

LINUX - LKM

Plan wykładu

- Wprowadzenie
- Moduły dla jądra ≥ 2.6
- Przechwytywanie odwołań systemowych
- Architektura LSM

Nitesh Dhanjani, Justin Clarke **Network Security Tools**

Peter J. Salzman, Ori Pomerantz **The Linux Kernel Module Programming Guide**

Wprowadzenie

- **Wprowadzenie**
- Moduły dla jądra ≥ 2.6
- Przechwytywanie odwołań systemowych
- Architektura LSM

LKM

Ładowalne moduły jądra **LKM** (*ang. Loadable Kernel Module*) są dynamicznie ładowanymi rozszerzeniami jądra.

- wprowadzone od Linux 1.2 (1995)
- działają w przestrzeni jądra (a nie użytkownika!)
- używane są m.in. do:
 - tworzenia sterowników urządzeń
 - tworzenia sterowników systemów plików
 - tworzenia sterowników sieciowych
 - przechwytywania lub dodawania nowych funkcji systemowych
 - pisania interpreterów niskiego poziomu
- Framework LSM

Funkcja `printk()`

```
printk( KERN_ALERT "modul: parametr == %d\n", parametr);
```

Dostępne jest 8 priorytetów, które są zdefiniowane w pliku `<linux/kernel.h>` rozwijają się do napisu "`<nr>`". Są to:

KERN_EMERG - zazwyczaj używany do komunikatów poprzedzających awarię systemu (najbardziej istotny komunikat - definiowany jako "`<0>`")

KERN_ALERT - sytuacje wymagające natychmiastowej uwagi.

KERN_CRIT - sytuacje krytyczne, najczęściej związane z poważnym błędem sprzętu lub oprogramowania.

KERN_ERR - błąd; sterowniki urządzeń używają tego do powiadamiania o problemie ze sprzętem

KERN_WARNING - ostrzeżenia o problemach, które nie powodują awarii całego systemu

KERN_NOTICE - sytuacje, które nie są błędami, ale warto je zasygnalizować (np. komunikaty o bezpieczeństwie)

KERN_INFO - komunikaty informacyjne (np. sterowniki wypisują w ten sposób dane o znalezionym sprzęcie)

KERN_DEBUG - używane do debugowania (najmniej istotny komunikat - definiowany jako "`<7>`")

Komunikaty i logi

- logami zajmuje się demon **syslogd**, umieszcza je w buforze kołowym dostępnym przez **/proc/kmsg** lub polecenie **dmesg** (opcje **-nlevel**, **-c**)
- demon **klogd** (/etc/init.d/klogd) pobiera dane od **syslog** i umieszcza je w **/var/log/messages** (opcja **-fname** kieruje logi do innego pliku)
- za pomocą pliku **/proc/sys/kernel/printk** mamy dostęp do ustawień konsoli związanych z logami
 - **console_loglevel**
 - **default_message_loglevel -> KERN_WARNING**
 - **minimum_console_level**
 - **default_console_loglevel**

```
echo "4 4" > /proc/sys/kernel/printk
```

Moduły dla jądra ≥ 2.6

- Wprowadzenie
- **Moduły dla jądra ≥ 2.6**
- Przechwytywanie odwołań systemowych
- Architektura LSM

Nagłówki jądra

- Do kompilacji modułów jądra niezbędne są pliki nagłówkowe źródeł aktualnie zainstalowanego jądra.

Przykładowo dla dystrybucji opartych na Debianie instalacja wymaganych pakietów może wyglądać następująco:

```
$ sudo apt-get install build-essential  
$ sudo apt-get install linux-headers-$(uname -r)
```

lub:

```
$ sudo -i  
# apt-get install module-assistant  
# m-a prepare
```


Przykład kodu

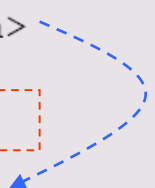
hello-1.c

```
/*
 * hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>          /* Needed by all modules */
#include <linux/kernel.h>          /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Hello world 1.\n");

    /*
     * A non 0 return means init_module failed; module can't be loaded.
     */
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye world 1.\n");
}
```



Kompilacja modułu

Makefile

```
obj-m += hello-1.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```



```
hostname:~/lkmpg-examples/02-HelloWorld# make
make -C /lib/modules/2.6.11/build M=/root/lkmpg-examples/02-HelloWorld modules
make[1]: Entering directory `/usr/src/linux-2.6.11'
  CC [M]  /root/lkmpg-examples/02-HelloWorld/hello-1.o
Building modules, stage 2.
MODPOST
  CC      /root/lkmpg-examples/02-HelloWorld/hello-1.mod.o
  LD [M]  /root/lkmpg-examples/02-HelloWorld/hello-1.ko
make[1]: Leaving directory `/usr/src/linux-2.6.11'
hostname:~/lkmpg-examples/02-HelloWorld#
```

Ładowanie i rozładowanie modułu

insmod [opcje] nazwa_modulu.ko [parametry]

Ładuje podany moduł do jądra. Jeśli są podane parametry, to przekazuje je modułowi. Najważniejsze opcje:

-o **nazwa** Ustala nazwę dla modułu na **nazwa** zamiast przyjąć nazwę pliku bez rozszerzenia .ko (lub .o), co jest zachowaniem domyślnym

-p Symuluje załadowanie modułu, ale go nie ładuje.

Parametry mają postać **zmienna=wartosc** np.: **insmod ne.o irq=7**

rmmod nazwa_modulu

Usuwa podany moduł z jądra (jeśli nie jest używany). **nazwa_modulu** to nazwa modułu, a nie nazwa pliku .ko!

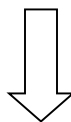
Informacje o modułach

`lsmod`

Wypisuje wszystkie załadowane moduły wraz z informacją od jakich innych modułów zależą (ten sam wynik daje `cat /proc/modules`).

`modinfo [opcje] nazwa_modulu.ko`

Bada plik obiektowy związany z modułem jądra i wypisuje wszelkie zebrane informacje.



```
hostname:~/lkmpg-examples/02-HelloWorld# modinfo hello-1.ko
filename:      hello-1.ko
vermagic:      2.6.11 preempt PENTIUMII 4KSTACKS gcc-3.3
depends:
```

Narzędzie **modprobe**

- Wylistowanie dostępnych modułów:
`$ modprobe -l`
- Wylistowanie załadowanych modułów:
`$ lsmod`
- Ładowanie (instalowanie) modułu **tralala**:
`$ modprobe tralala`
- Ładowanie (instalowanie) modułu **tralala** z nazwą **rumtum**:
`$ modprobe tralala -o rumtum`
- Odinstalowanie modułu **tralala**:
`$ modprobe -r tralala`

modprobe potrafi analizować i rozwikłać zależności instalowanego modułu

Makra `<linux/init.h>` (1/2)

Każdy moduł musi definiować funkcję inicjującą moduł (konstruktor) i zwalnającą moduł (destruktor). Standardowo są to funkcje :

- `int init_module(void) ;`
wywoływana przy ładowaniu modułu. Zwraca kod błędu, jeżeli nie udało się zainicjować modułu, w przeciwnym przypadku 0.
- `void cleanup_module(void) ;`
wywoływana przy usuwaniu modułu

Istnieją makra pozwalające zarejestrować f-kcje o dowolnej nazwie jako konstruktor i destruktor, odpowiednio:

```
module_init(nazwa_funkcji_inicjujacej) ;
```

```
module_exit(nazwa_funkcji_zwalnijacej) ;
```

Makra <linux/init.h> (2/2)

Makra wskazujące funkcje i dane, które mogą być usunięte po inicjalizacji modułu oraz kod i dane związane z usuwaniem modułu – mają znaczenie jedynie gdy sterownik jest skonsolidowany z jądrem.

<code>__init</code>	}	funkcje
<code>__exit</code>		
<code>__initdata</code>	}	dane
<code>__exitdata</code>		

Podczas rozruchu systemu pojawia się komunikat podobny do pokazanego poniżej:

Freeing unused kernel memory: 108k freed

Oznacza to, że zwolniono **108 kB** pamięci jądra zawierającej dane, o których wiadomo, że nie będą już potrzebne. Fragmenty pamięci zwalniane podczas działania systemu stanowią zawartość sekcji `__init` i `__initdata`.

Przykład kodu

hello-3.c

```
#include <linux/module.h>      /* Needed by all modules */
#include <linux/kernel.h>      /* Needed for KERN_INFO */
#include <linux/init.h>        /* Needed for the macros */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
    printk(KERN_INFO "Hello, world %d\n", hello3_data);
    return 0;
}

static void __exit hello_3_exit(void)
{
    printk(KERN_INFO "Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```


Licencjonowanie i opis modułu

Powyżej jądra 2.4 moduły powinny mieć identyfikator licencji **GPL** (lub pokrewny). Odpowiednie makra zdefiniowane są w `<linux/module.h>`

`MODULE_LICENSE(...)`

"GPL", "GPL v2", "Dual BSD/GPL", "Proprietary", itd.

`MODULE_AUTHOR(...)`

`MODULE_DESCRIPTION(...)`

`MODULE_SUPPORTED_DEVICE(...)`

Przykład kodu (1/2)

hello-4.c

```
#include <linux/module.h>      /* Needed by all modules */
#include <linux/kernel.h>      /* Needed for KERN_INFO */
#include <linux/init.h>        /* Needed for the macros */
#define DRIVER_AUTHOR "Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC  "A sample driver"

static int __init init_hello_4(void)
{
    printk(KERN_INFO "Hello, world 4\n");
    return 0;
}

static void __exit cleanup_hello_4(void)
{
    printk(KERN_INFO "Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);
```

Przykład kodu (2/2)

... cd. hello-4.c

```
/*
 * You can use strings, like this:
 */

/*
 * Get rid of taint message by declaring code as GPL.
 */
MODULE_LICENSE("GPL");

/*
 * Or with defines, like this:
 */
MODULE_AUTHOR(DRIVER_AUTHOR);    /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */

/*
 * This module uses /dev/testdevice. The MODULE_SUPPORTED_DEVICE macro might
 * be used in the future to help automatic configuration of modules, but is
 * currently unused other than for documentation purposes.
 */
MODULE_SUPPORTED_DEVICE("testdevice");
```

Przekazywanie argumentów linii komend (1/2)

Można zadeklarować, że określona zmienna będzie zawierała parametr, który może zostać **zmieniony przy ładowaniu** modułu. W czasie ładowania modułu w miejsce podanych zmiennych zostaną wstawione wartości podane przez użytkownika.

Do deklaracji, że pewna zmienna ma być wykorzystana jako parametr modułu służy makro (uprawnienia standardowo ustawiamy na 0):

```
module_param( zmienna, typ, uprawnienia)
```

Dostępne typy to:

```
byte, short, ushort, int, uint, long, ulong, charp,  
bool, invbool
```

Każdy parametr powinien posiadać opis. Opis nadaje się za pomocą makra `MODULE_PARM_DESC`:

```
MODULE_PARM_DESC( zmienna, opis)
```

Przekazywanie argumentów linii komend (2/2)

Aby zadeklarować tablicę parametrów trzeba użyć funkcji:

```
module_param_array( zmienna, typ, wskaznik_na_licznik,  
                    uprawnienia)
```

Wszystkie pola poza **wskaznik_na_licznik** mają takie same znaczenie jak w **module_param()**. **wskaznik_na_licznik** zawiera wskaźnik do zmiennej do której wpisana zostanie liczba elementów tablicy. Jeśli nie interesuje nas liczba argumentów, można podać NULL. Maksymalna liczba elementów tablicy jest określona przez deklarację tablicy.

Przykład kodu (1/3)

hello-5.c

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";
static int myintArray[2] = { -1, -1 };
static int arr_argc = 0;

/*
 * module_param(foo, int, 0000)
 * The first param is the parameters name
 * The second param is it's data type
 * The final argument is the permissions bits,
 * for exposing parameters in sysfs (if non-zero) at a later stage.
 */
```

Przykład kodu (2/3)

... cd. hello-5.c

```
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

/*
 * module_param_array(name, type, num, perm);
 * The first param is the parameter's (in this case the array's) name
 * The second param is the data type of the elements of the array
 * The third argument is a pointer to the variable that will store the number
 * of elements of the array initialized by the user at module loading time
 * The fourth argument is the permission bits
 */
module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
```

Przykład kodu (3/3)

... cd. hello-5.c

```
static int __init hello_5_init(void)
{
    int i;
    printk(KERN_INFO "Hello, world 5\n=====\\n");
    printk(KERN_INFO "myshort is a short integer: %hd\\n", myshort);
    printk(KERN_INFO "myint is an integer: %d\\n", myint);
    printk(KERN_INFO "mylong is a long integer: %ld\\n", mylong);
    printk(KERN_INFO "mystring is a string: %s\\n", mystring);
    for (i = 0; i < (sizeof myintArray / sizeof (int)); i++)
    {
        printk(KERN_INFO "myintArray[%d] = %d\\n", i, myintArray[i]);
    }
    printk(KERN_INFO "got %d arguments for myintArray.\\n", arr_argc);
    return 0;
}

static void __exit hello_5_exit(void)
{
    printk(KERN_INFO "Goodbye, world 5\\n");
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```


Działanie przykładowego kodu

```
satan# insmod hello-5.ko mystring="bebop" mybyte=255 myintArray=-1
mybyte is an 8 bit integer: 255
myshort is a short integer: 1
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: bebop
myintArray is -1 and 420
```

```
satan# insmod hello-5.ko mystring="supercalifragilisticexpialidocious" \
> mybyte=256 myintArray=-1,-1
mybyte is an 8 bit integer: 0
myshort is a short integer: 1
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
myintArray is -1 and -1
```

Kod modułu w wielu plikach

Kod modułu można rozdzielić na wiele plików źródłowych, tak jak każdy inny projekt. W takim przypadku zmiany wymaga jedynie plik **Makefile** – do zmiennej odpowiadającej nazwie modułu docelowego podstawiamy wszystkie nazwy obiektów pośrednich.

W przykładzie kod rozdzielono na dwa pliki: **start.c** i **stop.c**. Docelowo chcemy uzyskać moduł **startstop.ko**.

Przykład kodu

start.c

```
#include <linux/kernel.h>      /* We're doing kernel work */
#include <linux/module.h>      /* Specifically, a module */

int init_module(void)
{
    printk(KERN_INFO "Hello, world - this is the kernel speaking\n");
    return 0;
}
```

stop.c

```
#include <linux/kernel.h>      /* We're doing kernel work */
#include <linux/module.h>      /* Specifically, a module */

void cleanup_module()
{
    printk(KERN_INFO "Short is the life of a kernel module\n");
}
```

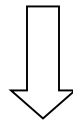
Kompilacja modułu

Makefile

```
obj-m += startstop.o
startstop-objs := start.o stop.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```



startstop.ko

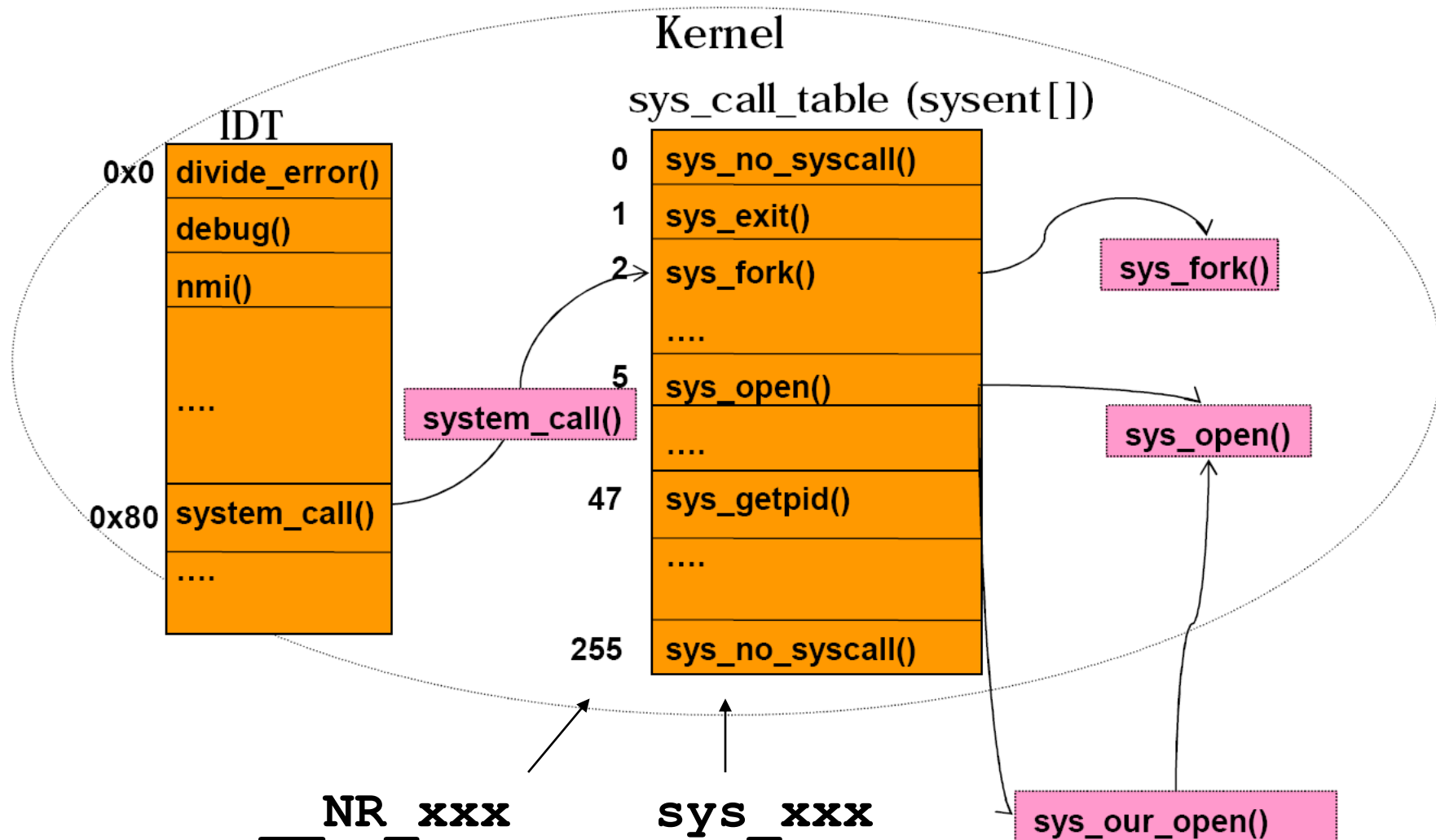
Funkcje dostępne w modułach

- W modułach możemy używać (poza swoimi własnymi) jedynie funkcje wyeksportowane przez jądro!
- Tablicę dostępnych (wyeksportowanych) symboli możemy sprawdzić za pomocą pliku **/proc/kallsyms** albo **/proc/ksyms** (jądra ≤ 2.5)

Przechwytywanie odwołań systemowych

- Wprowadzenie
- Moduły dla jądra ≥ 2.6
- **Przechwytywanie odwołań systemowych**
- Architektura LSM

Tablica funkcji systemowych



Polecenie **strace**

strace to narzędzie do analizy kodu badające interakcję programu z jądrem systemu operacyjnego - śledzi wywołania systemowe programu przestrzeni użytkownika, wyświetla nazwy wywołań, wyświetla argumenty w postaci symbolicznej, wyświetla symboliczną nazwę błędu oraz odpowiadający jej napis (jeżeli któreś z wywołań zakończy się błędem). Dane te uzyskuje z jądra - program może być śledzony bez wsparcia dla debugowania.

Opcje:

- t** kiedy nastąpiło wywołanie systemowe,
- T** czas spędzony w wywołaniu systemowym,
- e** ograniczenie typu śledzonych wywołań, np. -eopen (oznacza -e trace=open) pozwala śledzić tylko wywołania systemowe open,
- o** przekieruje wynik działania programu do pliku,
- f** śledzenie nie tylko procesu macierzystego ale i dzieci,
- p** możliwość "podłączenia" się do działającego w systemie procesu (**strace -p pid**)

Odnajdywanie adresu **sys_call_table**

1. Odwołanie w kodzie modułu do wyeksportowanej zmiennej **sys_call_table** (jedynie dla jąder ≤ 2.4)
2. Użycie pliku **/boot/System.map** (jeżeli dostępny)
3. Przeszukanie pamięci za adresem wyeksportowanej zmiennej **system_utsname** w poszukiwaniu adresu jednej z wyeksportowanych funkcji systemowych (np. **sys_close**)
4. Przeszukanie całego segmentu danych jądra w poszukiwaniu adresu jednej z wyeksportowanych funkcji systemowych (np. **sys_close**)

Przechwytywanie odwołań w jądrze 2.4 (1/4)

- Jądra w wersji 2.4 eksportują symbol `sys_call_table`
- Wyjątkiem są jądra dystrybuowane przez **Red Hat** (zawierają część funkcjonalności z jądra 2.6)
- miejsce w pamięci, gdzie chcemy umieścić wskaźnik do naszej funkcji wybieramy za pomocą indeksu `__NR_XXX` wskazując bezpośrednio wyeksportowanej przez jądro tablicy `sys_call_table` (np. `sys_call_table[__NR_close]`)
- W prezentowanym przykładzie podmieniana jest systemowa funkcja wyjścia `sys_exit`

Przechwytywanie odwołań w jądrze 2.4 (2/4)

intercept_exit.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <sys/syscall.h>

MODULE_LICENSE("GPL");

extern void *sys_call_table[];

asmlinkage int (*original_sys_exit)(int);

asmlinkage int our_fake_exit_function(int error_code)
{
    /*przy każdym wywołaniu wyświetla komunikat na konsoli*/
    printk("UWAGA! sys_exit wywołana z error_code=%d\n", error_code);

    /*wywołuje oryginalną sys_exit i zwraca jej wartość*/
    return original_sys_exit(error_code);
}
```

Przechwytywanie odwołań w jądrze 2.4 (3/4)

... cd. `intercept_exit.c`

```
int init_module(void)
{
    /*zapisujemy odwołanie do oryginalnej funkcji sys_exit*/
    original_sys_exit = sys_call_table[__NR_exit];

    /*zmieniamy sys_call_table, by zamiast niej wywoływała naszą podrobioną
    funkcję wyjścia*/
    sys_call_table[__NR_exit]=our_fake_exit_function;

    return 0;
}

void cleanup_module(void)
{
    /*przywracamy oryginalną sys_exit*/
    sys_call_table[__NR_exit]=original_sys_exit;
}
```

Przechwytywanie odwołań w jądrze 2.4 (4/4)

Kompilacja i testowe użycie:

```
[nieriroot]$ gcc -D__KERNEL__ -DMODULE -I/usr/src/linux/include -c intercept_exit.c
```



```
[root]# insmod ./intercept_exit.o
```



```
[nieriroot]$ ls /tmp/nonexistent  
ls: /tmp/nonexistent: No such file or directory  
UWAGA! sys_exit wywołana z error_code=1
```



```
[root]# rmmod intercept_exit
```

Przechwytywanie odwołań za pomocą **System.map** (1/2)

- W systemie mogą istnieć dwa pliki zawierające adresy i symbole (zmienne i funkcje) jądra:
 - **/boot/System.map** zawiera tablicę symboli generowaną jednorazowo podczas kompilacji jądra, korzystają z niej m.in. **ps**, **syslog**, **klogd**
 - **/proc/kallsyms** (albo **/proc/ksysm**) interfejs w pseudosystemie plików **/proc** generowany przez program **kallsyms** podczas ładowania systemu; również zawiera tablicę symboli
- technika polega na odnalezieniu w pliku **System.map** adresu tablicy **sys_call_table** (np. poleceniem **grep**) i wpisaniu go „na sztywno” do kodu modułu

Przechwytywanie odwołań za pomocą **System.map (2/2)**

```
[nieroot]$ grep sys_call_table /boot/System.map  
c044fd00 D sys_call_table
```



```
...  
*(long *)&sys_call_table=0xc044fd00;  
...  
  
asmlinkage long hacked_sys_unlink(const char *pathname)  
{  
    return -1;  
}  
...  
  
original_sys_unlink =(void * )xchg(&sys_call_table[__NR_unlink],  
hacked_sys_unlink);  
...
```

Wymuszanie dostępu do **sys_call_table** użycie **system_utsname** (1/4)

- Struktura **system_utsname** zawiera listę informacji o systemie i jest eksportowana przez jądro (czyli przez jej nazwę/symbol mamy dostęp jej adresu)
- Tablica **sys_call_table** umieszczona jest w pamięci **za** strukturą **system_utsname**
- Jądro eksportuje część funkcji systemowych (np. **sys_exit**, **sys_close**, itd.)
- Począwszy od adresu **system_utsname** przeszukujemy pamięć porównując zawartość danej lokalizacji z adresem wybranej f-cji systemowej

Wymuszanie dostępu do **sys_call_table**

użycie **system_utsname** (2/4)

- Przykładowy moduł przechwytuje odwołanie systemowe **sys_open** i nie pozwala na otwarcie **/tmp/test**

intercept_open.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/unistd.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <linux/namei.h>
```

```
int flag=0;
```

```
#define MAX_TRY 1024
```

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

```
unsigned long *sys_call_table;
```

```
asmlinkage long (*original_sys_open) (const char __user *filename, int
flags, int mode);
```

Wymuszanie dostępu do `sys_call_table`

użycie `system_utsname` (3/4)

cd. `intercept_open.c`

```
asmlinkage int our_fake_open_function(const char __user *filename, int
flags, int mode)
{
    int error;
    struct nameidata nd, nd_t;
    struct inode *inode, *inode_t;
    mm_segment_t fs;

    error = user_path_walk(filename, &nd);

    if (!error)
    {
        inode = nd.dentry->d_inode;

        /* To trzeba zrobić przed wywołaniem user_path_walk()
        z przestrzeni jądra: */
        fs = get_fs();
        set_fs(get_ds());

        /* Chroni plik /tmp/test. Można zmienić na inny */
        error = user_path_walk("/tmp/test", &nd_t);

        set_fs(fs);

        if (!error)
        {
            inode_t = nd_t.dentry->d_inode;

            if (inode == inode_t)
                return -EACCES;
        }
    }

    return original_sys_open(filename, flags, mode);
}
```

Wymuszanie dostępu do **sys_call_table** użycie **system_utsname** (4/4)

cd. **intercept_open.c**

```
static int __init my_init (void)
{
    int i=MAX_TRY;
    unsigned long *sys_table;
    sys_table = (unsigned long *)&system_utsname;

    while(i)
    {
        if(sys_table[__NR_read] == (unsigned long)sys_read)
        {
            sys_call_table=sys_table;
            flag=1;
            break;
        }
        i--;
        sys_table++;
    }

    if(flag)
    {
        original_sys_open =(void *)xchg(&sys_call_table[__NR_open],
        our_fake_open_function);
    }
    return 0;
}

static void my_exit (void)
{
    xchg(&sys_call_table[__NR_open], original_sys_open);
}

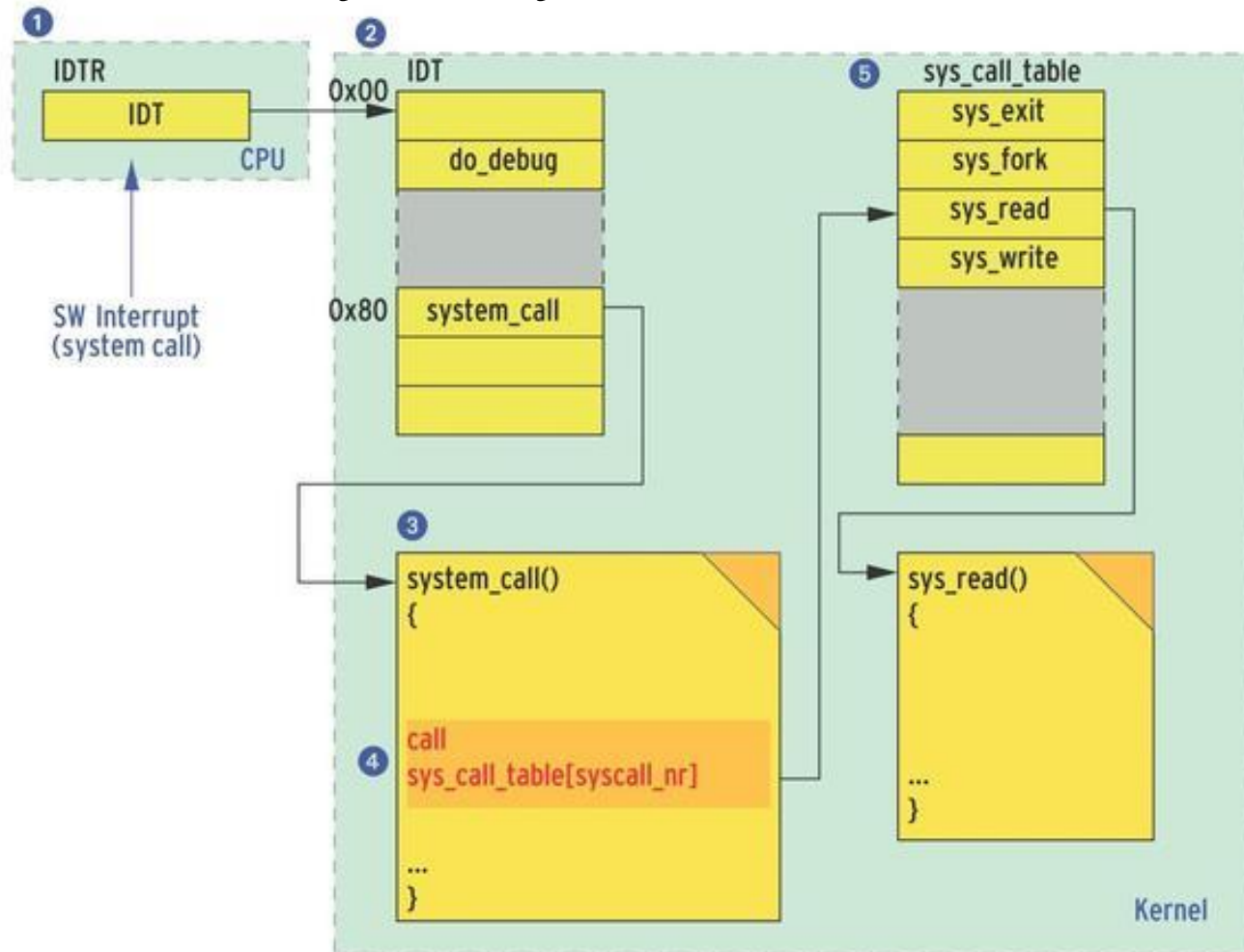
module_init(my_init);
module_exit(my_exit);
```

Wymuszanie dostępu do **sys_call_table** przeszukiwanie całego obszaru danych

```
int get_sct ()
{
    unsigned long *ptr;
    ptr=(unsigned long *)(( init_mm . end_code + 4) & 0xffffffffc);
    printk (KERN_INFO "Searching for sys_call_table address...\n");
    printk ("Start: %p End: %p\n", init_mm.end_code, init_mm.end_data);
    printk ("Ptr: %p\n",ptr);
    /* Lookup for the table in the data section. */
    while((unsigned long )ptr < (unsigned long)init_mm.end_data)
    {
        /* The hit has happend! */
        if((unsigned long *)*ptr == (unsigned long *)sys_close)
            break;

        ptr++;
    }
    printk ("Ptr: %p\n",ptr);
}
```

Wymuszanie dostępu do **sys_call_table** wykorzystanie IDT 1/2



Wymuszanie dostępu do **sys_call_table** wykorzystanie IDT 2/2

Adres f-cji **system_call()** umieszczony jest w **Interrupt Descriptor Table** (IDT). Adres samego IDT umieszczony jest w rejestrze IDTR procesora.

1. Czytamy rejestr IDTR i uzyskujemy adres IDT
2. W tablicy IDT pozycja o indeksie **0x80** wskazuje funkcję **system_call()**
3. Zaczynamy przeszukiwanie pamięci od początku wskazanej funkcji **system_call()** ...
4. ... do miejsca w którym wołane jest polecenie **call** z argumentem będącym ...
5. ... adresem konkretnego wywołania systemowego w **syscall()**

Wymuszanie dostępu do **sys_call_table** obszar pamięci tylko do odczytu

Sekwencja dla architektury x86 ustawiająca 16 bit w rejestrze kontrolnym **CR0**. Bit ten umożliwia włączenie bądź wyłączenie ochrony przed zapisem (określa czy procesor może pisać do stron zaznaczonych tylko do odczytu).

```
void disable_write_protection_cr0 ( void )
{
    unsigned long value;
    asm volatile("mov %%cr0, %0" : "=r" (value));
    if ( value & 0x00010000 ) {
        value &= ~0x00010000;
        asm volatile("mov %0, %%cr0" : : "r" (value));
    }
}
```

Architektura LSM

- Wprowadzenie
- Moduły dla jądra ≥ 2.6
- Przechwytywanie odwołań systemowych
- **Architektura LSM**

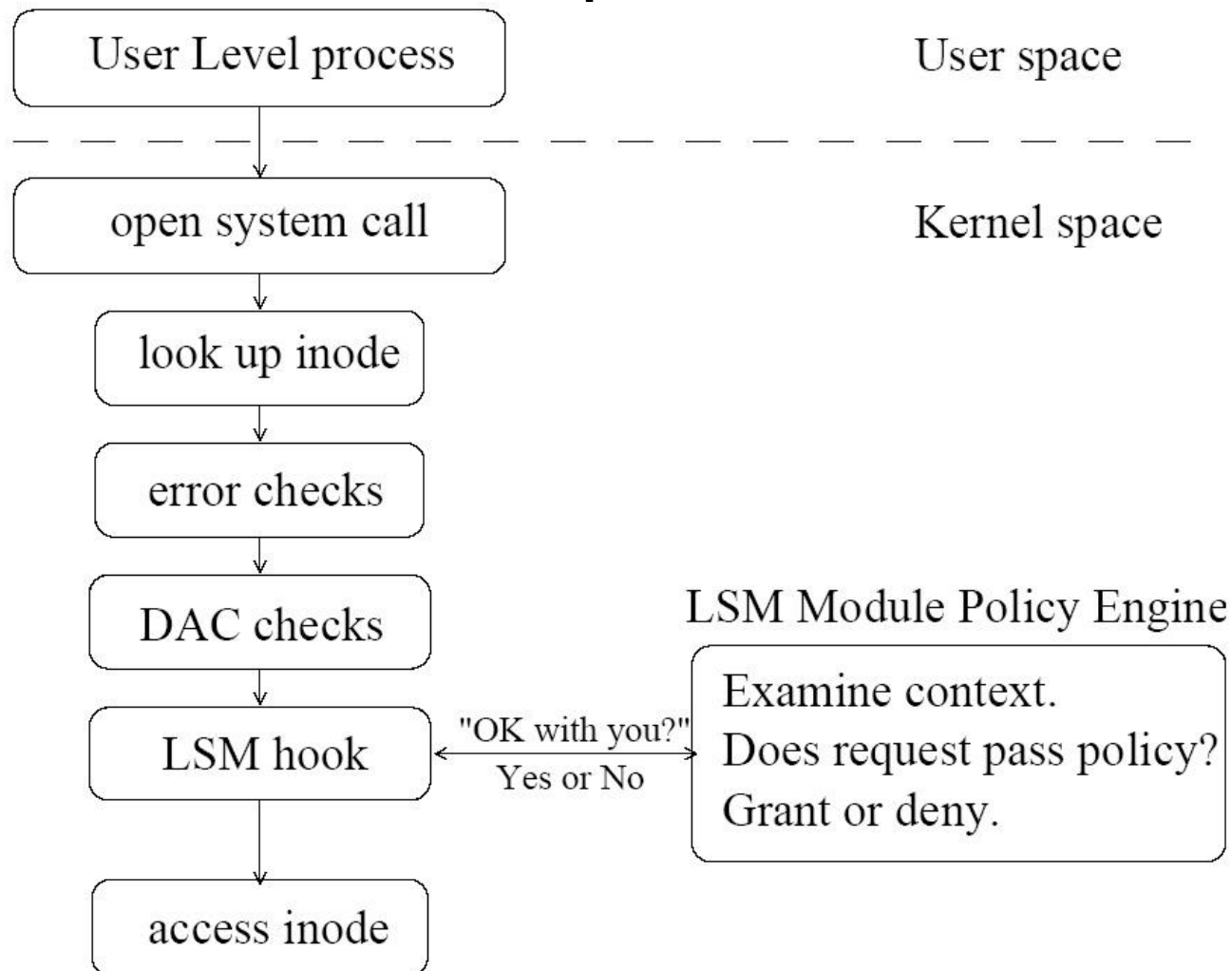
Założenia implementacyjne LSM

- Dodanie **pól** dotyczących bezpieczeństwa do wybranych struktur danych jądra
- Wprowadzenie **haków** dla funkcji bezpieczeństwa
- Dodanie nowych **funkcji systemowych**
- Wprowadzenie funkcji pozwalających na zarejestrowanie i wyrejestrowanie modułów bezpieczeństwa

Struktury
modyfikowane przez
LSM

STRUCTURE	OBJECT
task_struct	Task (Process)
linux_binprm	Program
super_block	Filesystem
inode	Pipe, File, or Socket
file	Open File
sk_buff	Network Buffer (Packet)
net_device	Network Device
kern_ipc_perm	Semaphore, Shared Memory Segment, or Message Queue
msg_msg	Individual Message

Architektura haków LSM na przykładzie f-cji open



Interfejs LSM

- Rejestracja polityki (*Policy Registration*)
- Haki do zadań (*Task Hooks*)
- Haki do ładowania programów (*Program Loading Hooks*)
- Haki do systemu plików (*File system hooks*)
- Haki do IPC (*IPC Hooks*)
- Haki modułów (*Module Hooks*)
- Haki sieciowe (*Network Hooks*)
- Inne haki systemowe

Przykład modułu LSM 1/5

Budowa modułu

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/config.h>
#include <linux/security.h>
#include <linux/syscalls.h>
#include <linux/init.h>

static int __init emod_init(void) (1)
{
    printk(KERN_ALERT "[LSM] Modul aktywny\n");
    return 0;
}

static void __exit emod_exit(void) (2)
{
    printk(KERN_ALERT "[LSM] Modul nieaktywny\n");
}

security_initcall(emod_init); (3)
module_exit(emod_exit); (4)

MODULE_LICENSE("GPL"); (5)
MODULE_DESCRIPTION("LKM-LSM-Sample"); (6)
MODULE_AUTHOR("Autor"); (7)
```

- 1 - inicjalizacja modułu
- 2 - odładowanie modułu
- 3 - wskazanie funkcji inicjującej
- 4 - wskazanie procedury wywoływanej przy odinstalowywaniu modułu
- 5 - typ licencji moduły
- 6 - opis modułu
- 7 - autor

Przykład modułu LSM 2/5

Wykorzystanie LSM

```
...
static struct security_operations SEC; (8)

static int __init emod_init(void)
{
    memset(&SEC,0,sizeof(SEC));
    if(!register_security(&SEC)) (9) printk(KERN_ALERT "[LSM] Moduł aktywny\n");
    else printk(KERN_ALERT "[LSM] Niepoprawna inicjalizacja\n");
    return 0;
}

static void __exit emod_exit(void)
{
    unregister_security(&SEC); (10)
    printk(KERN_ALERT "[LSM] Moduł nieaktywny\n");
}
...
```

8 - struktura pozwalająca zarejestrować obsługę bezpieczeństwa

9 - rejestrowanie obsługi na podstawie w/w struktury

10 - odłączenie obsługi

Przykład modułu LSM 3/5

Kontrola uruchamiania programu

```

...
static int emod_bprm(struct linux_binprm *B) (11)
{
    if(!strstr(B->filename, "/no/")) {
        printk(KERN_ALERT "[LSM] Uruchamianie '%s' zablokowane\n", B->filename);
        return -EPERM; (12)
    }
    return 0;
}

static int __init emod_init(void)
{
    memset(&SEC, 0, sizeof(SEC));
    SEC.bprm_check_security = emod_bprm; (13)
    ...
}
...

```

11 - składnia funkcji wywoływanej przy kontroli uruchamiania pliku

12 - zablokowanie akcji

13 - wskazanie funkcji kontrolującej uruchamianie plików

char[128]	buf	bufor, do którego ładowany jest początkowy fragment pliku wykonywalnego, określający format tego pliku
unsigned long [MAX_ARG_PAGES]	page	tablica stron zawierających argumenty wywołania i środowisko
unsigned long	p	adres pamięci, która zawiera argumenty wywołania i środowisko
int	sh_bang	włącznik znacznika interpretowania (zapewnia on, że interpreterem języka skryptowego nie jest program napisany w języku skryptowym)
struct inode*	inode	i-węzeł pliku wykonywalnego
int	e_uid	obowiązujący identyfikator użytkownika dla nowego programu
int	e_gid	obowiązujący identyfikator dla grupy użytkownika dla nowego programu
int	argc	liczba argumentów wywołania
int	envc	liczba argumentów środowiskowych
char *	filename	nazwa ścieżkowa pliku wykonywalnego
unsigned long	loader	pomocniczy adres używany podczas ładowania
unsigned long	exec	pomocniczy adres używany podczas ładowania
int	dont_iput	informuje, że i-węzeł został ustanowiony przez funkcję obsługi binfmt

Przykład modułu LSM 4/5

Blokowanie wywołania f-cji systemowych

```
...
static int emod_capable(struct task_struct *T, int C) (14)
{
    if(C==CAP_SYS_TIME) (15)
    {
        printk(KERN_ALERT "[LSM] Blokada zmiany czasu\n");
        return -EPERM;
    }
    // więcej -- linux/include/linux/capability.h, linux/include/linux/capability.c (16)
    return 0;
}

static int __init emod_init(void)
{
    memset(&SEC,0,sizeof(SEC));
    SEC.capable = emod_capable; (17)
    SEC.bprm_check_security = emod_bprm;
    ...
}
...
```

14 - składnia funkcji wywoływanej przy kontroli uprawnień do wywoływania akcji systemowych

15 - kontrola zmiany czasu w systemie

16 - we wskazanym pliku znajduje się więcej informacji

17 - wskazanie funkcji kontrolującej uprawnienia

Przykład modułu LSM 5/5

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/config.h>
#include <linux/security.h>
#include <linux/syscalls.h>
#include <linux/init.h>
#include <linux/string.h>

static struct security_operations SEC;

static int emod_capable(struct task_struct *T, int C)
{
    if(C==CAP_SYS_TIME)
    {
        printk(KERN_ALERT "[LSM] Blokada zmiany czasu\n");
        return -EPERM;
    }
    // więcej -- linux/include/linux/capability.h, linux/include/linux/capability.c
    return 0;
}

static int emod_bprm(struct linux_binprm *B)
{
    if(!strstr(B->filename, "/no/"))
    {
        printk(KERN_ALERT "[LSM] Uruchamianie '%s' zablokowane\n", B->filename);
        return -EPERM;
    }
    return 0;
}

static int __init emod_init(void)
{
    memset(&SEC, 0, sizeof(SEC));
    SEC.capable = emod_capable;
    SEC.bprm.check_security = emod_bprm;
    if(!register_security(&SEC)) printk(KERN_ALERT "[LSM] Moduł aktywny\n");
    else printk(KERN_ALERT "[LSM] Niepoprawna inicjalizacja\n");
    return 0;
}

static void __exit emod_exit(void)
{
    unregister_security(&SEC);
    printk(KERN_ALERT "[LSM] Moduł nieaktywny\n");
}

security_initcall(emod_init);
module_exit(emod_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("LKM-LSM-Sample");
MODULE_AUTHOR("Autor");
```