

Introducción

Hola a todos.

En este tutorial voy a intentar mostrar las bases de como iniciar una depuración en tiempo real del kernel del Linux. Tras mucho buscar por la red voy a intentar poner todos los pasos que he tenido que dar, espero que a alguien más le sirva.

Se va a considerar que el lector sabe compilar el kernel de Linux.

Configurando el núcleo

Uno de los principales problemas en la depuración de un driver/modulo es que no puedes establecer puntos de ruptura, ver el valor de las variables en un momento dado, ... La depuración básica cuando depuras un módulo sigue siendo el uso del “printk”, el cual deja una cadena de texto dentro del fichero /var/log/message o en tiempo de ejecución en /dev/kmsg (aconsejo usar este último).

Si la zona a depurar se encuentra dentro de una gestión de IRQ o simplemente se llama muchísimas veces a esa zona, la cantidad de mensajes puede ser tal que realmente puede ser de poca ayuda. O fallos por ejemplo de errores de segmentación que te pueden dejar la máquina congelada y no sabes de donde puede venir. Es el momento en el que uno echa de menos un depurador.

Las técnicas que dispone Linux para ayudar a la depuración del kernel es:

- KGDB
- KDB

El primero es un depurador el cual permite depurar desde una máquina remota conectado por puerto serie.

El segundo realmente es una Shell con comandos mínimos que pueden servir para por ejemplo ver los módulos cargados, o los procesos.

En la máquina remota se lanzaría una instancia de GDB y se indicaría que la depuración se realizará por puerto serie.

En versiones antiguas del kernel estos módulos no venían y había que añadirlos via comando patch, ya, por lo menos con la versión de Kernel 3.10 (la que he utilizado) vienen en la rama principal del núcleo, solo debemos de asegurarnos de que lo tenemos activo.

Si desde el código del kernel ejecutamos make menuconfig, tenemos que asegurarnos de tener activados por lo menos los siguientes parámetros dentro de la rama de kernel hacking

```
Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted
letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*]
built-in [ ] excluded <M> module < > module capable
↑ (-)
[*] Debug Filesystem
[*] Run 'make headers_check' when building vmlinux
[*] Enable full Section mismatch analysis
[*] Kernel debugging
[*] Debug shared IRQ handlers
[*] Detect Hard and Soft Lockups
[*] Panic (Reboot) On Hard Lockups
[ ] Panic (Reboot) On Soft Lockups
[*] Panic on Oops
[*] Detect Hung Tasks
(120) Default timeout for hung task detection (in seconds)
[ ] Panic (Reboot) On Hung Tasks
(0) panic timeout
[*] Collect scheduler debugging info
↓ (+)
<Select> <Exit> <Help> <Save> <Load>
```

Activación de Kernel debugging

```
↑ (-)
[ ] Lock usage statistics
[ ] Sleep inside atomic section checking
[ ] Locking API boot-time self-tests
[ ] Stack utilization instrumentation
[*] kobject debugging
[*] Verbose BUG() reporting (adds 70K)
[*] Compile the kernel with debug info
[ ] Reduce debugging information
[ ] Debug VM
```

Indicamos que la compilación se cree con símbolos de depuración

```
[*] Sample kernel code --->
[*] KGDB: kernel debugger --->
< > Test functions located in the string_helpers module at runtime
```

Revisar que el KGDB está con * es decir se compila de modo Build in y no como módulo, esto es que se tiene cargado desde el inicio.

```

[*] Check for stack overflows
[*] Early printk via the EFI framebuffer
[*] Export kernel pagetable layout to userspace via debugfs
[ ] Write protect kernel read-only data structures
[ ] Set loadable kernel module data as NX and text as RO
<M> Testcase for the NX non-executable stack feature
[ ] Set upper limit of TLB entries to flush one-by-one

```

Para poder disponer de breakpoint software (los BP de siempre) sobre módulos cargados manualmente (lo que vamos a hacer) tenemos que tener desactivadas estas 2 opciones, de lo contrario la sección .text del módulo se cargará como ReadOnly y no podremos poner puntos de rupturas normales, solo trabajar con hw breakpoint o de acceso.

```

-- KGDB: kernel debugger
<*> KGDB: use kgdb over the serial console
[*] KGDB: internal test suite
[ ] KGDB: Run tests on boot
[*] KGDB: Allow debugging with traps in notifiers
[*] KGDB_KDB: include kdb frontend for kgdb
[*] KGDB_KDB: keyboard as input device
(0) KDB: continue after catastrophic errors

```

La configuración que me venía por defecto en el Centos 7 para el KGDB me venía bien, la muestro por si alguno quiere probar y le difiere.

Una vez comprobado eso solo hace falta compilar e instalar el núcleo. La instalación es estrictamente necesaria en la máquina TARGET.

La compilación dejará en el raíz del código del núcleo compilado un fichero llamado vmlinux (ojo no es el vmlinuz que solemos encontrar en el /boot). Este fichero es un fichero elf del núcleo con los símbolos de depuración en él. Es el fichero que usaremos en la máquina HOST.

Depurando vía Ethernet

Como hemos dicho antes, si alguno buscar referencias sobre KGDB por la red, Linux dota de posibilidad de depuración en tiempo real del Kernel vía remota usando un puerto Serie.

Si alguno busca un poco más en google, verá que versiones antiguas de KGDB disponía de extensiones para el uso de Ethernet, pero no sé muy bien porque motivo la versión que viene con el kernel 3.10 no dispone de esa opción, se retiró.

En el caso del trabajo, la máquina que se quiere depurar tiene problemas con los puertos series, así que esto tal cual no lo puedo usar, aparte que un puerto serie, aun configurado a 115200 baudios es lento, así que tocaba seguir buscando y di con este módulo.

kgdboe que lo podéis descargar: <http://sysprogs.com/VisualKernel/kgdboe/kgdboe.tgz>

Es un componente de la casa sysprogs. Uno de sus productos es VisualKernel, el cual al parecer hace uso de Visual Studio y facilita la vida a la hora de crear módulos de kernel de Linux y depurarlos. Para la conexión entre el VisualKernel y la máquina TARGET usan este módulo que ofrecen gratuitamente, el resto es de pago y curiosamente su producto es solo para Windows.

Este módulo lo instalaremos en la máquina TARGET donde tendremos que compilarlo para el núcleo que previamente habíamos compilado.

Conectando los 2 ordenadores

Para esta experiencia vamos a usar:

- a) VMWare con Centos 7 núcleo 3.10.0-299 (HOST)
- b) Portatil con Centos 7 y mismo kernel.(TARGET)

El kgdboe solo funciona por conexión de cable, use como HOST la VMWare ya que la que tengo es la versión 10 y tiene ciertos problemas con el Kernel 3.X. Tenía cierta inestabilidad y con esta configuración iba mejor.

Nos vamos al TARGET, descomprimos el paquete que nos bajamos y compilamos así:

```
# make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

Si todo fue bien tendremos un fichero llamado kdbgoe.ko

Tenemos que ver ahora como se llama la interfaz de Ethernet por el cual vamos a realizar la conexión, para ello podemos hacer uso del comando ifconfig

En mi caso tengo:

```
enp0s25: flags=4163<UP, BROADCAST, RUNNING, MULTICAST> mtu 1500
    inet 192.168.1.45 ...
```

Importante que se me olvidaba, la conexión la vamos a realizar via socket, al puerto UDP que indiquemos como ya veremos. Si tenéis algún tipo de cortafuegos, iptables, ... asegurados de que está permitida la conexión.

Bueno hecho este inciso seguimos con el TARGET.

Vamos a arrancar una consola y vamos a hacer un cat /dev/kmsg

Esta venta simplemente va a ser un chivato para verificar que la carga del kgdboe es correcta

Y en otra ventana hacemos

```
# insmod kgddboe device_name=enp0s25 force_single_core=1
```

No he especificado el puerto, así que toma por defecto el 31337

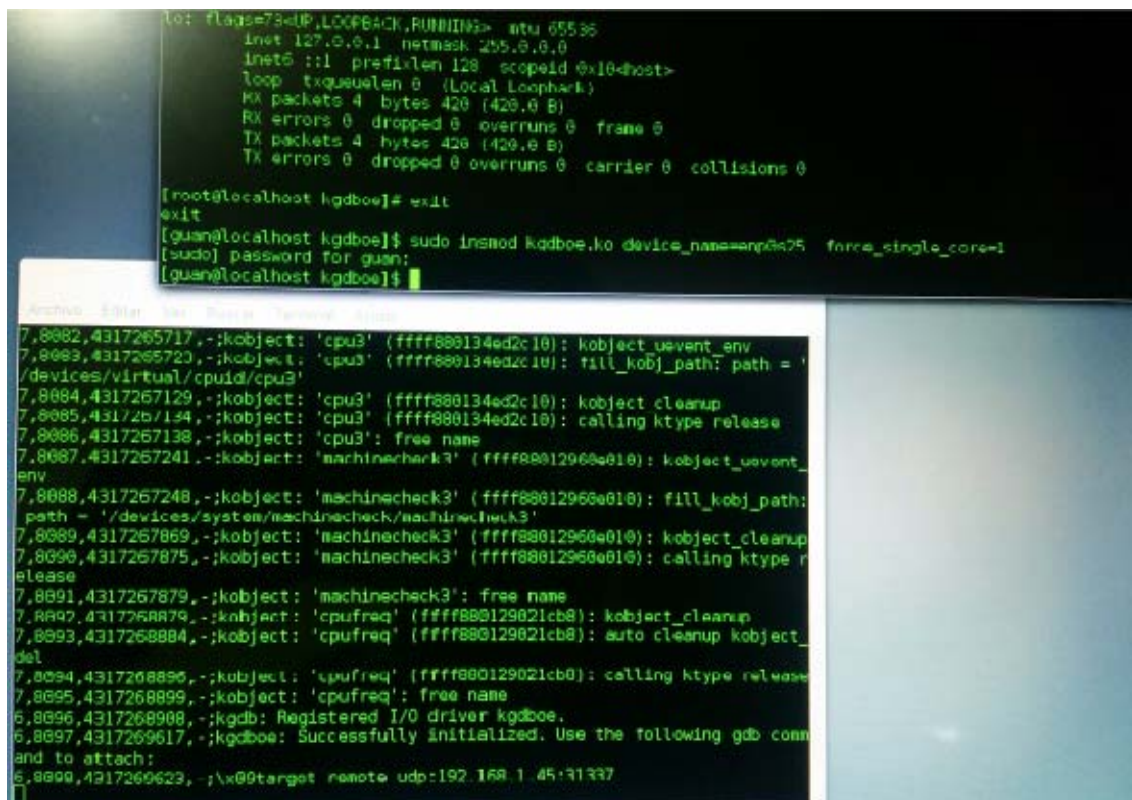
Lo de force_single_core=1 es una recomendación de fabricante de este módulo para evitar errores con el módulo que gestiona la red. Si necesitáramos depurar algo que deba ir en multicore pues poner a 0 ese parámetro.

Si todo fue bien veremos en la terminal donde hicimos el cat algo así:

kgdb: Registered I/O Driver kgddboe

kgddboe; Successfully initialized. Use the following gdb command to attach:

kgddboe: target remote udp:192.168.1.45:31337



```
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 0 (Local Loopback)
RX packets 4 bytes 420 (420.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 4 bytes 420 (420.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@localhost kgddboe]# exit
exit
[guan@localhost kgddboe]$ sudo insmod kgddboe.ko device_name=enp0s25 force_single_core=1
[sudo] password for guan:
[guan@localhost kgddboe]$
```

```
7.8082,4317265717,-;kobject: 'cpu3' (ffff880134ed2c10): kobject_uevent_env
7.8083,4317265720,-;kobject: 'cpu3' (ffff880134ed2c10): fill_kobj_path: path =
'/devices/virtual/cpu/cuid/cuid3'
7.8084,4317267129,-;kobject: 'cpu3' (ffff880134ed2c10): kobject_cleanup
7.8085,4317267134,-;kobject: 'cpu3' (ffff880134ed2c10): calling ktype release
7.8086,4317267138,-;kobject: 'cpu3': free name
7.8087,4317267241,-;kobject: 'machinecheck3' (ffff88012960e010): kobject_uevent_
env
7.8088,4317267248,-;kobject: 'machinecheck3' (ffff88012960e010): fill_kobj_path:
path = '/devices/system/machinecheck/machinecheck3'
7.8089,4317267869,-;kobject: 'machinecheck3' (ffff88012960e010): kobject_cleanup
7.8090,4317267875,-;kobject: 'machinecheck3' (ffff88012960e010): calling ktype r
elease
7.8091,4317267879,-;kobject: 'machinecheck3': free name
7.8092,4317268879,-;kobject: 'cpufreq' (ffff880129021cb0): kobject_cleanup
7.8093,4317268884,-;kobject: 'cpufreq' (ffff880129021cb0): auto cleanup kobject_
del
7.8094,4317268896,-;kobject: 'cpufreq' (ffff880129021cb0): calling ktype release
7.8095,4317268899,-;kobject: 'cpufreq': free name
6.8096,4317268908,-;kgdb: Registered I/O driver kgddboe.
6.8097,4317269617,-;kgddboe: Successfully initialized. Use the following gdb comm
and to attach:
6.8098,4317269623,-;\x00target remote udp:192.168.1.45:31337
```

Ahora nos vamos a la máquina HOST

Abrimos una terminal y nos vamos a donde teníamos el fichero vmlinux que se generó al compilar el kernel.

Y hacemos gdb vmlinux

Si todo fue bien nos debe haber cargado los símbolos del kernel.

Y dentro de gdb hacemos:

```
target remote udp:192.168.1.45:31337
```

Si todo fue bien nos parará y la máquina TARGET estará congelada.

```
[guan@PCAS linux-3.10.0-229.el7.centos.x86_64]$ gdb vmlinux
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-64.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /Datos/kernel-3.10.0-229.el7/linux-3.10.0-229.el7.centos.x86_64/vmlinux...done.
(gdb) target remote udp:192.168.1.45:31337
warning: The remote protocol may be unreliable over UDP.
Some events may be lost, rendering further debugging impossible.
udp:192.168.1.45:31337: La red es inaccesible.
(gdb) target remote udp:192.168.1.45:31337
udp:192.168.1.45:31337: La red es inaccesible.
(gdb) target remote udp:192.168.1.45:31337
Remote debugging using udp:192.168.1.45:31337
kgdb_breakpoint () at kernel/debug/debug_core.c:1043
1043      wmb(); /* Sync point after breakpoint */
(gdb) █
```

Con esto ya tenemos comunicación entre las 2 máquinas.

Ahora para poder parar la máquina TARGET cuando queramos pulsando Ctrl-C vamos a introducir el siguiente comando:

set remote interrupt-sequence Ctrl-C

Si aplicamos el commando c, haremos que la máquina TARGET siga su curso
Y haciendo Ctrl-C la volvemos a parar

```
(gdb) set remote interrupt-sequence Ctrl-C
(gdb) c
Continuing.
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
kgdb_breakpoint () at kernel/debug/debug_core.c:1043
1043      wmb(); /* Sync point after breakpoint */
(gdb) █
```

Para cerrar el debugger y que la máquina TARGET siga su curso normalmente, lo más aconsejable es usar primero el comando detach y luego quit.

```
kgdb_breakpoint () at kernel/debug/debug_core.c:1043
1043      wmb(); /* Sync point after breakpoint */
(gdb) detach
Detaching from program: /Datos/kernel-3.10.0-229.el7/linux
linux, Remote target
Ending remote debugging.
(gdb) quit
[guan@PCAS linux-3.10.0-229.el7.centos.x86_64]$ █
```

Creando módulo Dummy

Hasta ahora hemos conseguido la comunicación de los 2 PCs, pero para conseguir depurar un módulo propio aún queda un pequeño detalle.

GDB necesita conocer los símbolos del programa a depurar, y como el módulo lo vamos a instalar a mano, no tenemos los símbolos dentro de vmlinux.

Con el comando `add-symbol-file` podemos indicar la dirección de nuestro módulo, pero se queja de que no sabe en qué zona de memoria debe mapearlo lo que nos imposibilita poder poner breakpoint.

Si no necesitarías depurar el inicio del módulo, lo podéis cargar en memoria, ver en el target donde se instaló haciendo un `cat /sys/module/<tu modulo>/sections/<diferentes secciones>`

Con los valores obtenidos luego puedes cargar en gdb los símbolos y visualizar variables, o poner bps, pero la idea de este tutorial es mostrar como poder depurar desde el arranque de nuestro módulo, por lo que tocará buscar una zona donde esté en memoria para cargar los símbolos y aun no se haya lanzado.

Para esta demostración he preparado un módulo dummy con los siguientes ficheros:

Makefile

```
# To build modules outside of the kernel tree, we run "make"
# in the kernel source tree; the Makefile these then includes this
# Makefile once again.
# This conditional selects whether we are being included from the
# kernel Makefile or not.
ifeq ($(KERNELRELEASE),)

    # Assume the source tree is where the running kernel was built
    # You should set KERNELDIR in the environment if it's elsewhere
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    # The current directory is passed to sub-makes as argument
    PWD := $(shell pwd)

    KERNEL_MODULE_NAME := cls_module
    ccflags-y := -ggdb -O0 -std=gnu99
```

modules:

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

modules_install:

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
```



```

clean:
    rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c .tmp_versions

.PHONY: modules modules_install clean

else
    # called from kernel build system: just declare what our modules are
    obj-m := cls_module.o
endif

```

Fijaros que se ha activado las directivas para que se cree con los símbolos de depuración

```
ccflags-y := -ggdb -O0
```

cls_module.c

```

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/sched.h> /* current and everything */
#include <linux/kernel.h> /* printk() */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */

#include "cls_module.h" /* local definitions */

static void cls_module_exit(void){
    printk(KERN_ALERT "ADIOS CLS\n");
}

static int cls_module_init(void){
    printk(KERN_ALERT "HOLA CLS\n");
    return 0; //OK
}

/*-----
 *
 * Descripcion: Define valores que se mostrarán al hacer un modinfo, al mismo tiempo
 * que se mapea que funciones se deben de llamar al cargar el módulo (module_init) y al
 * descargar el módulo (module_exit)
 *-----
 */
/*****/
MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("GUAN DE DIO CLS");
MODULE_DESCRIPTION("CLS DRIVER DUMMY");

```


cls_module.h

```
#ifndef _CLS_MODULE_H_
#define _CLS_MODULE_H_

#define FALSE 0
#define TRUE !(FALSE)

#endif
```

Para compilarlo simplemente hacemos make y si todo fue bien se crea el .ko
Para verificar que funciona abrimos un terminal donde mostramos al chivato que nos muestra los printk haciendo cat /dev/kmsg

Y en la ventana donde tenemos el .ko hacemos sudo insmod cls_module.ko

```
[guan@PCAS CSL_module]$ make
make -C /lib/modules/3.10.0-229.el7.x86_64/build M=/home/guan/CSL_module
make[1]: se ingresa al directorio `/usr/src/kernels/3.10.0-229.el7.x86_64/build'
CC [M] /home/guan/CSL_module/cls_module.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/guan/CSL_module/cls_module.mod.o
LD [M] /home/guan/CSL_module/cls_module.ko
make[1]: se sale del directorio `/usr/src/kernels/3.10.0-229.el7.x86_64/build'
[guan@PCAS CSL_module]$ sudo insmod cls_module.ko
[sudo] password for guan:
[guan@PCAS CSL_module]$
```

Y para quitarlo de memoria hacemos

sudo rmmod csl_module

```
[guan@PCAS CSL_module]$ sudo insmod cls_module.ko
[sudo] password for guan:
[guan@PCAS CSL_module]$ sudo rmmod csl_module
[guan@PCAS CSL_module]$
```

Para ver que están los símbolos de depuración hacemos

nm csl_module.ko

```
[guan@PCAS CSL_module]$ nm cls_module.ko
0000000000000000 T cleanup_module
0000000000000000 t cls_module_exit
0000000000000020 t cls_module_init
U __fentry__
0000000000000020 T init_module
000000000000007c r __module_depends
U printk
0000000000000000 D __this_module
000000000000001d r __UNIQUE_ID_author1
0000000000000000 r __UNIQUE_ID_description2
0000000000000034 r __UNIQUE_ID_license0
0000000000000049 r __UNIQUE_ID_rhelversion2
0000000000000059 r __UNIQUE_ID_srcversion1
0000000000000085 r __UNIQUE_ID_vermagic0
0000000000000000 r __versions
[guan@PCAS CSL_module]$
```

Ahora solo tenemos que copiar el .ko, o todo según os sea más fácil a la máquina host, ya que cuando carguemos los símbolos gdb los buscará en la máquina donde gdb se está ejecutando.

Depurando módulos propios

Recargamos el depurador y estamos parados en el breakpoint inicial

```
[guan@PCAS linux-3.10.0-229.el7.centos.x86_64]$ gdb vmlinux
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-64.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/g
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show cop
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /Datos/kernel-3.10.0-229.el7/linux-3.10.0-229.el
6_64/vmlinux...done.
(gdb) target remote udp:192.168.1.45:31337
warning: The remote protocol may be unreliable over UDP.
Some events may be lost, rendering further debugging impossible.
udp:192.168.1.45:31337: La red es inaccesible.
(gdb) target remote udp:192.168.1.45:31337
Remote debugging using udp:192.168.1.45:31337
kgdb_breakpoint () at kernel/debug/debug_core.c:1043
1043          wmb(); /* Sync point after breakpoint */
(gdb) set remote interrupt-sequence Ctrl-C
(gdb)
```

La función en Linux encargada de cargar los módulos es “load_module”, así que ponemos un breakpoint en esa función

```
(gdb) b load_module
Breakpoint 1 at 0xffffffff810dab60: file kernel/module.c, line 3305.
(gdb) c
Continuing.
```

Ahora en el target cargamos el módulo manualmente con insmod como antes

```
[New Thread 5145]
[Switching to Thread 5145]

Breakpoint 1, load_module (info=info@entry=0xffff880134d2bef0,
    uargs=0x0 <irq_stack_union>, uargs@entry=0x7fb08eca4d89 "",
    flags=flags@entry=0) at kernel/module.c:3305
3305     {
(gdb)
```

Y aquí estamos parados. Con el comando list podemos ver el código en C del módulo.

Aquí voy a ir un poco más rápido.

Si a uno le da por revisar el código verá que la última línea hace una llamada a `do_init_module()`

```
3428      /* Done! */
3429      trace_module_load(mod);
(gdb) l
3430
3431      return do_init_module(mod);
3432
3433  bug_cleanup:
3434      /* module_bug_cleanup needs module_mutex protection */
3435      mutex_lock(&module_mutex);
```

Viendo la filosofía de otros módulos del Kernel, la verdad es que pensaba que allí se hacía la llamada a la función inicial de nuestro módulo y el primero instinto fue poner un bp en esa línea de código. Ya en una de las pruebas desesperado con el módulo que usaba en el trabajo, como era una interfaz con el tty, puse un bp en una de las funciones del tty que usaba en el inicio y mi sorpresa cuando paró en la función del tty antes que en la del return del `load_module`.

Me puse a ejecutar línea a línea con el comando `next` y vi que parando en la siguiente línea el módulo ya se encontraba en memoria y no se había ejecutado.

```
3414      /* Module is ready to execute: parsing args may do that. */
3415      err = parse_args(mod->name, mod->args, mod->kp, mod->num_kp,
3416                     -32768, 32767, &ddebug_dyndbg_module_param_cb);
3417      if (err < 0)
3418          goto bug_cleanup;
3419
```

Así que ponemos un breakpoint en `parse_args` y damos a `continue`

```
continuing.
Breakpoint 2, parse_args (doing=doing@entry=0xfffffffffa001a0b8 "cls_module",
args=0x0 <irq_stack_union>, params=0x0 <irq_stack_union>, num=0,
min_level=min_level@entry=-32768, max_level=max_level@entry=32767,
unknown=0xffffffff812f8550 <ddebug_dyndbg_module_param_cb>)
at kernel/params.c:187
187      {
(gdb)
```

Si miramos la carga de los módulos haciendo uso de KDB

```
{ dm_mod dm_mod }
(gdb) monitor lsmod
Module                Size  modstruct    Used by
cls_module            946  0xfffffffffa001a0a0  1  (Loading) 0xfffffffffa001a000 [ ]
```

Ahí vemos ya nuestro módulo en memoria.

El primer argumento es el puntero al inicio de la estructura `modstruct` y navegando por ella se puede obtener mucha información como por ejemplo las diversas secciones de nuestro módulo.

El segundo argumento es la dirección de memoria de la sección `.text`.

En este ejemplo solo vamos a hacer uso de esta sección.

```
(gdb) add-symbol-file /home/guan/CSL_module/cls_module.ko 0xffffffffa001a000
add symbol table from file "/home/guan/CSL_module/cls_module.ko" at
      .text_addr = 0xffffffffa001a000
(y or n) y
Reading symbols from /home/guan/CSL_module/cls_module.ko...done.
(gdb) █
```

Y ahora si podemos poner un bp en el comienzo

```
(gdb) b cls_module_init

Breakpoint 3 at 0xffffffffa001a020: file /home/guan/CSL_module/cls_module.c, line 20.
(gdb) info b
Num      Type             Disp Enb Address              What
1        breakpoint      keep y   0xffffffff810dab60  in load_module
                        breakpoint already hit 1 time
                        at kernel/module.c:3305
2        breakpoint      keep y   0xffffffff81094200  in parse_args
                        breakpoint already hit 1 time
                        at kernel/params.c:187
3        breakpoint      keep y   0xffffffffa001a020  in cls_module_init
                        at /home/guan/CSL_module/cls_module.c:20
(gdb) █
```

Damos a continuar y

```
Breakpoint 3, cls_module_init () at /home/guan/CSL_module/cls_module.c:20
20     static int cls_module_init(void){
(gdb) l
15         printk(KERN_ALERT "ADIOS CLS\n");
16     }
17
18
19
20     static int cls_module_init(void){
21
22         printk(KERN_ALERT "HOLA CLS\n");
23
24
(gdb) l
25         return 0; //OK
26     }
27
```

Aquí nos vemos ya parados y con los símbolos cargados de nuestro módulo. Todo listo para la depuración desde cero.

Espero que os haya gustado.

Saludos a todo CLS, en especial a Boken, Nahuel y Shaddy que me han estado sufriendo hasta que hemos llegado a este tuto.

