis para LKMs en Linux 2.6.

LKMs en Linux 2.6

Hello, kernel 2.6!

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>
static int __init init_routine(void) {
    printk("Hello, kernel 2.6!\n");
    return 0;
}
static void __exit cleanup_routine(void) {
    printk("Bye, bye, RING-0!\n");
}
module_init(init_routine);
module_exit(cleanup_routine);
```

LKMs en Linux 2.6

Compilación en Linux 2.6:

- Necesitamos crear un Makefile con el siguiente contenido:

```
obj-m := modulo.o
```

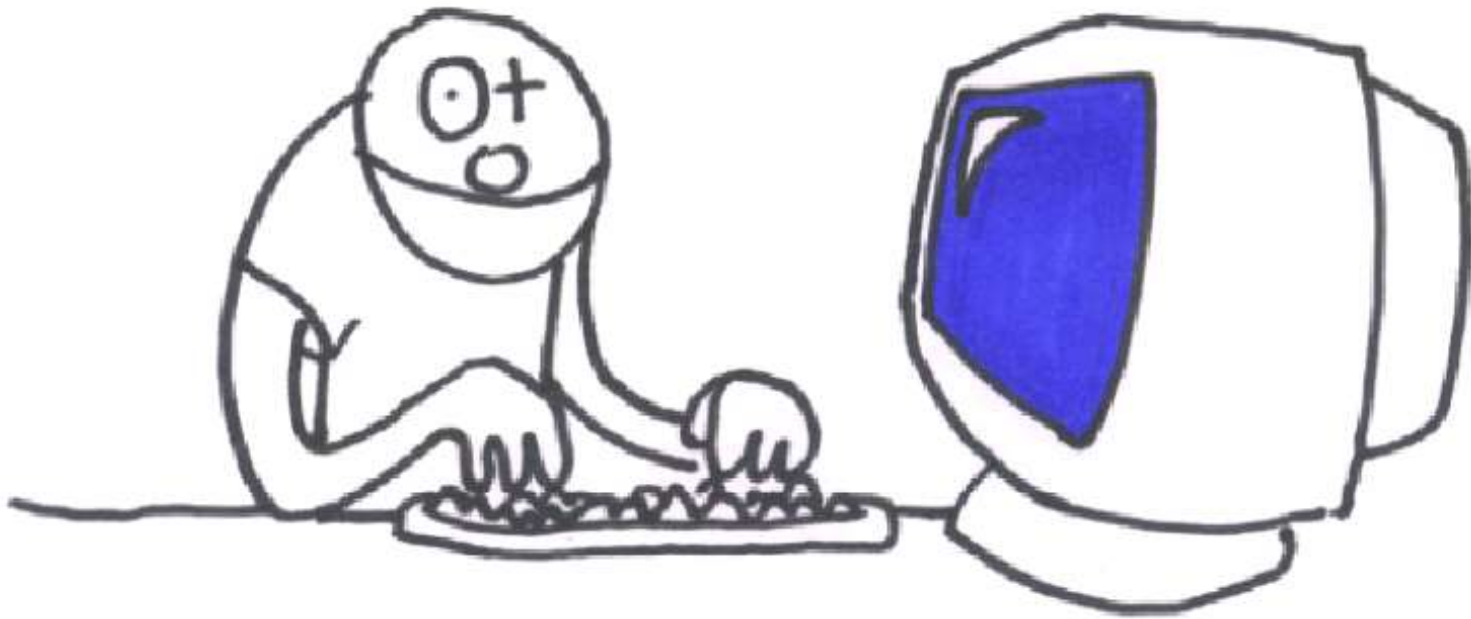
- Luego lo compilamos así:

```
#!/bin/sh
```

```
make -C /usr/src/linux SUBDIRS=$(pwd) modules
```

```
make -C /usr/src/linux SUBDIRS=$(pwd) modules_install
```

LKMs en Linux 2.6



Ejemplos:

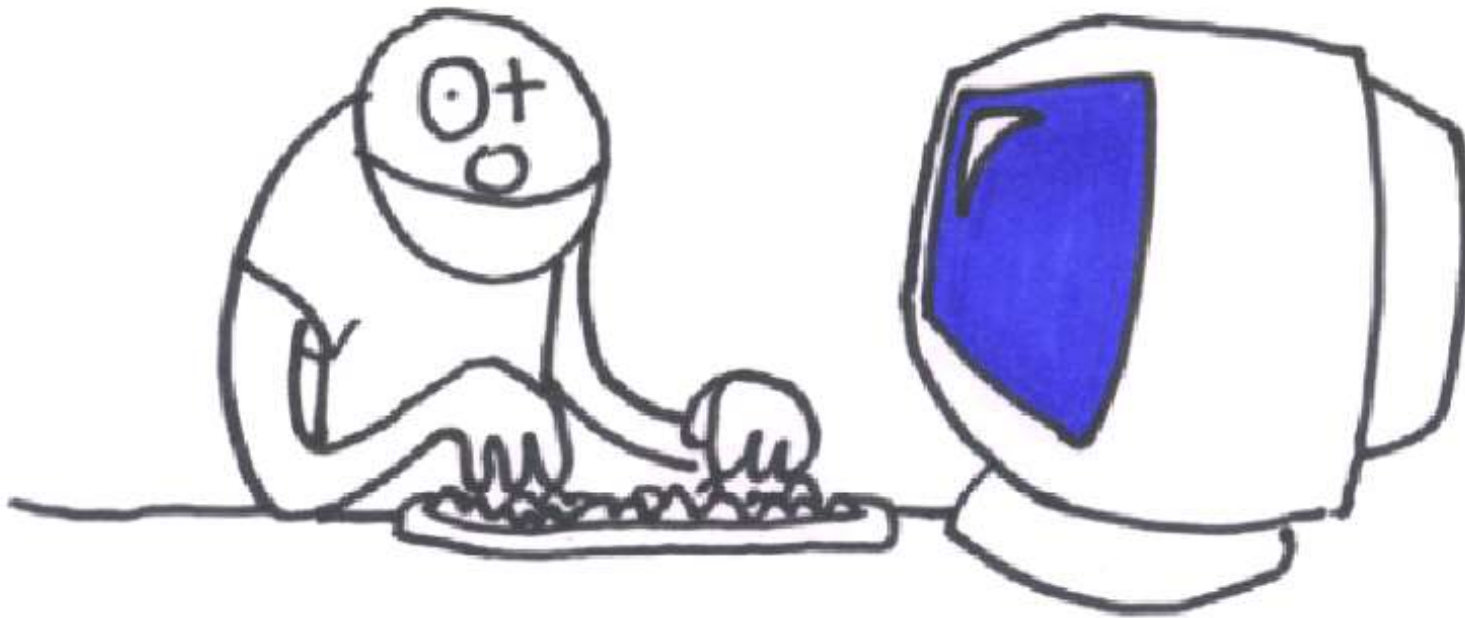
- Conseguir la `sys_call_table` en Linux 2.6

Técnicas alternativas

Adore LKM (Teso Team):

- No utiliza la `sys_call_table` para ocultar ficheros, procesos y sockets abiertos, sino que lo hace a través de la capa VFS.
- No utiliza la lista enlazada de módulos para ocultarse, sino que emplea un cliente en UserLand específico para conectarse con el LKM en KernelLand (una especie de “mando a distancia”).

LKMs en Linux 2.6



Ejemplos:

- Adore y ava

Agradecimientos

A Ripe, Doing, Teso-team, etc. por todos los conocimientos técnicos.

A ender por haberme dejado el portátil para preparar la charla (y por casi 20 años de colegueo ;-P).

A toda la gente que ha hecho posible el hackmeeting