

Universidad Nacional de Córdoba



Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Electrónica y Computación

Cátedra de Sistemas de Computación

Trabajo de Laboratorio 4

Profesor Titular: Ing. Javier Alejandro JORGE

Profesor Adjunto: -

Integrantes:

Trucchi, Genaro

Trachta, Agustin

Rodriguez, Mateo

26 de mayo de 2025

Índice

1. ¿Qué es un módulo del kernel y para qué se usa?	2
2. Estructura de un módulo de kernel (<code>module_init</code> , <code>module_exit</code> , <code>Makefile</code>)	2
3. Compilación e instalación de módulos (<code>make</code> , <code>insmod</code> , <code>rmmod</code> , <code>lsmod</code> , <code>modinfo</code>)	4
4. Seguridad: firma de modulos para Secure Boot (por que es importante y como se realiza?)	6
5. Diferencia entre espacio de usuario y espacio de kernel (seguridad e implicancias de errores como segmentation fault)	7
6. Llamadas al sistema con <code>strace</code> y diferencia con las funciones disponibles en el kernel	8
7. Archivos y directorios relevantes: <code>/proc</code> , <code>/dev</code> , <code>/lib/modules/\$(uname -r)/kernel</code>	9
8. Drivers y deteccion de hardware (<code>hwinfo</code> , <code>lspci</code> , <code>lsusb</code> , etc.)	10
9. Caso de estudio: Secure Boot, GRUB y el parche de Microsoft (Análisis del artículo de Ars Technica)	12
10. ¿Qué es <code>checkinstall</code> y para que sirve?	14
11. Empaquetar un “Hello World” con <code>checkinstall</code>	15
12. Mejorar la seguridad del kernel: evitar modulos no firmados y mitigar rootkits	17

1. ¿Qué es un módulo del kernel y para qué se usa?

Un **módulo del kernel** es un fragmento de código que puede ser cargado o descargado en el kernel de Linux según sea necesario, extendiendo la funcionalidad del sistema operativo sin necesidad de recompilar o reiniciar el sistema. En esencia, los módulos permiten tener un kernel modular y flexible: podemos agregar o quitar características del kernel en tiempo de ejecución.

Algunos usos típicos de los módulos del kernel son

- **Controladores de dispositivos (drivers):** permiten al kernel comunicarse con hardware específico (ej. controladores de gráficos, Wi-Fi, dispositivos de audio, etc.). De hecho, muchos controladores de hardware se implementan como módulos cargables.
- **Sistemas de archivos:** el soporte para nuevos tipos de sistemas de archivos puede añadirse como módulo.
- **Extensiones del núcleo:** cualquier funcionalidad adicional (monitorizar temperaturas, agregar funcionalidades de seguridad, etc.) puede implementarse como módulo cargable.
- **Módulos privativos o de terceros:** por ejemplo, controladores propietarios (como el driver NVIDIA) que no están incluidos en el kernel base se distribuyen como módulos separados.

Sin módulos, tendríamos que compilar todas estas funcionalidades directamente en el kernel monolítico, haciendo el núcleo más grande y obligando a reconstruir y reiniciar el sistema cada vez que quisiéramos una nueva funcionalidad. Gracias a los módulos, el kernel puede mantenerse ligero y ampliable dinámicamente.

Módulos *vs.* programas en espacio de usuario

A diferencia de una aplicación de usuario normal (un programa común que ejecutamos en Linux), un módulo del kernel corre en modo privilegiado dentro del núcleo. Los programas de usuario se ejecutan en el *espacio de usuario* y deben llamar a servicios del sistema operativo (mediante llamadas al sistema) para realizar acciones privilegiadas. En cambio, un módulo forma parte del kernel: tiene acceso directo al hardware y a las estructuras internas del sistema, con las ventajas y riesgos que esto implica (ver sección 3). Además, un programa de usuario es un ejecutable con un punto de entrada (`main()` típicamente), mientras que un módulo es esencialmente un fichero objeto especial (`.ko`) que se inserta en el kernel y no tiene `main()` propio. Más adelante profundizaremos en estas diferencias de estructura y contexto de ejecución.

2. Estructura de un módulo de kernel (`module_init`, `module_exit`, `Makefile`)

Para crear un módulo de kernel simple, debemos seguir una estructura específica ya que, como mencionamos, no existe la función `main()` en un módulo. En su lugar, un

módulo define dos funciones especiales registradas mediante macros proporcionadas por el kernel:

- **Función de inicialización (`module_init`):** es la función que se ejecuta cuando el módulo se inserta (carga) en el kernel. Se registra con la macro `module_init(nombre_función_ini`. En esta rutina se realiza la configuración necesaria: por ejemplo, registrar manejadores, inicializar estructuras de datos, imprimir un mensaje de log, etc. Al insertarse el módulo, el kernel llamará automáticamente a esta función.
- **Función de salida (`module_exit`):** es la función ejecutada al remover el módulo del kernel. Se registra con la macro `module_exit(nombre_función_exit)`. En esta rutina se liberan recursos, se anulan registros realizados en la `init` y, en general, se realiza cualquier limpieza necesaria antes de que el módulo sea descargado.

Estas dos funciones actúan, en cierta forma, como el “constructor” y “destructor” del módulo. Los módulos deben tener una función de entrada y otra de salida definidas de esta forma. Un código mínimo de un módulo podría ser:

```
#include <linux/module.h>
#include <linux/init.h>

static int __init mi_modulo_init(void) {
    printk(KERN_INFO "Mi_modulo_ha_sido_insertado!\n");
    return 0; /* 0 indica carga exitosa */
}

static void __exit mi_modulo_exit(void) {
    printk(KERN_INFO "Mi_modulo_ha_sido_removido.\n");
}

module_init(mi_modulo_init);
module_exit(mi_modulo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tu_Nombre");
MODULE_DESCRIPTION("Modulo_de_ejemplo_Hola_Mundo");
```

Ademas del código C del módulo, es crucial proveer un **Makefile** adecuado para compilar el módulo contra el kernel. Los módulos se compilan utilizando el sistema de construcción del kernel (*Kbuild*). Un Makefile típico para un módulo simple podría contener:

```
obj-m := mimodulo.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

Este Makefile indica que queremos construir un objeto modular `mimodulo.ko` a partir de `mimodulo.c`. Usamos la regla especial `obj-m` para listar el nombre del modulo. Luego invocamos `make -C /lib/modules/$(uname -r)/build M=$(PWD)` para que el sistema de compilación del kernel compile nuestro código como modulo externo. Esto requiere tener instalados los *headers* o el código fuente del kernel correspondiente a nuestra versión (en Debian/Ubuntu, instalar el paquete `linux-headers-$(uname -r)`).

Una vez escrito el código fuente y el Makefile, el flujo es: compilar con `make` para obtener el archivo `mimodulo.ko`, y luego podremos insertarlo en el kernel con herramientas como `insmod` (ver sección 3).

3. Compilación e instalación de módulos (make, insmod, rmmod, lsmod, modinfo)

Compilación

Tras preparar el código fuente del módulo y su Makefile, la compilación se realiza normalmente con el comando `make`. Esto genera un archivo objeto con extensión `.ko` (*kernel object*). Por ejemplo, siguiendo el Makefile antes mencionado, simplemente ejecutar `make` en el directorio del código producirá `mimodulo.ko`. Si es necesario limpiar archivos de compilación previos, `make clean` ayudará.

Inserción de un módulo (insmod/modprobe)

Para instalar un módulo en el kernel en ejecución, se utiliza la utilidad `insmod`. Por ejemplo:

```
sudo insmod mimodulo.ko
```

`insmod` inserta el modulo especificado directamente, pero no resuelve dependencias automáticamente. Si el modulo requiere de otros modulos (por ejemplo, un driver puede depender de otro modulo *core*), esas dependencias deben estar cargadas previamente. Alternativamente, se puede usar `modprobe`, que si resuelve dependencias consultando la base de datos de modulos del sistema. `modprobe mimodulo` buscaria `mimodulo.ko` en las rutas de modulos instalados del sistema (normalmente en `/lib/modules/$(uname -r)/...`) e intentaria cargarlo junto con cualquier dependencia declarada.

Verificación de carga (lsmod, /proc/modules, dmesg)

Después de insertar un módulo, es útil comprobar que esté efectivamente cargado y revisar mensajes de log:

- El comando `lsmod` lista todos los módulos actualmente cargados en el kernel. Si nuestro módulo se cargó correctamente, `lsmod | grep mimodulo` debería mostrarlo, indicando también su tamaño y el número de referencias (usos) que tiene. Internamente, `lsmod` simplemente formatea la información disponible en el archivo virtual `/proc/modules`. Podemos inspeccionar directamente `/proc/modules` para ver información similar en formato texto.

- El comando `dmesg` muestra el registro de mensajes del kernel (el *kernel ring buffer*). Nuestro módulo, al cargar, probablemente imprimió algo con `printk()`. Usando `dmesg | tail` veremos los últimos mensajes del kernel, donde debería aparecer el mensaje de inicialización del módulo (ej. “Mi módulo ha sido insertado!”). Esto confirma que la función `module_init` se ejecutó. Igualmente, al remover, habrá mensajes de salida del módulo.

Remoción de un módulo (rmmod)

Para quitar un módulo del kernel, se utiliza `rmmod`. Ejemplo:

```
sudo rmmod mimodulo
```

Esto invoca la rutina de limpieza (`module_exit`) del módulo y luego descarga el código del kernel. Tras hacer `rmmod`, conviene nuevamente chequear `lsmod` (para confirmar que ya no figura) y `dmesg` (debería mostrar el mensaje de que el módulo se removió correctamente).

Información de un módulo (modinfo)

La herramienta `modinfo` muestra metadatos y detalles de un módulo compilado. Por ejemplo,

```
modinfo mimodulo.ko
```

listará información como: descripción, autor, licencia, alias, versión de *vermagic* (la versión de kernel para la que fue compilado) y si está firmado digitalmente (más sobre firma en la siguiente sección). Esto es útil para inspeccionar rápidamente las propiedades de un módulo antes de cargarlo o para verificar si la compilación se hizo para la versión correcta del kernel.

Ejemplo práctico resumido

Supongamos que ya compilamos `mimodulo.ko` exitosamente:

1. Insertamos el módulo:
`sudo insmod mimodulo.ko`
2. Verificamos con `lsmod` – deberíamos ver `mimodulo` en la lista de módulos cargados. También podríamos hacer:
`cat /proc/modules | grep mimodulo`
3. Miramos el log:
`dmesg | tail -n 5`
y vemos algo como “Mi módulo ha sido insertado!” indicando que la `init` corrió.
4. Probamos quitarlo:
`sudo rmmod mimodulo`
5. Nuevamente `dmesg | tail` mostrará “Mi módulo ha sido removido.” de la función `exit`, y `lsmod` confirmará que ya no está en memoria.

6. Ejecutamos `modinfo mimodulo.ko` para ver los metadatos: por ejemplo, confirmamos que la licencia es GPL (lo cual es importante; si un módulo no está marcado GPL, el kernel lo registra como “*proprietary*” y ciertas funciones internas podrían no estar disponibles para ese módulo).

Esta secuencia básica permite compilar un módulo fuera del árbol del kernel, insertarlo, verificar su funcionamiento y luego retirarlo de manera limpia.

4. Seguridad: firma de modulos para Secure Boot (por que es importante y como se realiza?)

Cuando Secure Boot (arranque seguro UEFI) esta habilitado en un sistema, el firmware y el cargador de arranque solo permitiran arrancar codigo firmado con claves de confianza. Esta cadena de confianza se extiende hasta el propio kernel y sus modulos: no basta con que el kernel este firmado, los modulos del kernel tambien deben estar firmados digitalmente para que el kernel los cargue bajo Secure Boot. El objetivo es prevenir que codigo no autorizado o malicioso (como un rootkit) se inyecte en el kernel.

Por que es importante?

Sin firma obligatoria, un atacante con privilegios de superusuario podria cargar un modulo malicioso que comprometa el sistema a nivel de kernel (por ejemplo, ocultar procesos, capturar contraseñas, etc.). La firma de modulos garantiza que solo modulos con una firma valida—es decir, provenientes de una fuente confiable—puedan insertarse cuando Secure Boot esta activo, reforzando la seguridad del sistema. Una recomendacion frecuente de seguridad del kernel es evitar cargar modulos no firmados para dificultar la instalacion de rootkits.

Como funciona la verificacion?

Al compilar el kernel, las distribuciones incluyen una clave privada con la que firman todos sus modulos oficiales; el kernel incorpora (o tiene acceso a) la correspondiente clave publica. Cuando Secure Boot esta activo, el kernel comprueba la firma de cada modulo cargado contra las claves X.509 de confianza que posee en sus llaveros internos (`.builtin_trusted_keys` y `.platform`). En terminos simples: el modulo lleva una firma digital anexada y el kernel solo lo aceptara si puede verificar esa firma con alguna clave publica reconocida como confiable. Si la firma no es valida o el modulo no esta firmado, la carga falla.

La documentacion de Red Hat indica que con Secure Boot activo (o con la opcion `module.sig_enforce` habilitada) “solo se pueden cargar modulos del kernel firmados cuyas firmas se hayan autenticado contra claves del llavero de confianza del sistema (`.builtin_trusted_keys`) o de la plataforma (`.platform`); ademas la clave publica del firmante no debe estar revocada (`.blacklist`)”. En cambio, si Secure Boot esta deshabilitado, el kernel permite cargar modulos no firmados libremente (o modulos firmados cuya clave no conozca), es decir, la verificacion de firmas deja de ser obligatoria.

Como se firma un modulo?

El proceso general para firmar un modulo personalizado es:

1. **Generar un par de claves X.509 (privada y publica).** Puede ser una clave autofirmada. Por ejemplo, con `openssl` se crea una clave RSA y un certificado X.509, indicando en el campo *Common Name* algo como “Module Signing Key”.
2. **Firmar el modulo .ko.** El kernel provee `scripts/sign-file`, que aplica una firma PKCS#7 al modulo usando la clave privada. Ejemplo: `scripts/sign-file sha256 clave_privada.pem certificado_publico.pem mimodulo.ko` Esto calcula un hash SHA-256 del modulo y lo firma; `modinfo mimodulo.ko` mostrara campos como `sig_id`, `sig_key`, `sig_hash` algo.
3. **Registrar la clave publica en el sistema.** En entornos con Secure Boot se suele usar MOK (Machine Owner Key). Se convierte el certificado a DER y se ejecuta `sudo mokutil -import cert.der` Tras reiniciar y confirmar en la interfaz UEFI, la clave quedara en el llavero `.platform` o en el llavero de sistema, y el kernel la reconocera.

Una vez registrada la clave, `insmod mimodulo.ko` funcionara sin error; si la clave no estuviera registrada, apareceria “Required key not available” y el modulo no se cargaria.

5. Diferencia entre espacio de usuario y espacio de kernel (seguridad e implicancias de errores como segmentation fault)

Los sistemas operativos modernos dividen la ejecucion en dos espacios de memoria bien diferenciados: *espacio de usuario* y *espacio de kernel*.

Espacio de usuario (user space)

- Aloja las aplicaciones normales con privilegios limitados.
- Cada proceso tiene su propio espacio de direcciones virtuales y no puede acceder directamente a la memoria del kernel ni a la de otros procesos.
- Si un programa intenta leer o escribir fuera de su rango permitido, el hardware y el kernel lo impiden, provocando un *segmentation fault*.
- Un fallo en una aplicacion de usuario rara vez compromete todo el sistema: se termina solo ese proceso.

Espacio de kernel

- Reservado para el nucleo y sus extensiones (drivers, modulos cargados).
- Corre en modo privilegiado (ring 0 en x86), con acceso completo al hardware y a toda la memoria.

- Un error aqui (por ejemplo, desreferenciar un puntero invalido) puede causar un *oops* del kernel o un *kernel panic*, bloqueando por completo el sistema.
- No existen las mismas barreras de aislamiento que en user space; por eso los modulos deben ser confiables.

Implicancias de seguridad

La separacion usuario/kernel existe para que el kernel actue de guardian: un programa malicioso en user space esta confinado; si intenta algo prohibido el kernel lo detiene. En cambio, ejecutar codigo en kernel space (mediante un modulo malicioso o explotando un bug en un driver) otorga control total: se pueden desactivar protecciones, ocultar procesos, espiar datos de cualquier aplicacion, etc. De ahi la importancia de cargar solo modulos de confianza.

Segmentation fault vs. errores en kernel

- **User space:** un *segmentation fault* ocurre al acceder memoria invalida; el kernel mata el proceso y libera recursos, manteniendo el sistema estable. Herramientas como *gdb* y *core dumps* facilitan la depuracion.
- **Kernel space:** un acceso invalido genera un *oops*; si es grave, un *kernel panic*. No hay entorno aislado: todo el sistema puede caer. Depurar implica revisar trazas del kernel o usar herramientas como *kgdb*.

6. Llamadas al sistema con strace y diferencia con las funciones disponibles en el kernel

Una llamada al sistema (*system call*) es el mecanismo mediante el cual un programa en espacio de usuario solicita un servicio al kernel (por ejemplo, “lee este archivo”, “envia estos datos por la red”, etc.). Se puede pensar en las *syscalls* como la frontera de comunicacion entre *user space* y *kernel space*. La herramienta **strace** resulta sumamente util para observar en tiempo real que llamadas al sistema hace un programa de usuario; basicamente traza esa “capa fina” de interaccion entre un proceso y el kernel.

Por ejemplo, si escribimos un programa en C que hace `printf("Hola mundo\n")` y lo ejecutamos, internamente ese `printf` terminara invocando la llamada al sistema `write()` para enviar texto a la salida estandar (descriptor 1). Si corremos **strace ./programa**, veremos una traza de todas las *syscalls* realizadas por el proceso: aparecera una linea con `write(1, "Hola mundo\n", 11)` y su valor de retorno, ademas de otras llamadas como `brk` (ajuste de *heap*), `exit` al terminar, etc. **strace** intercepta estas llamadas mediante la funcionalidad `ptrace` del kernel y las imprime para nuestro analisis. Tambien podemos usar **strace -c** para un resumen estadistico (cuentas y tiempo), o **strace -tt** para marcas de tiempo, entre otras opciones.

Esto evidencia que un programa en *user space* no accede al hardware ni realiza funciones privilegiadas por si mismo, sino que todo lo ejecuta pidiendoselo al kernel mediante *syscalls*. La libreria de C (*glibc*) provee funciones de mas alto nivel (`printf`, `open`, `socket`, etc.), pero casi todas terminan en una llamada al sistema subyacente (`write`, `openat`, `socket`, ...). De esta forma, las operaciones disponibles para un programa de

usuario son esencialmente las que brinda la API del sistema operativo via llamadas al sistema (junto con las funciones de biblioteca que las envuelven). Podemos listar todas las *syscalls* consultando `man 2 syscalls`: Linux define cientos de llamadas, pero siguen siendo un conjunto limitado y bien especificado de servicios que el kernel expone.

En contraste, un modulo de kernel no realiza llamadas al sistema (no tendria sentido: ya se ejecuta dentro del kernel). En su lugar, un modulo puede invocar directamente cualquier funcion exportada por el kernel. El conjunto de funciones disponibles es enorme: basicamente todas las declaradas con `EXPORT_SYMBOL`, `EXPORT_SYMBOL_GPL`, etc. Por ejemplo, un modulo puede llamar a `printk()` para imprimir logs, usar `kmalloc` para asignacion de memoria en kernel, registrar dispositivos, etc. No puede usar funciones de `glibc` u otras librerias de usuario (`printf`, `malloc`, ...); en su lugar existen equivalentes del kernel.

Si un modulo intenta invocar una funcion que el kernel no haya exportado, la carga fallara con “simbolo desconocido”. Ademas, no existe una herramienta equivalente a `strace` para modulos, porque no hay *syscalls* involucradas una vez dentro del kernel. Para analizar un modulo se recurre a depuracion del kernel, revisando `/proc/kallsyms` (tabla de simbolos) o instrumentando el codigo con `printk`. `/proc/kallsyms` muestra direcciones de funciones y variables exportadas—requiere normalmente permisos de `root` y, segun configuracion, puede ocultar direcciones exactas.

7. Archivos y directorios relevantes: `/proc`, `/dev`, `/lib/modules/$(uname -r)/kernel`

Linux expone informacion sobre el kernel, los procesos y el hardware mediante sistemas de archivos virtuales y directorios especificos. Los mas importantes al trabajar con modulos y drivers son `/proc`, `/dev` y el arbol de modulos instalados en `/lib/modules`.

`/proc`

Es un sistema de archivos virtual (no existe en disco) montado en `/proc`. Contiene una vista en forma de archivos de muchos aspectos del estado del kernel, generada dinamicamente. Archivos clave:

- `/proc/modules`: lista los modulos actualmente cargados (nombre, tamano, cuenta de usos). Equivale a la salida de `lsmod`. Se actualiza al cargar o descargar modulos.
- `/proc/kallsyms`: tabla de simbolos del kernel (funciones y variables exportadas). Una vez cargado un modulo, sus simbolos exportados aparecen aqui. Por seguridad, algunos kernels restringen la visibilidad de direcciones a usuarios no privilegiados.
- Otros archivos como `/proc/cpuinfo`, `/proc/iomem`, `/proc/interrupts` proveen detalles de hardware, utiles para diagnosticar que recursos manejan ciertos drivers.

`/dev`

Contiene los nodos de dispositivo (*device nodes*) que representan perifericos manejados por drivers. Al cargar un modulo que implementa un driver de caracter o bloque, este registra un mayor/menor y usualmente `udev` crea una entrada en `/dev`. Por ejemplo, al

insertar el driver de una camara web puede aparecer `/dev/video0`. Los programas de usuario interactuan con el driver leyendo o escribiendo en ese nodo. No todos los modulos crean nodos en `/dev`; un modulo de sistema de archivos o uno de *netfilter* no requiere hacerlo.

Directorio de modulos instalados

Los modulos compilados para una version determinada del kernel se almacenan en `/lib/modules/<version>/kernel/`. Por ejemplo, para un kernel 5.15.0-50-generic existe:

```
/lib/modules/5.15.0-50-generic/kernel/
```

Alli hay subdirectorios por categoria (drivers de sonido, red, FS, etc.) con archivos `.ko`. Este es el repositorio que `modprobe` consulta.

- `ls -R /lib/modules/$(uname -r)/kernel`: lista recursivamente todos los modulos disponibles para el kernel actual.
- Archivos como `modules.dep` indican dependencias entre modulos (generados por `depmod`).
- Algunas distribuciones mantienen `modules.sign` o `modules.sig` con hashes o firmas de los modulos oficiales.

Herramientas de consulta

- `lsmod`: muestra modulos cargados (lee `/proc/modules`).
- `modinfo <modulo>`: consulta metadatos del modulo ubicado en `/lib/modules/<ver>` sin requerir ruta completa.
- `lspci -k`: indica para cada dispositivo PCI que driver (modulo) esta en uso y que otros podrian manejarlo.

En conjunto, `/proc`, `/dev` y `/lib/modules` permiten observar el estado del kernel y gestionar modulos. Por ejemplo, si `insmod` falla, se revisa `dmesg`; si `modprobe` no encuentra un modulo, se comprueba que exista en `/lib/modules` y se ejecuta `depmod`. Conocer estas rutas y archivos es esencial para diagnostico y desarrollo de drivers.

8. Drivers y deteccion de hardware (hwinfo, lspci, lsusb, etc.)

Los drivers de hardware en Linux suelen ser modulos del kernel que se encargan de controlar un dispositivo fisico especifico o una familia de dispositivos. Para saber que hardware tenemos y que modulo (driver) lo maneja, disponemos de varias herramientas:

- **lspci**: Lista los dispositivos PCI/PCIe del sistema (tarjetas graficas, controladoras SATA, tarjetas de red, etc. en el bus PCI). Use `-v` (verbose) o `-vv` para mas detalle. De especial utilidad es `lspci -k`, que muestra, para cada dispositivo PCI, que *kernel driver* esta usando y que *kernel modules* estan disponibles para ese dispositivo.

Ejemplos:

```
$ lspci -v -s 02:00.0      # informacion detallada de un dispositivo espec.
$ lspci -nn -k | grep -A3 Network
```

En la salida veremos lineas como: **Kernel driver in use: e1000e** y **Kernel modules: e1000e**, indicando que el dispositivo (p. ej. una tarjeta de red Intel) esta siendo manejado por el driver **e1000e**. Si hubiera modulos alternativos compatibles, aparecerian listados despues de **Kernel modules**. Para una GPU NVIDIA, podria verse **Kernel driver in use: nvidia** y **Kernel modules: nvidia, nouveau**. `lspci -k` agrega dos lineas clave segun documentacion de Red Hat: **Kernel driver in use** y **Kernel modules**.

- **lsusb**: Similar a `lspci` pero para dispositivos USB. Lista los dispositivos conectados al bus USB. No indica directamente el modulo, pero sabemos que muchos dispositivos USB (almacenamiento, camaras, etc.) son manejados por subsistemas como **usb-storage**, **uvcvideo**, etc. Se puede correlacionar con `dmesg` al conectar el dispositivo o revisando `/sys/bus/usb/devices`.
- **hwinfo**: Herramienta exhaustiva (comun en OpenSUSE, disponible en otras distros) que provee informacion detallada del hardware. `hwinfo -short` da un resumen y `hwinfo -all` da detalles, incluyendo que driver esta asociado a cada dispositivo. Instalable en Debian/Ubuntu con `sudo apt install hwinfo`.
- Otros: `lshw` (lista hardware en jerarquia y puede mostrar drivers), `inxi -Fxxx` (info general del sistema), `dmidecode` (info BIOS/UEFI), y archivos en `/sys` como `/sys/bus/pci/devices/...` que exponen relaciones dispositivo-modulo.

Deteccion de hardware y modulos cargados. Una practica comun es comparar la lista de modulos cargados (`lsmod`) entre diferentes sistemas para ver diferencias debidas al hardware. Por ejemplo, un equipo con GPU Intel tendra **i915**, otro con NVIDIA tendra **nvidia** o **nouveau**, otro con AMD tendra **amdgpu**. Tambien se puede listar “modulos disponibles pero no cargados”: p. ej. **pcspkr** puede existir en `/lib/modules` pero estar en `blacklist`. Estas comparaciones ayudan a mapear hardware ↔ modulo.

En Linux moderno, *udev* y el kernel auto-cargan modulos al detectar hardware: al conectar un USB, el kernel emite un *uevent* y *udev* carga el driver adecuado si esta disponible. Si un dispositivo no funciona, se puede buscar su ID con `lspci -nn` o `lsusb -v` y averiguar que modulo lo soporta.

Resumen: con estas herramientas mapeamos hardware y modulos, diagnostico esencial: “¿por que no funciona el WiFi?” → revisamos `lspci -k`; si el driver no esta en uso, sabemos que modulo falta o no se cargo.

9. Caso de estudio: Secure Boot, GRUB y el parche de Microsoft (Análisis del artículo de Ars Technica)

Como aplicación de los conceptos de módulos, firmas y Secure Boot, analicemos un incidente real mencionado en un artículo de Ars Technica: un parche de Microsoft relacionado con Secure Boot causó problemas en sistemas Linux dual-boot a mediados de 2024.

Contexto

Existía una vulnerabilidad en GRUB2 (“BootHole”, 2020) que permitía sortear Secure Boot. Microsoft, que gestiona la base de datos de firmas UEFI (vía Windows Update), preparó un parche para revocar componentes vulnerables. Tras dos años de pruebas, el parche se lanzó en agosto de 2024. Se esperaba que solo afectara a sistemas Windows-only, pero impactó a equipos con dual-boot.

Sintoma

Después de instalar la actualización, muchos usuarios no pudieron arrancar Linux con Secure Boot habilitado. Apareció el mensaje:

```
Verifying shim SBAT data failed: Security Policy Violation.  
Something has gone seriously wrong: SBAT self-check failed: Security Policy Violation.
```

Esto indica que *shim* (bootloader firmado que carga GRUB) se bloqueó a sí mismo al quedar en la lista de revocación SBAT incluida por Microsoft.

Consecuencia

Los sistemas no podían iniciar Linux (ni siquiera GRUB); algunos quedaban solo con Windows. La solución inmediata fue deshabilitar Secure Boot en el firmware UEFI. Alternativa más técnica: actualizar shim/GRUB a una versión no revocada. Microsoft detuvo temporalmente la propagación del parche en ciertos equipos tras reportes.

Lecciones

- La cadena de confianza de Secure Boot llega hasta shim/GRUB: si se revoca, el SO no carga.
- Los módulos del kernel firmados también están sujetos a revocación; un módulo crítico con firma revocada sería rechazado.
- Mantener actualizados los componentes de arranque de Linux en dual-boot es vital para evitar bloqueos tras actualizaciones de seguridad.
- El rol de Microsoft como autoridad de firmas Secure Boot puede impactar a Linux; requiere coordinación con las distribuciones.

Este caso demuestra como un cambio en las reglas de confianza (revocar un certificado) puede impedir cargar un SO, similar a lo que ocurre si intentamos cargar un modulo no firmado o con firma desconocida: el kernel lo rechazara por “violacion de politica de seguridad”. Comprender estos mecanismos es esencial al desarrollar o instalar modulos en sistemas con Secure Boot.

10. Que es checkinstall y para que sirve?

`checkinstall` es una utilidad que reemplaza temporalmente a la invocacion tradicional de `make install`. Su proposito principal es **generar un paquete binario** (DEB, RPM o TGZ, segun la distribucion) a partir de la instalacion que normalmente realizaria un script `Makefile`. En lugar de copiar archivos directamente al sistema de archivos, `checkinstall` intercepta el proceso, registra todos los archivos instalados y crea un paquete nativo:

- **Trazabilidad:** mantiene un registro exacto de los archivos instalados, lo que permite una desinstalacion limpia mediante el gestor de paquetes.
- **Consistencia:** integra el software compilado manualmente dentro de la base de datos de paquetes de la distribucion, evitando conflictos y archivos “huerfanos”.
- **Portabilidad:** facilita distribuir el paquete a otros equipos con la misma distribucion sin recompilar.

Flujo simplificado de uso:

1. `./configure && make` (compila el proyecto)
2. `sudo checkinstall -pkgname=mi-paquete -pkgversion=1.0` (intercepta `make install` y crea el `.deb/.rpm`)
3. El paquete resultante aparece, por ejemplo, como `mi-paquete_1.0-1_amd64.deb`
4. Instalacion normal con `sudo dpkg -i mi-paquete_1.0-1_amd64.deb`

11. Empaquetar un “Hello World” con checkinstall

A continuacion se muestra un ejemplo paso a paso para empaquetar un programa *hello world* en C:

1. Codigo fuente

```
#include <stdio.h>

int main(void) {
    puts("Hello World!");
    return 0;
}
```

Listing 1: Archivo `hello.c`

2. Makefile minimal

```
CC := gcc
CFLAGS := -Wall -O2
TARGET := hello

all:
    $(CC) $(CFLAGS) hello.c -o $(TARGET)

install:
    install -Dm755 hello /usr/local/bin/hello

clean:
    $(RM) hello
```

Listing 2: Archivo Makefile

3. Crear el paquete

a) Compilar: `make`

b) Ejecutar:

```
sudo checkinstall --pkgname=hello --pkgversion=1.0 \
    --maintainer="tu@email.com" --pakdir=. \
    --install=no
```

c) `checkinstall` preguntara descripcion, categoria, dependencias, etc.

d) El resultado sera `hello_1.0-1_amd64.deb` (o equivalente RPM/TGZ segun distro).


e) Instalar con `sudo dpkg -i hello_1.0-1_amd64.deb`

4. Verificar

```
$ which hello
/usr/local/bin/hello
$ hello
Hello World!
$ dpkg -L hello | head
/.
/usr
/usr/local
/usr/local/bin
/usr/local/bin/hello
```

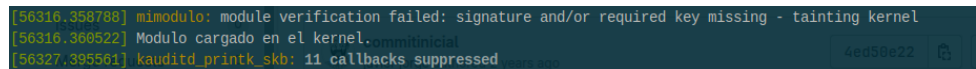
El paquete puede desinstalarse con `sudo apt remove hello` o `sudo dpkg -r hello`, asegurando limpieza total.

Nosotros usamos el modulo de GitLab, con los siguientes resultados



```
agustin: module main $ ls
Makefile mimodulo.c mimodulo.ko mimodulo.mod mimodulo.mod.c mimodulo.mod.o mimodulo.o modules.order Module.symvers
agustin: module main $ ls -lh mimodulo.
ls: cannot access 'mimodulo.': No such file or directory
agustin: module main $ ls -lh mimodulo.ko
-rw-rw-r-- 1 agustin agustin 156K May 25 21:29 mimodulo.ko
agustin: module main $ sudo insmod mimodulo.ko
agustin: module main $ lsmod | head -n 5
Module
Size Used by
mimodulo 12288 0
nf_conntrack_netlink 57344 0
xt_nat 12288 0
veth 45856 0
agustin: module main $ grep mimodulo /proc/modules
mimodulo 12288 0 - Live 0x0000000000000000 (0E)
```

Figura 1: Compilación y analisis del modulo.



```
[56316.358788] mimodulo: module verification failed: signature and/or required key missing - tainting kernel
[56316.360522] Modulo cargado en el kernel...mmibinicial
[56327.395561] kauditd_printk_skb: 11 callbacks suppressed
```

Figura 2: Log del kernel respecto al modulo propio.

12. Mejorar la seguridad del kernel: evitar modulos no firmados y mitigar rootkits

Para fortalecer la seguridad del kernel es fundamental restringir la carga de modulos a aquellos que esten firmados y provenientes de fuentes confiables. A continuacion se enumeran acciones concretas respaldadas por la bibliografia consultada¹:

1. **Habilitar Secure Boot:** garantiza que el kernel y los modulos solo se carguen si estan firmados con claves de confianza (seccion 4).
2. **Compilar el kernel con CONFIG_MODULE_SIG_FORCE=y:** obliga a que cada modulo tenga firma valida; el kernel rechazara los no firmados.
3. **Usar module.sig_enforce=1** en la linea de comandos del kernel: equivalente en tiempo de arranque si el kernel ya se compilo con CONFIG_MODULE_SIG.
4. **Implementar Lockdown Mode:** en kernels recientes (CONFIG_LOCKDOWN_LSM), opcionalmente activa restricciones adicionales que bloquean la manipulacion de memoria del kernel desde user space, incluso para root.
5. **Activar SELinux/AppArmor perfiles estrictos:** reduce superficie de ataque para procesos potencialmente comprometidos que podrian intentar insertar modulos.
6. **Deshabilitar LOAD_PIN_ENFORCE** (Lockdown PIN) solo si se requiere cargar modulos externos; de lo contrario, mantenerlo en enforcing.
7. **Depurar modulos propios:** compilar con `-fstackprotectorstrong`, `-D_FORTIFY_SOURCE=2`, habilitar CFI (*Control Flow Integrity*) donde sea posible.
8. **Monitorear firmas y blacklist:**
 - Revisar `/proc/sys/kernel/module_sig_enforce` (1 = obligatorio).
 - Gestionar `.blacklist` del keyring para revocar claves comprometidas.
9. **Herramientas de deteccion de rootkits:** instalar y ejecutar periodicamente `rkhunter` y `chkrootkit`. Configurar alertas ante cambios en modulos del kernel o binarios criticos.

Estas medidas, combinadas con un proceso de firmas propio (ver seccion de firma de modulos) y la generacion de paquetes reproducibles mediante herramientas como `checkinstall` o `dpkg-buildpackage`, minimizan el riesgo de rootkits que se basan en la insercion de modulos no autorizados.

¹Referencias: *Linux Kernel Documentation: Kernel Lockdown*, *Red Hat Secure Boot Guide*, *The Art of Linux Kernel Design*.