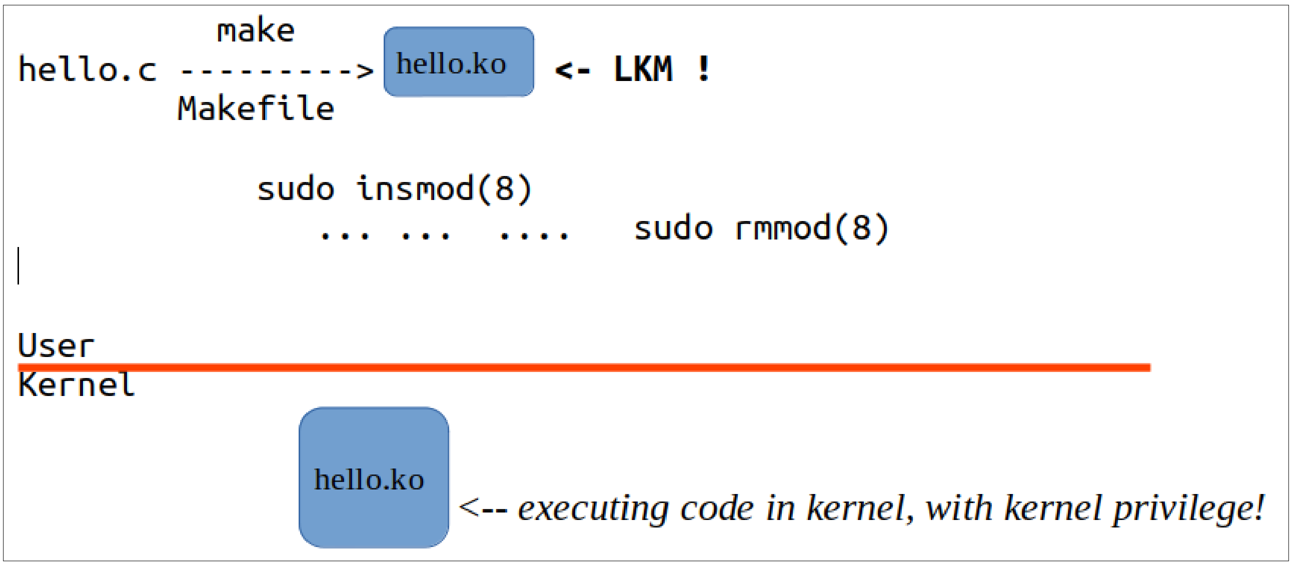
A Practical Guide to Writing Loadable Kernel Modules (LKMs)

Kaiwan N Billimoria
kaiwanTECH, Bangalore, India.

Loadable Kernel Modules (LKMs)

- Loadable kernel modules (LKMs) are programs that can be dynamically loaded and unloaded from the Linux kernel
- The Linux kernel is has a *modular* architecture (in this type, the kernel does not has to be re-built every time new functionality is added or some functionality removed)
- Device drivers are a typical example of kernel code written as LKMs
- When writing kernel code, programming is complex and debugging is difficult; not only that, to incrementally test every change, a new kernel will have to be built & installed, and the system rebooted - every time!
- A far better approach would be to code the kernel functionality as loadable modules; they will be built outside of the kernel, inserted into the kernel, tested and removed - all without rebuilding the kernel or rebooting the system.
- Also, an end-user would prefer to use new functionality built in this manner..



What code should be a kernel module, an LKM? Which can thus also be out-of-tree code (as is typically evidenced / required / desired by SoC vendors trying to differentiate their Android-based offerings):

- peripheral (chip) device drivers
- filesystems
- some types of networking services
  - drivers
  - firewalls

OTOH, biased toward ARM(64) (src):

"… before the kernel can load a single module, it must be able to boot to a point where it has the platform in a known, stable state and is able to mount a RAM-based root filesystem. That can only happen if the drivers needed to boot that far are built into the kernel itself. Thus, the generic kernel contains a long list of platform-specific drivers to configure clocks, pin controllers, and more; without them, the kernel would never boot. The maintainers' policy has long stated that any drivers which are essential for the boot process must be built into the kernel itself. …"

We introduce the writing and building of our first "Hello, world" kernel module below. This section has been taken from the "open source" device driver author's "bible" for Linux - **"LINUX Device Drivers" by Allesandro Rubini, Jonathan Corbet and Greg Kroah-Hartman, 3rd Edition, published by O'Reilly & Associates.**
Chapter 2 "Building and Running Modules" section "Setting up Your Test System".

The book is available online here: https://lwn.net/Kernel/LDD3/ .

Setting Up Your Test System

Starting with this chapter, we present example modules to demonstrate programming concepts. (All of these examples are available on O'Reilly's FTP site, as explained in Chapter 1.)Building, loading, and modifying these examples are a good way to improve your understanding of how drivers work and interact with the kernel.

The example modules should work with almost any 2.6.x kernel, including those provided by distribution vendors. However, we recommend that you obtain a "mainline" kernel directly from the kernel.org mirror network, and install it on your system.
Vendor kernels can be heavily patched and divergent from the mainline; at times, vendor patches can change the kernel API as seen by device drivers. If you are writing a driver that must work on a particular distribution, you will certainly want to build and test against the relevant kernels. But, for the purpose of learning about driver writing, a standard kernel is best.

Regardless of the origin of your kernel, building modules for 2.6.x requires that you have a configured and built kernel tree on your system. This requirement is a change from previous versions of the kernel, where a current set of header files was sufficient. 2.6 modules are linked against object files found in the kernel source tree; the result is a more robust module loader, but also the requirement that those object files be available. So your first order of business is to come up with a kernel source tree (either from the kernel.org network or your distributor's kernel source package), build a new kernel, and install it on your system. For reasons we'll see later, life is generally easiest if you are actually running the target kernel when you build your modules, though this is not required.

You should also give some thought to where you do your module experimentation, development, and testing. We have done our best to make our example modules safe and correct, but the possibility of

bugs is always present. Faults in kernel code can bring about the demise of a user process or, occasionally, the entire system. They do not normally create more serious problems, such as disk corruption.
Nonetheless, it is advisable to do your kernel experimentation on a system that does not contain data that you cannot afford to lose, and that does not perform essential services. Kernel hackers typically keep a “sacrificial” system around for the purpose of testing new code.

So, if you do not yet have a suitable system with a configured and built kernel source tree on disk, now would be a good time to set that up. We’ll wait. Once that task is taken care of, you’ll be ready to start playing with kernel modules.

The Hello World Module

Many programming books begin with a “hello world” example as a way of showing the simplest possible program. This book deals in kernel modules rather than programs; so, for the impatient reader, the following code is a complete “hello world” module:

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

/* From the official kernel doc:
“Loadable kernel modules also require a MODULE_LICENSE() tag. This tag is neither a
replacement for proper source code license information (SPDX-License-Identifier) nor in
any way relevant for expressing or determining the exact license under which the source
code of the module is provided. [...]”
*/
MODULE_LICENSE("Dual BSD/GPL");

static int __init hello_init(void)
{
	printk(KERN_ALERT "Hello, world\n");
	return 0; /* success */
}
static void __exit hello_exit(void)
{
	printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

This module defines two functions, one to be invoked when the module is loaded into the kernel (hello_init)and one for when the module is removed (hello_exit). The module_init and module_exit lines use special kernel macros to indicate the role of these two functions.

Licensing

Do see: https://choosealicense.com/

Another special macro (MODULE_LICENSE) is used to tell the kernel that this module bears a free license (or not); without such a declaration, the kernel complains (and is "tainted") when the module is loaded.
[*Note- interestingly, when I once spelled the macro wrognly :-) see what I got when compiling!*

```
error: expected declaration specifiers or '...' before string constant
 MODULE_LICENCE("GPL");
]
```

See the header *include/linux/module.h* for a comment on Licensing:

```
...
/*
 * The following license idents are currently accepted as indicating free
 * software modules
 *
 *       "GPL"                          [GNU Public License v2 or later]
 *       "GPL v2"                       [GNU Public License v2]
 *       "GPL and additional rights"    [GNU Public License v2 rights and more]
 *       "Dual BSD/GPL"                 [GNU Public License v2
 *                                       or BSD license choice]
 *       "Dual MIT/GPL"                 [GNU Public License v2
 *                                       or MIT license choice]
 *       "Dual MPL/GPL"                 [GNU Public License v2
 *                                       or Mozilla license choice]
 *
 * The following other idents are available
 *
 *       "Proprietary"                  [Non free products]
 *
 * There are dual licensed components, but when running with Linux it is the
 * GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL
 * is a GPL combined work.
 *
 * This exists for several reasons
 * 1.    So modinfo can show license info for users wanting to vet their setup
 *       is free
 * 2.    So the community can ignore bug reports including proprietary modules
 * 3.    So vendors can do likewise based on their own policies
 */
#define MODULE_LICENSE(_license) MODULE_INFO(license, _license)
...
```

<<
Licensing

SPDX = Software Package Data Exchange (SPDX) standard

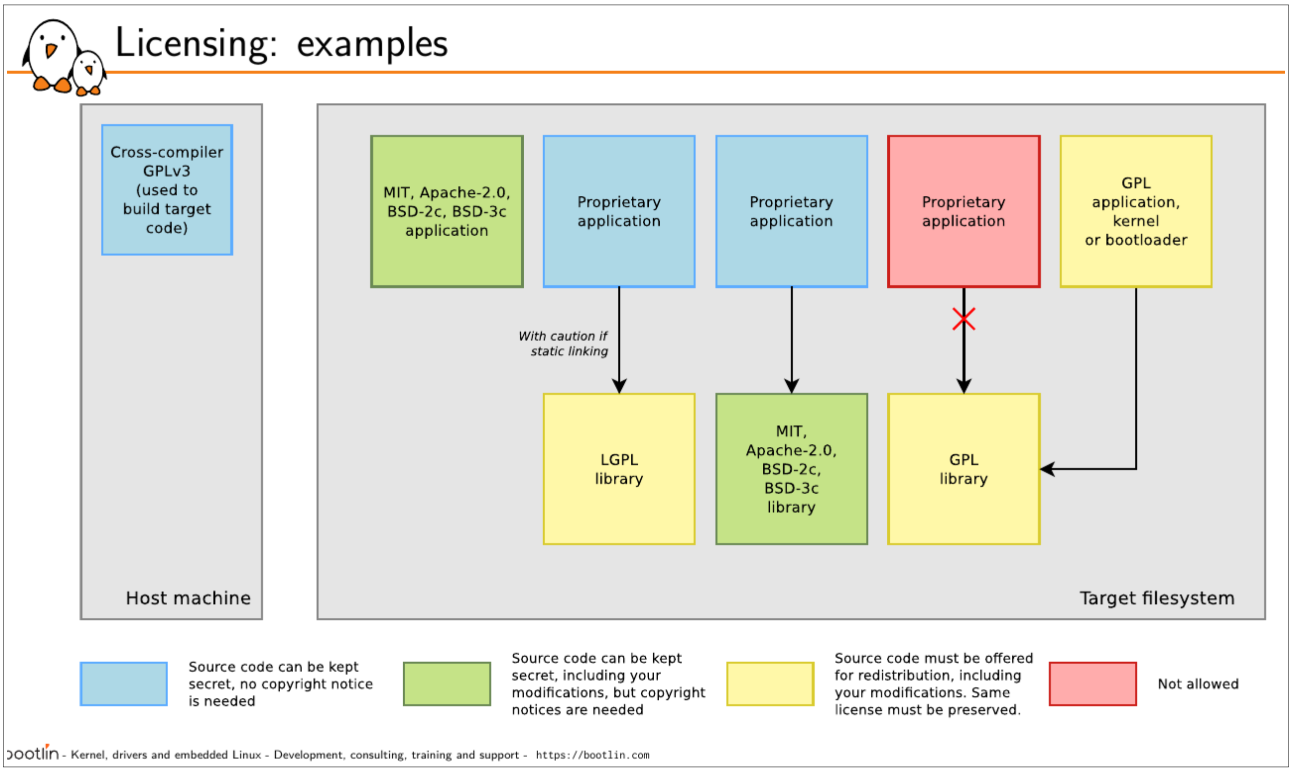
Official kernel doc:
Linux kernel licensing rules

SPDX identifiers in the kernel, Jon Corbet, Nov 2017, LWN

The 2 slides below are from the excellent Bootlin docs (here):

Dual licensing

- Some companies use a *dual licensing* business model, mainly for software libraries
- Their software is offered under two licenses:
  - A strong copyleft license, typically GPL, to encourage adoption of the software by the open-source world, allow the development and distribution of GPL licensed applications based on this library
  - A commercial license, offered against a fee, which allows to develop and distribute proprietary applications based on this library.
- Examples: Qt (only parts), MySQL, wolfSSL, Asterisk, etc.



Licensing and Proprietary Modules
Ref: *A stable bug fix bites proprietary modules, Jon Corbet, June 2021*

"… The kernel community's policy on loadable modules has, over the years, drawn criticism from many sides.

Some see even a grudging tolerance of proprietary modules as a weakening of the protections provided by the GPL and a cover for vendors that don't want to play by the rules.

Others see it as an impediment to the use of Linux in general that reduces available hardware support and makes users jump through unnecessary hoops.

The best way to judge this policy, though, is to look at what its results have been over nearly three decades.

Proprietary modules still exist, but they are in the minority; most hardware is supported with free drivers, and the situation seems to continue to slowly improve.
Vendors that have clung to proprietary modules in the past have found ways to change their approach; this might not have happened if those vendors had been excluded from the Linux community entirely. So, perhaps, making life uncomfortable for distributors of such modules while not trying to ban them outright may be the most productive policy in the long run."
>>

<<
FAQs:

1. What do the '__init' and '__exit' directives mean?

A. Essentially, they're (compiler) optimization attributes; they tell the compiler that the function (or memory) marked as such will be used only once (during initialization or exit)- hence, once done, the memory can be discarded from RAM.

__init => discard the function (from RAM) after it has run.
__exit => the function code under __exit() is not included in the kernel code for a built-in module, as built-in modules typically never use the exit function. In loadable modules this is ignored; the exit function is required during rmmod and hence is included.

Some more details here.

2. Module Load and Unload via system calls

The insmod and rmmod utility programs get the actual work done by issuing system calls:
insmod : init_module(2)
rmmod : delete_module(2)

In addition, the *finit_module(2)* system call can be used as well (it is typically used these days):

```
$ man init_module
...
   The  finit_module() system call is like init_module(), but reads the module to be loaded from
the file descriptor fd.  It is useful when the authenticity of a kernel module can be determined
from its location in the filesystem; in cases where that  is  possible,  the  overhead  of using
cryptographically  signed  modules  to  determine  the  authenticity of a module can be avoided.
The param_values argument is as for init_module().
...
```

Also, realize that all the module-manipulation programs are nothing but symbolic links to the program "**kmod**"!

...
***kmod** is a multi-call binary which implements the programs used to control Linux Kernel modules. Most users will only run it using its other names.*
...

```
$ kmod
missing command
kmod - Manage kernel modules: list, load, unload, etc
Usage:
       kmod [options] command [command_options]
...
...

kmod also handles gracefully if called from following symlinks:
  lsmod         compat lsmod command
  rmmod         compat rmmod command
  insmod        compat insmod command
  modinfo       compat modinfo command
  modprobe      compat modprobe command
  depmod        compat depmod command
$
```

The kernel-space return convention

Kernel code is expected to return 0 on success and an appropriate *negative* errno value on failure; it's called the 0/-E return convention.

errno - C Error Codes in Linux

- Base errno values [1 to 34] : */usr/include/asm-generic/errno-base.h*
- Extended errno values [35 to 133] */usr/include/asm-generic/errno.h*

All the Linux/C error codes are listed here-
https://gist.github.com/kaiwan/cd4985f3dbfeb8dfc44d1f0e4ee67dec

Only use a negative errno value within the legal range, else it won't show…

3. The printk: where does printk output go, etc?

Kernel logging: APIs and implementation - *From the kernel to user space logs*

http://stackoverflow.com/questions/4518420/where-does-output-of-print-in-kernel-go

A note on the printk Logging Level

The printk format string includes a printk "logging level" directive; you should use one of: (from *include/linux/kern_levels.h*)

```
#ifndef __KERN_LEVELS_H__
#define __KERN_LEVELS_H__

#define KERN_SOH        "\001"          /* ASCII Start Of Header */
#define KERN_SOH_ASCII  '\001'

#define KERN_EMERG      KERN_SOH "0"    /* system is unusable */
#define KERN_ALERT      KERN_SOH "1"    /* action must be taken immediately */
#define KERN_CRIT       KERN_SOH "2"    /* critical conditions */
#define KERN_ERR        KERN_SOH "3"    /* error conditions */
#define KERN_WARNING    KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE     KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO       KERN_SOH "6"    /* informational */
#define KERN_DEBUG      KERN_SOH "7"    /* debug-level messages */

#define KERN_DEFAULT    KERN_SOH "d"    /* the default kernel loglevel */
```

It is possible to read and modify the console loglevel using the text file */proc/sys/kernel/printk* .

The file hosts four integer values:

- the current loglevel *[all messages < this value appear on the console]*
  - *in effect* - KERN_EMERG, KERN_ALERT, KERN_CRIT, KERN_ERR – *automatically appear on the console by default*
- the default level for messages that lack an explicit loglevel
- the minimum allowed loglevel
- the boot-time default loglevel

```
# cat /proc/sys/kernel/printk
4       4       1       7
```

Writing a single value to this file changes the *current loglevel* to that value; thus, for example, you can cause all kernel messages to appear at the console by simply entering:

```
# echo “8 4 1 7” > /proc/sys/kernel/printk
```

It should now be apparent why the *hello.c* sample had the `KERN_ALERT` markers; they are there to make sure that the messages appear on the console.

The change in logging is only in effect for the current session; query with:

```
sysctl -n kernel.printk
4       4       1       7
```

Can make it permanent by wirting into the system config file (often at /etc/sysctl/conf) via:

```
/sbin/sysctl -w kernel.printk=“8 4 1 7”
```

<<
On Ubuntu at least, the relevant config file’s here:

/etc/sysctl.d/10-kernel-hardening.conf
Read /etc/sysctl.d/README.sysctl
>>

<<
printk output can be read from / using:

- kernel RAM-based circular log buffer via dmesg
- via /proc/kmsg (buffered printk records)
- non-volatile log file: */var/log/<distro-specific-logfile>*
  - *usually /var/log/syslog (Debian/Ubuntu-based)*
  - */var/log/messages (RedHat/Fedora-based)*
  - don't forget the new-ish systemd logging (use journalctl(1), as mentioned above)

>>

Recent kernels:
SHOULD use the ***pr_<level>***[1] macros as a convenience:

```
include/linux/printk.h
...
/*
 * These can be used to print at the various log levels.
 * All of these will print unconditionally, although note that pr_debug()
 * and other debug macros are compiled out unless either DEBUG is defined
 * or CONFIG_DYNAMIC_DEBUG is set.
 */
#define pr_emerg(fmt, ...) \
    printk(KERN_EMERG pr_fmt(fmt), ##__VA_ARGS__)
#define pr_alert(fmt, ...) \
    printk(KERN_ALERT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_crit(fmt, ...) \
    printk(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_err(fmt, ...) \
    printk(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warning(fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn pr_warning
#define pr_notice(fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
...
/* pr_devel() should produce zero code unless DEBUG is defined */ << the symbol DEBUG
       should be defined within the inline kernel for pr_devel() to emit any output! >>
#ifdef DEBUG
#define pr_devel(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
...
```

1 Where <*level*> is one of: emerg / alert / crit / err / warning / warn / notice / info / debug .

TIP

End printk's with a newline; if not, it's possible that – like the usermode printf buffer – it doesn't get flushed and thus the printk output doesn;t appear in the kernel log whne expected!

```
pr_info("Hello, world");    // NO !
pr_info("Hello, world\n");  // Yes
```

From my LKD book:

While on the subject of using the `pr_*()` macros, there's one called `pr_cont()`. Its job is to act as a continuation string, continuing the previous printk! This can be useful... here's an example of its usage:

```
// kernel/module.c
   if (last_unloaded_module[0])
       pr_cont(" [last unloaded: %s]",
               last_unloaded_module);
   pr_cont("\n");
```

We typically ensure that only the final `pr_cont()` contains the newline character.

How can I check the log level that the printk's are emitted at?

1. Via dmesg:
[sudo] dmesg -x
-x, --decode
Decode facility and level (priority) numbers to human-readable prefixes.

2. Via crash:
crash> log -m

NOTE!!!

Driver authors are highly encouraged to use the equivalent **dev_*() macros**; they're the same as the pr_*() except that the extra first parameter is now a pointer to the device structure (that all drivers will possess). The advantage is getting extra information regarding the driver subsystem / device also printed out…

```
include/linux/dev_printk.h

#define dev_emerg(dev, fmt, ...)                    \
    _dev_emerg(dev, dev_fmt(fmt), ##__VA_ARGS__)
#define dev_crit(dev, fmt, ...)                     \
    _dev_crit(dev, dev_fmt(fmt), ##__VA_ARGS__)
#define dev_alert(dev, fmt, ...)                    \
    _dev_alert(dev, dev_fmt(fmt), ##__VA_ARGS__)
#define dev_err(dev, fmt, ...)                      \
    _dev_err(dev, dev_fmt(fmt), ##__VA_ARGS__)
#define dev_warn(dev, fmt, ...)                     \
    _dev_warn(dev, dev_fmt(fmt), ##__VA_ARGS__)
#define dev_notice(dev, fmt, ...)                   \
    _dev_notice(dev, dev_fmt(fmt), ##__VA_ARGS__)
#define dev_info(dev, fmt, ...)                     \
    _dev_info(dev, dev_fmt(fmt), ##__VA_ARGS__)
```

Plus, the `dev_dbg()` as well...

<<
Why use the 'dev' pointer?
It has the kernel's driver core supply additional information as to the origin of the printk; for example, from an RTC (I2C) driver (for the DS3231 chip):

```
...
[ 2109.612806] ks3231 1-0068: IRQ! #5
...
```

Here, ks3231 is the driver name, 1-0068: 1 is the I2C bus# and 0x68 is the client chip – the DS3231 RTC chip's – I2C address! Useful.
>>

3A. The pr_fmt() and dev_fmt() macros

There's often a

```
#define pr_fmt(fmt) "%s:%s(): " fmt, KBUILD_MODNAME, __func__
```

or similar, at the very top of a kernel source file (or module source). It *must* be before the #include block; it's a way of forcing every printk() to adhere to the format specified by this *pr_fmt()* macro; thus, above, every printk will prepend the kernel module name and function it's currently in!

Eg.
```
[ 5980.719483] dht2x_kdrv:dht2x_read_sensors(): str_crc=4b
```

Further, the useful `pr_fmt()` 'prefixing' doesn't work with the `dev_*()` macros; for them, substitute the **dev_fmt()** (at the beginning of the source file)!

Eg.
```
#define dev_fmt(fmt) "%s(): " fmt, __func__

...

[ 6225.672886] dht2x_kdrv:dht2x_read_sensors(): str_crc=32
[ 6225.672904] dht2x 1-0038: dht2x_read_sensors(): crc obtd=0x32 crc=0x32
[ 6225.672920] dht2x 1-0038: dht2x_temp_show(): Temperature=24826 milliC
```

(BTW, using the `dev_*()` macros to emit printk's are the reason why the useful drivername bus#-device_addr prefix shows up!)

Also, there are:

- *printk_once*
- *printk_<level>_once*
- ~~*printk_ratelimited*~~
- *pr_<level>_ratelimited()* *<- recommended*

available.

Other useful print routines to dump a buffer:

- *print_hex_dump()*
- ***print_hex_dump_bytes()***
- *hex_dump_to_buffer()*

Eg. screenshot when using print_hex_dump_bytes() to see a buffer 's content:

```
[630715.232780] slab_custom:use_our_cache(): Our cache object (@ ffff9f1e64266c00, actual=ffff9f1e64266c00)
[630715.233839] obj: 00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.234542] obj: 00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.235339] obj: 00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.236063] obj: 00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.236796] obj: 00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.237534] obj: 00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.238521] obj: 00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.239236] obj: 00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.239936] obj: 00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.240650] obj: 00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.241360] obj: 000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.242061] obj: 000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.242767] obj: 000000c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.243471] obj: 000000d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.244180] obj: 000000e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.245174] obj: 000000f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.245876] obj: 00000100: 00 00 00 00 00 00 00 00 39 36 30 30 34 38 2e 39  ........960048.9
[630715.246577] obj: 00000110: 36 30 30 34 38 2c 30 2e 30 2c 32 32 39 2c 30 00  60048,0.0,229,0.
[630715.247279] obj: 00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.247996] obj: 00000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[630715.248723] obj: 00000140: 00 00 00 00 00 00 00 00                          ........
slab_custom $
```

3B. printk formats

IMP: see https://www.kernel.org/doc/Documentation/printk-formats.txt

3C. Security: the %pK in printk:

```
/*
 * !IMP! NOTE reg the (2.6.38 onward) security-conscious printk
 * formats!
  * SHORT story:
  * We need to avoid 'leaking' kernel addr to userspace (hackers
  * have a merry time!). Now %p will _not_ show the actual kernel        * addr,
but rather a hashed value.
  *
  * To see the actual addr use %px : dangerous!
  * To see the actual addr iff root, use %pK : tunable via
  *   /proc/sys/kernel/kptr_restrict :
  *               = 0 : hashed addr [default]
  *               = 1 : _and_ root, actual addr displayed!
  *               = 2 : never displayed.
  *
  * DETAILS:
  * See https://www.kernel.org/doc/Documentation/printk-formats.txt
  */
```

https://docs.kernel.org/core-api/printk-formats.html

"… the aim of printing the address is to provide more information for debugging, use %p and boot the kernel with the no_hash_pointers parameter during debugging, which will print all %p addresses unmodified. If you *really* always want the unmodified address, see %px below. …"

Do see:

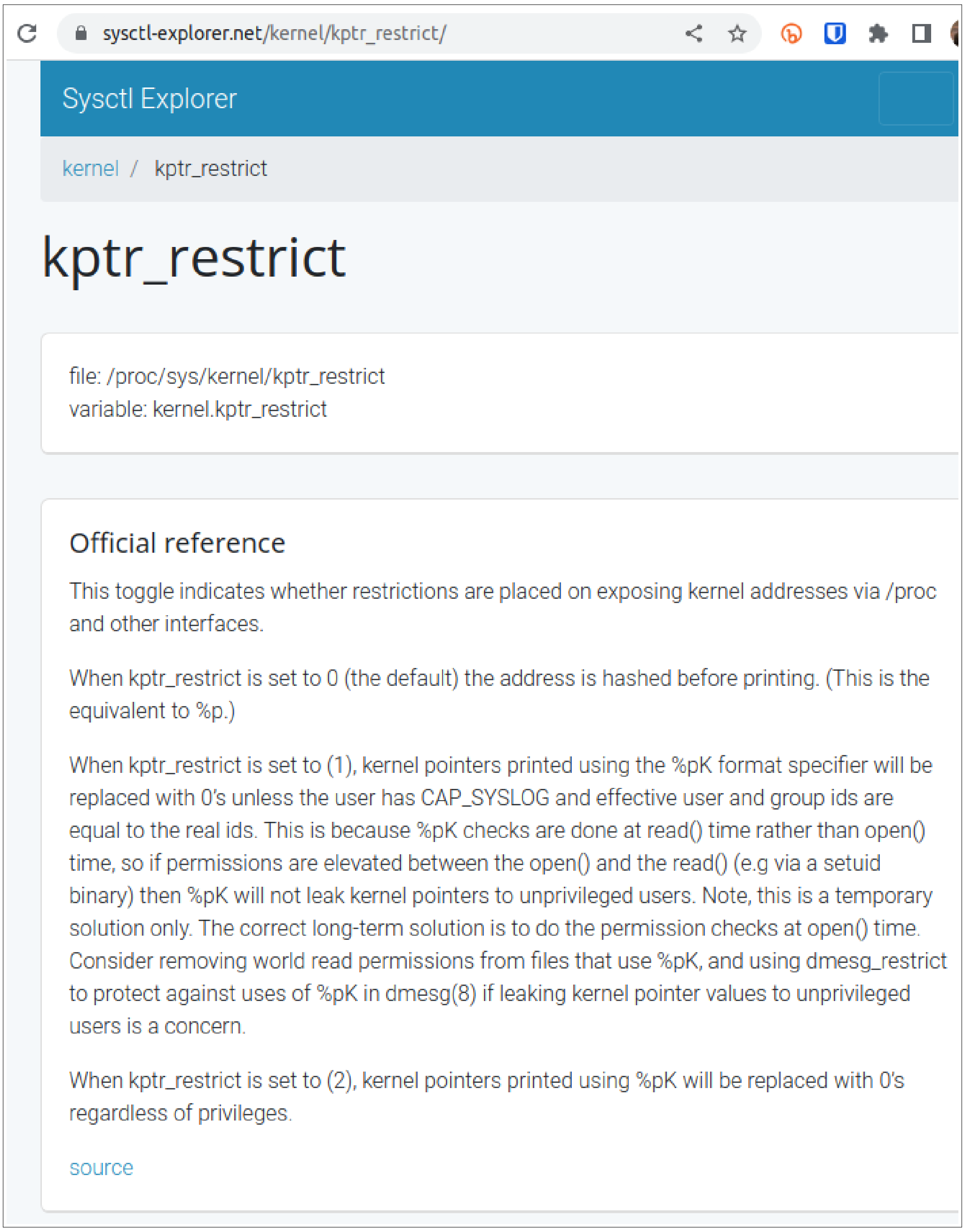
Serious Security: The Linux kernel bugs that surfaced after 15 years, Mar 2021

Commit: kptr_restrict for hiding kernel pointers from unprivileged users *[2.6.38-rc1]*

"Add the %pK printk format specifier and
`the /proc/sys/kernel/kptr_restrict sysctl.`

```
The %pK format specifier is designed to hide exposed kernel pointers,
specifically via /proc interfaces.  Exposing these pointers provides an
easy target for kernel write vulnerabilities, since they reveal the
locations of writable structures containing easily triggerable function
pointers.  The behavior of %pK depends on the kptr_restrict sysctl. …"
```

Useful! Sysctl Explorer : https://sysctl-explorer.net/


sysctl-explorer.net/kernel/kptr_restrict/

Sysctl Explorer

kernel / kptr_restrict

kptr_restrict

file: /proc/sys/kernel/kptr_restrict
variable: kernel.kptr_restrict

Official reference

This toggle indicates whether restrictions are placed on exposing kernel addresses via /proc and other interfaces.

When kptr_restrict is set to 0 (the default) the address is hashed before printing. (This is the equivalent to %p.)

When kptr_restrict is set to (1), kernel pointers printed using the %pK format specifier will be replaced with 0's unless the user has CAP_SYSLOG and effective user and group ids are equal to the real ids. This is because %pK checks are done at read() time rather than open() time, so if permissions are elevated between the open() and the read() (e.g via a setuid binary) then %pK will not leak kernel pointers to unprivileged users. Note, this is a temporary solution only. The correct long-term solution is to do the permission checks at open() time. Consider removing world read permissions from files that use %pK, and using dmesg_restrict to protect against uses of %pK in dmesg(8) if leaking kernel pointer values to unprivileged users is a concern.

When kptr_restrict is set to (2), kernel pointers printed using %pK will be replaced with 0's regardless of privileges.

source

In effect:

- kptr_restrict == 0
  - using %pK, not root : address is hashed and displayed
  - using %pK, got root : address is displayed
- kptr_restrict == 1
  - using %pK, not root : address is zeroed out
  - using %pK, got root : address is displayed (good for dev/debug)
- kptr_restrict == 2

- using %pK, not *or* got root : address is zeroed out! SECURE.

PRODUCTION : prevent info leaks

- ~~CFLAGS_EXTRA~~ (deprecated, use ccflags-y instead)
  ccflags-y += -UDEBUG and use only pr_debug() (they won't appear)
- USE %pK to print any pointers… (avoid printing pointers in any case)
- use sysctl -w to
  - set /proc/sys/kernel/kptr_restrict to 2
  - set /proc/sys/kernel/dmesg_restrict to 1
  - *set kernel config CONFIG_SECURITY_DMESG_RESTRICT to 'y'*

4. What if we do not return a value in the "init" code?

A. This is considered wrong; if you omit the return statement, a random value will be returned. In earlier kernel's, this caused the module to be *aborted* from kernel address space!

In recent kernels, however, the action isn't that drastic: insted the kernel complains via a 'noisy' printk. Eg:

```
...
static int __init hello_init(void)
{
	printk(KERN_ALERT "Hello, world\n");
#if 0
      // deliberately comment out the return statement
	return 0; // success
#endif
}
...
```

dmesg output:

```
...
[70444.430869] Hello, world
[70444.430873] do_init_module: 'hello'->init suspiciously returned 12, it should follow 
0/-E convention
[70444.430873] do_init_module: loading module anyway...
[70444.430876] CPU: 1 PID: 7743 Comm: insmod Tainted: G          OX 3.13.0-37-generic 
#64-Ubuntu
[70444.430878] Hardware name: LENOVO 4291GG9/4291GG9, BIOS 8DET42WW (1.12 ) 04/01/2011
[70444.430879]  00000000 00000000 de133e78 c1653867 fad4d00c de133f3c c10c43a6 c18358d4
[70444.430883]  c1671e72 fad4d00c 0000000c c1671e72 00000000 de133f94 f845f41c fad4d00c
[70444.430887]  d8116e88 f845f000 d8116ea4 00000001 00000001 fad4d048 00000000 de133f60
[70444.430891] Call Trace:
[70444.430898]  [<c1653867>] dump_stack+0x41/0x52
[70444.430902]  [<c10c43a6>] load_module+0x1156/0x18e0
[70444.430909]  [<c10c4c95>] SyS_finit_module+0x75/0xc0  << the finit_module() syscall has
                                                            loaded the lkm >>
[70444.430913]  [<c113a02b>] ? vm_mmap_pgoff+0x7b/0xa0
[70444.430921]  [<c1661bcd>] sysenter_do_call+0x12/0x12
...
```

5. Interpreting the kernel's TAINTED flags

The Linux kernel has a 'tainted' flag; if the value is zero, the kernel is considered 'clean', untainted. If non-zero, one can interpret the bits like so:

Source : https://www.kernel.org/doc/html/latest/admin-guide/tainted-kernels.html

Table for decoding tainted state

Bit	Log	Number	Reason that got the kernel tainted
0	G/P	1	proprietary module was loaded
1	_/F	2	module was force loaded
2	_/S	4	SMP kernel oops on an officially SMP incapable processor
3	_/R	8	module was force unloaded
4	_/M	16	processor reported a Machine Check Exception (MCE)
5	_/B	32	bad page referenced or some unexpected page flags
6	_/U	64	taint requested by userspace application
7	_/D	128	kernel died recently, i.e. there was an OOPS or BUG
8	_/A	256	ACPI table overridden by user
9	_/W	512	kernel issued warning
10	_/C	1024	staging driver was loaded
11	_/I	2048	workaround for bug in platform firmware applied
12	_/O	4096	externally-built ("out-of-tree") module was loaded
13	_/E	8192	unsigned module was loaded
14	_/L	16384	soft lockup occurred
15	_/K	32768	kernel has been live patched
16	_/X	65536	auxiliary taint, defined for and used by distros
17	_/T	131072	kernel was built with the struct randomization plugin

More detailed explanation for tainting

(the number in the left column is the bit #)

0. `G` if all modules loaded have a GPL or compatible license, `P` if any proprietary module has been loaded. Modules without a MODULE_LICENSE or with a MODULE_LICENSE that is not recognised by insmod as GPL compatible are assumed to be proprietary.
1. `F` if any module was force loaded by `insmod -f`, `' '` if all modules were loaded normally.
2. `S` if the oops occurred on an SMP kernel running on hardware that hasn't been certified as safe to run multiprocessor. Currently this occurs only on various Athlons that are not SMP capable.
3. `R` if a module was force unloaded by `rmmod -f`, `' '` if all modules were unloaded normally.
4. `M` if any processor has reported a Machine Check Exception, `' '` if no Machine Check Exceptions have occurred.
5. `B` If a page-release function has found a bad page reference or some unexpected page flags. This indicates a hardware problem or a kernel bug; there should be other information in the log indicating why this tainting occured.
6. `U` if a user or user application specifically requested that the Tainted flag be set, `' '` otherwise.
7. `D` if the kernel has died recently, i.e. there was an OOPS or BUG.
8. `A` if an ACPI table has been overridden.
9. `W` if a warning has previously been issued by the kernel. (Though some warnings may set more specific taint flags.)
10. `C` if a staging driver has been loaded.
11. `I` if the kernel is working around a severe bug in the platform firmware (BIOS or similar).
12. `O` if an externally-built ("out-of-tree") module has been loaded.
13. `E` if an unsigned module has been loaded in a kernel supporting module signature.
14. `L` if a soft lockup has previously occurred on the system.
15. `K` if the kernel has been live patched.
16. `X` Auxiliary taint, defined for and used by Linux distributors.
17. `T` Kernel was build with the randstruct plugin, which can intentionally produce extremely unusual kernel structure layouts (even performance pathological ones), which is important to know when debugging. Set at build time.

TIP
It's just simpler to run a script which will interpret the tainted flags (available on recent kernels); as an example, an Oops I got had the tainted flags value as 4097; to interpret this:

```
$ cd <kernel-src-tree-root>
$ cat /proc/sys/kernel/tainted
12289

$ tools/debugging/kernel-chktaint 12289
Kernel is "tainted" for the following reasons:
 * proprietary module was loaded (#0)
 * externally-built ('out-of-tree') module was loaded  (#12)
 * unsigned module was loaded (#13)
For a more detailed explanation of the various taint flags see
 Documentation/admin-guide/tainted-kernels.rst in the the Linux kernel sources
 or https://kernel.org/doc/html/latest/admin-guide/tainted-kernels.html
```

```
Raw taint value as int/string: 12289/'P           OE     '
$
```

Or,

```
$ tools/debugging/kernel-chktaint $(cat /proc/sys/kernel/tainted)
...
```

6. A Note on the 'dmesg' utility

When working in *console (non-graphical) mode*, the printk's emitted by the kernel appear on the console device directly (just like a printf) provided the "log-level" is of sufficient priority.

In GUI (X-Windows) mode, this does not happen; the kernel printk's have to be looked up. A convenient way to dump printk contents (the contents of the kernel printk ring buffer, to be precise) onto stdout is to use the *dmesg* utility.

Security Note:
There is a kernel build-time configuration option that can prevent non-root users using dmesg:

CONFIG_SECURITY_DMESG_RESTRICT:
This enforces restrictions on unprivileged users reading the kernel syslog via dmesg(8).
If this option is not selected, no restrictions will be enforced unless the dmesg_restrict sysctl is explicitly set to (1).
If you are unsure how to answer this question, answer N.

The man page on dmesg(1) is useful; dmesg takes several option switches – learn how to use them by looking up it's man page! (Of course you must have a recent enough version of dmesg installed; we're using:

```
$ dmesg --version
dmesg from util-linux 2.27.1
$
```

For example:

```
$ man dmesg
...
       -C : clear the ring buffer.

       -c : clear the ring buffer after first printing its contents.

       -D, --console-off
              Disable the printing of messages to the console.

       -d, --show-delta
              Display  the  timestamp and the time delta spent between messages.  If
used together with --notime then only the time delta without the timestamp is printed.

       -E, --console-on
              Enable printing messages to the console.
```

```
       -e, --reltime
              Display the local time and the delta in human-readable
              format.

       -F, --file file
              Read the messages from the given file.

       -f, --facility list
              Restrict output to the given (comma-separated) list of
              facilities.  For example:

                     dmesg --facility=daemon

              will print messages from system daemons only.  For all
              supported facilities see the --help output.

       -H, --human
              Enable human-readable output.  See also --color, --reltime
              and --nopager.

       -k, --kernel
              Print kernel messages.
...
        -l, --level list
              Restrict output to the given (comma-separated) list of
              levels.  For example:

                     dmesg --level=err,warn

              will print error and warning messages only.  For all
              supported levels see the --help output.

       -n, --console-level level
              Set the level at which printing of messages is done to the
              console.  The level is  a  level  number  or abbreviation
              of the level name.  For all supported levels see the --help
              output.
...
        -P, --nopager
              Do not pipe output into a pager.  A pager is enabled by
              default for --human output.

       -r, --raw
              Print the raw message buffer, i.e. do not strip the log-
              level prefixes.
      ...

       -T, --ctime
              Print human-readable timestamps.

              Be aware that the timestamp could be inaccurate!  The time
              source used for  the  logs  is  not  updated after system
              SUSPEND/RESUME.

       -t, --notime
              Do not print kernel's timestamps.
...
       -u, --userspace
```

```
              Print userspace messages.

       -w, --follow
              Wait for new messages.  This feature is supported only on
              systems with a readable /dev/kmsg (since kernel 3.5.0).

       -x, --decode
              Decode facility and level (priority) numbers to human-
              readable prefixes.

$
```

Eg.

```
$ dmesg --human --nopager --decode --userspace --kernel --ctime --show-delta

kern  :info  : [Fri Aug 19 11:43:12 2016 <    0.000000>] Initializing cgroup subsys cpuset
kern  :info  : [Fri Aug 19 11:43:12 2016 <    0.000000>] Initializing cgroup subsys cpu
kern  :info  : [Fri Aug 19 11:43:12 2016 <    0.000000>] Initializing cgroup subsys cpuacct
kern  :notice: [Fri Aug 19 11:43:12 2016 <    0.000000>] Linux version 4.2.0-42-generic (buildd@lgw01-54) (gcc version 5.2.1 20151010 (Ubuntu 5.2.1-22ubuntu2) ) #49-Ubuntu SMP Tue Jun 28 21:26:26 UTC 2016 (Ubuntu 4.2.0-42.49-generic 4.2.8-ckt12)

…

kern  :info  : [Fri Aug 19 11:43:12 2016 <    0.000000>] Initmem setup node 0 [mem 0x0000000000001000-0x000000003ffeffff]
kern  :debug : [Fri Aug 19 11:43:12 2016 <    0.000000>] On node 0 totalpages: 262030
kern  :debug : [Fri Aug 19 11:43:12 2016 <    0.000000>]   DMA zone: 64 pages used for memmap
kern  :debug : [Fri Aug 19 11:43:12 2016 <    0.000000>]   DMA zone: 21 pages reserved
kern  :debug : [Fri Aug 19 11:43:12 2016 <    0.000000>]   DMA zone: 3998 pages, LIFO batch:0
kern  :debug : [Fri Aug 19 11:43:12 2016 <    0.000000>]   DMA32 zone: 4032 pages used for memmap
kern  :debug : [Fri Aug 19 11:43:12 2016 <    0.000000>]   DMA32 zone: 258032 pages, LIFO batch:31
kern  :info  : [Fri Aug 19 11:43:12 2016 <    0.000000>] ACPI: PM-Timer IO Port: 0x4008

…

kern  :notice: [Fri Aug 19 11:43:20 2016 <    0.387369>] random: nonblocking pool is initialized
daemon:warn  : [Fri Aug 19 11:43:21 2016 <    0.890801>] systemd[1]: Failed to insert module 'kdbus': Function not implem
ented
…

$
```

A useful alias, perhaps:

```
$ alias dmesg='/bin/dmesg --human --decode --reltime --nopager --color=always'
```

7. Note : systemd logging

On a recent (Fedora 21, kernel ver 3.17.4-301.fc21.x86_64), the 'System Log Viewer' GUI app (*/usr/bin/logview*) has a README that displays the following:

"You are looking for the traditional text log files in /var/log, and they are gone?

Here's an explanation on what's going on:

You are running a systemd-based OS << *systemd* >> where traditional syslog has been
replaced with the Journal. The journal stores the same (and more)
information as classic syslog. To make use of the journal and access
the collected log data simply invoke "journalctl", which will output
the logs in the identical text-based format the syslog files in
/var/log used to be. For further details, please refer to
journalctl(1).

Alternatively, consider installing one of the traditional syslog
implementations available for your distribution, which will generate
the classic log files for you. Syslog implementations such as
syslog-ng or rsyslog may be installed side-by-side with the journal
and will continue to function the way they always did.

Thank you!

Further reading:
 man:journalctl(1)
 man:systemd-journald.service(8)
 man:journald.conf(5)
 http://0pointer.de/blog/
"

In fact, systemd, more specifically, **journald**, more or less completely replaces the older disparate logging systems. It can continue to work with syslog if required, or replace it.

journalctl is the utility to view logs in a wide variety of ways. Please see this article for useful detailed usage of the newer logging with systemd/journalctl:
How To Use Journalctl to View and Manipulate Systemd Logs**.**

<<
A few possibly useful aliases for journalctl:
NOTE! Using journalctl -k will have it show ONLY kernel messages

```
export SYSTEMD_PAGER=cat  # no pager !

#--- journalctl aliases
# jlog: current boot only, everything
alias jlog='/bin/journalctl -b --all --catalog --no-pager'
# jlogr: current boot only, everything, *reverse* chronological order
alias jlogr='/bin/journalctl -b --all --catalog --no-pager --reverse'
# jlogall: *everything*, all time; --merge => _all_ logs merged
alias jlogall='/bin/journalctl --all --catalog --merge --no-pager'
# jlogf: *watch* log, 'tail -f' mode
alias jlogf='journalctl -f'
# jlogk: only kernel messages, this boot
alias jlogk='journalctl -b -k -no-pager'

# show today's errors/fail/fatal messages
alias jlog_err_today="journalctl --no-pager --since today --grep 'fail|error|fatal' | sort
| uniq -c | sort --numeric --reverse --key 1"
# show today's errors/fail/fatal messages in JSON format
```

```
alias jlog_err_today_json="journalctl --no-pager --since today \
--grep 'fail|error|fatal' --output json|jq '._EXE' | \
sort | uniq -c | sort --numeric --reverse --key 1"
```
>>

Also, v useful:
See all kernel log messages emitted in the last hour:
`journalctl -k --since="1 hour ago"`

<<
How exactly can an application (or a kernel module) log messages into systemd?
Easy:

systemd for Developers III, Poeterring
…
The good thing is that getting log data into the Journal is not particularly hard, since there's a good chance the Journal already collects it anyway and writes it to disk. **The journal collects:**

1. **All data logged via libc `syslog()`**
2. **The data from the kernel logged with `printk()`**
3. **Everything written to STDOUT/STDERR of any system service**

This covers pretty much all of the traditional log output of a Linux system, including messages from the kernel initialization phase, the initial RAM disk, the early boot logic, and the main system runtime.

…

ALSO:

Use ***systemd-cat(1)***; whatever is sent to it's stdin is written to the journal!
Eg.

```
$ echo "hi $(date)" | systemd-cat
$ journalctl -e
...
Sep 05 17:20:04 osboxes bash[8914]: hi Tue Sep  5 05:20:04 PM IST 2023


 -e, --pager-end
           Immediately jump to the end of the journal inside the implied pager
tool. This implies -n1000  ...
```
>>

IMP Tip
When systemd-journald says '**`Forwarding to syslog missed 6 messages`**'...
the logs are visible in **/var/log/messages** or /var/log/syslog !
(The actual syslog conf file determining which file is written into is here:
(Yocto Poky): /etc/syslog.conf (busybox)
(Ubuntu): /etc/rsyslog.d/50-default.conf).

8. Permissions to insmod / rmmod

Traditionally, only superuser could perform *insmod / rmmod*. Now, with the modern Linux Capabilities model, we have a finer-grained permission allowing the same if that capability is set:

```
$ man 7 capabilities
       ...
       CAP_SYS_MODULE
            Load  and unload kernel modules (see init_module(2) and delete_module(2)); in
       kernels before 2.6.25: drop capabilities from the system-wide capability bounding set.
      ...
```

>>

[Back to the LDD3 book notes]

The printk function is defined in the Linux kernel and made available to modules; it behaves similarly to the standard C library function printf. The kernel needs its own printing function because it runs by itself, without the help of the C library. The module can call printk because, after insmod has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables, as detailed in the next section). The string KERN_ALERT is the priority of the message.*

* The priority is just a string, such as <1>, which is prepended to the printk format string. Note the lack of a comma after KERN_ALERT; adding a comma there is a common and annoying typo (which, fortunately, is caught by the compiler).

We've specified a high priority in this module, because a message with the default priority might not show up anywhere useful, depending on the kernel version you are running, the version of the klogd daemon, and your configuration. You can ignore this issue for now; we explain it in Chapter 4.

You can test the module with the *insmod* and *rmmod* utilities, as shown below. Note that **only the superuser** can load and unload a module.

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
CC [M] /home/ldd3/src/misc-modules/hello.o
Building modules, stage 2.
MODPOST
CC /home/ldd3/src/misc-modules/hello.mod.o
LD [M] /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
      << su is deprecated; use sudo /bin/bash to get a root shell >>
root# insmod ./hello.ko
Hello, world
root# rmmod hello
```

```
Goodbye cruel world
root#
```

What if module loading via insmod fails with this info in the kernel log?

```
Lockdown: insmod: unsigned module loading is restricted; see man
kernel lockdown.7
```

... If you got this message, the simplest way (to fix it) is to disable the UEFI SecureBoot from the PC/laptop boot menu to have your "hello-1" to be inserted. Of course you can go through complicated steps to generate keys, install keys to your system, and finally sign your module to make it work. However, this is not suitable for beginners. You could read and follow the steps in SecureBoot if you are interested.

Src

SIDEBAR :: Running the 'Hello, world' kernel module on an ARM-64 Raspberry Pi !

On the Raspberry Pi running **64-bit** Ubuntu 18.04.2 LTS !

```
rpi64 ~ $ lsb_release -a
No LSB modules are available.
Distributor ID:	Ubuntu
Description: Ubuntu 18.04.2 LTS
Release:     18.04
Codename:    bionic

rpi64 ~ $ uname -a
Linux ubuntu 4.15.0-1040-raspi2 #43-Ubuntu SMP PREEMPT Tue Jun 25 10:43:11 UTC 2019
aarch64 aarch64 aarch64 GNU/Linux

rpi64 ~ $ lscpu
Architecture:        aarch64
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               4
Model name:          Cortex-A53
Stepping:            r0p4
CPU max MHz:         1400.0000
CPU min MHz:         600.0000
BogoMIPS:            38.40
Flags:               fp asimd evtstrm crc32 cpuid
rpi64 ~ $
```

```
rpi64 helloworld $ make
make -C /lib/modules/4.15.0-1040-raspi2/build  M=/home/ubuntu/L2_kernel_trg/helloworld
modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-1040-raspi2'
@@@ Building for ARCH=arm64, CROSS_COMPILE=, KERNELRELEASE=4.15.0-1040-raspi2;
EXTRA_CFLAGS= -DDEBUG  @@@
  CC [M]  /home/ubuntu/L2_kernel_trg/helloworld/hello.o
  Building modules, stage 2.
@@@ Building for ARCH=arm64, CROSS_COMPILE=, KERNELRELEASE=4.15.0-1040-raspi2;
EXTRA_CFLAGS=-DDEBUG  @@@
  MODPOST 1 modules
  CC      /home/ubuntu/L2_kernel_trg/helloworld/hello.mod.o
  LD [M]  /home/ubuntu/L2_kernel_trg/helloworld/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-1040-raspi2'
rpi64 helloworld $ ls -l
total 32
-rw-rw-r-- 1 ubuntu ubuntu  778 Jul 17 12:14 Makefile
-rw-rw-r-- 1 ubuntu ubuntu    0 Aug 12 13:24 Module.symvers
-rw-rw-r-- 1 ubuntu ubuntu  529 Jul 17 12:14 hello.c
-rw-rw-r-- 1 ubuntu ubuntu 4784 Aug 12 13:24 hello.ko
-rw-rw-r-- 1 ubuntu ubuntu  857 Aug 12 13:24 hello.mod.c
-rw-rw-r-- 1 ubuntu ubuntu 3064 Aug 12 13:24 hello.mod.o
-rw-rw-r-- 1 ubuntu ubuntu 3384 Aug 12 13:24 hello.o
-rw-rw-r-- 1 ubuntu ubuntu   54 Aug 12 13:24 modules.order
rpi64 helloworld $ sudo insmod ./hello.ko
rpi64 helloworld $ dmesg |tail
...
[   23.430888] fuse init (API version 7.26)
[  320.530330] Hello, world
rpi64 helloworld $ lsmod |grep hello
hello                  16384  0
rpi64 helloworld $ rmmod hello
rmmod: ERROR: ../libkmod/libkmod-module.c:793 kmod_module_remove_module() could not
remove 'hello': Operation not permitted
rmmod: ERROR: could not remove module hello: Operation not permitted
rpi64 helloworld $ sudo rmmod hello
rpi64 helloworld $ dmesg |tail
...
[  320.530330] Hello, world
[  343.175478] Goodbye, cruel world
rpi64 helloworld $
```

>>

Who exactly is using a given kernel module?

There is no definitive answer to this question, unfortunately. However, useful tips/tricks can be gleaned from here:
Is there a way to figure out what is using a Linux kernel module? [SO]

--snip--

A Few Other Details

Kernel programming differs from user-space programming in many ways. We'll point things out as we get to them over the course of the book, but there are a few fundamental issues which, while not warranting a section of their own, are worth a mention. So, as you dig into the kernel, the following issues should be kept in mind.

- Applications are laid out in virtual memory with a very large stack area. The stack, of course, is used to hold the function call history and all automatic variables created by currently active functions. The kernel, instead, has a very small stack; it can be as small as a single, 4096-byte page. Your functions must share that stack with the entire kernel-space call chain. Thus, it is never a good idea to declare large automatic variables; if you need larger structures, you should allocate them dynamically at call time.

- Often, as you look at the kernel API, you will encounter function names starting with a double (or more!) underscore (__). Functions so marked are generally a low-level component of the interface and should be used with caution. Essentially, the double underscore says to the programmer: "If you call this function, be sure you know what you are doing."

- Kernel code cannot do floating point arithmetic. Enabling floating point would require that the kernel save and restore the floating point processor's state on each entry to, and exit from, kernel space—at least, on some architectures. Given that there really is no need for floating point in kernel code, the extra overhead is not worthwhile.

<<

FP in the kernel?

```
#include <asm/fpu/api.h>
...

kernel_fpu_begin();
my_cirarea = (22.0/7.0) * r * r;
kernel_fpu_end();

// Do NOT try and printk it! will result in a big WARNing message (via
WARN_ONCE())
#if 0
pr_info("mycalc = %.3f\n", mycalc);
#endif
...
```

>>

Compiling and Loading

The “hello world” example at the beginning of this chapter included a brief demonstration of building a module and loading it into the system. There is, of course, a lot more to that whole process than we have seen so far. This section provides more detail on how a module author turns source code into an executing subsystem within the kernel.

Compiling Modules

As the first step, we need to look a bit at how modules must be built. The build process for modules differs significantly from that used for user-space applications; the kernel is a large, standalone program with detailed and explicit requirements on how its pieces are put together. The build process also differs from how things were done with previous versions of the kernel; the new build system is simpler to use and produces more correct results, but it looks very different from what came before. The kernel build system is a complex beast, and we just look at a tiny piece of it. The files found in the Documentation/kbuild directory in the kernel source are required reading for anybody wanting to understand all that is really going on beneath the surface.

There are some prerequisites that you must get out of the way before you can build kernel modules. The first is to ensure that you have sufficiently current versions of the compiler, module utilities, and other necessary tools. The file Documentation/Changes in the kernel documentation directory always lists the required tool versions; you should consult it before going any further. Trying to build a kernel (and its modules) with the wrong tool versions can lead to no end of subtle, difficult problems. Note that, occasionally, a version of the compiler that is too new can be just as problematic as one that is too old; the kernel source makes a great many assumptions about the compiler, and new releases can sometimes break things for a while.

If you still do not have a kernel tree handy, or have not yet configured and built that kernel, now is the time to go do it. You cannot build loadable modules for a 2.6 kernel without this tree on your filesystem. It is also helpful (though not required) to be actually running the kernel that you are building for.

<< *Note:*
1. It is still possible to build loadable kernel modules even if the kernel source tree is not present in it's entirity and not explicitly built; what *is definitely* required (minimally) are the kernel headers (within the include/ branch).

Also, it is a good idea to at least do a minimal configuration on the kernel prior to building modules (one can copy across a relevant *arch/<architecture>/configs/<some-config-file>* to the root of the kernel source tree as '.config' and do a 'make menuconfig' (or at least a 'make defconfig' which does the configuration using the default answer to all options), saving & exiting.

2. Practically speaking, on Ubuntu (should work on any Debian-based Linux), do:
```
sudo apt-get install linux-headers-generic gcc
```

to get all dependencies in place in order to compile kernel modules.
>>

Once you have everything set up, creating a makefile for your module is straightforward.
In fact, for the “hello world” example shown earlier in this chapter, a single
line will suffice:

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

Readers who are familiar with make, but not with the 2.6 kernel build system, are likely to be wondering how this makefile works. The above line is not how a traditional makefile looks, after all. The answer, of course, is that the kernel build system handles the rest. The assignment above (which takes advantage of the extended syntax provided by GNU make) states that there is one module to be built from the object file *hello.o*. The resulting module is named *hello.ko* after being built from the object file.

<<
Simplest Makefile for a kernel module (LKM):

```
$ cat Makefile
# Simplest LKM Makefile
obj-m = hello.o
KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)

all:
	make -C $(KDIR) M=$(PWD) modules
install:
	make -C $(KDIR) M=$(PWD) modules_install
clean:
	make -C $(KDIR) M=$(PWD) clean
$
```

Build with:
$ make
>>

From the official kernel doc [here](#):

... The command to build an external module is:

```
$ make -C <path_to_kernel_src> M=$PWD
```

The kbuild system knows that an external module is being built due to the "M=<dir>" option given in the command.

...

If, instead, you have a module called my_module.ko that is generated from two source files (called, say, file1.c and file2.c), the correct incantation would be:

```
obj-m := my_module.o
my_module-objs := file1.o file2.o
```

[...]

1. Makefile for LKM – variation 1: simplest

```
# Simplest LKM Makefile
MODNAME = hello

obj-m := $(MODNAME).o

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

all: module

module:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
install:
	make -C $(KDIR) M=$(PWD) modules_install
clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

<<
Tip:

1. *Be wary of copying and pasting (via the clipboard) the above from a PDF document into an editor – strange characters could get introduced causing* make *to complain and fail. It's best to take the trouble to actually type this in!*

2. *Also, you should know that the above indented lines in a Makefile are indented by a TAB character and not whitespace.*

>>

2. Makefile for LKM – variation 2: simple, with check for kernel Makefile parsing (from LDD3)

```
$ cat Makefile
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
	obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
	KERNELDIR	:= /lib/modules/$(shell uname -r)/build
	PWD		:= $(shell pwd)
default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
$
```

Official kernel documentation on kernel Makefile(s) and the kbuild system:
https://www.kernel.org/doc/html/latest/kbuild/makefiles.html .

Useful; f.e., common variables (like $KERNELRELEASE are documented here:
https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#kbuild-variables

3. Makefile for LKM – variation 3: simple, with cross-compile + extra cflags usage example

Alternatively, you could use any of the following templates shown below for a good 2.6 kernel loadable module's Makefile:

```
$ cat Makefile
# This Makefile layout is based on orig src from here:
# http://www.linux.com/news/software/linux-kernel/23685-the-kernel-newbie-corner-your-first-loadable-kernel-module

ifeq ($(KERNELRELEASE),)
  # To support cross-compiling for the ARM:
  # For ARM, invoke make as:
  # make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
  ifeq ($(ARCH),arm)
    # *UPDATE* 'KDIR' below to point to the ARM Linux kernel
    # source tree on your box
    KDIR ?= ~/4.9.1
  else
    KDIR ?= /lib/modules/$(shell uname -r)/build
  endif

PWD := $(shell pwd)

.PHONY: build clean

build:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
# Assuming ‘install’ is called with root access
install:
	$(MAKE) -C $(KDIR) M=$(PWD) modules_install
	depmod
clean:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

else
   ccflags-y += -DDEBUG
   #  EXTRA_CFLAGS += -DDEBUG  (deprecated)
   $(info +++ Building for ARCH=${ARCH} , KERNELRELEASE=${KERNELRELEASE} , ccflags-y+=${ccflags-y})
   obj-m :=    hello.o
endif
$
>>
```

<<
Assignment

Write a ‘hello, world’ kernel module. After you have the basic one working, emit 8 printk’s in the init method, at all logging levels (using the *pr_*()* macros).

If you’re using Linux on an embedded board, test and see which appear on the console device…
>>

4. Makefile for LKM – variation 4: the ‘better’ Makefile

Do learn to use [the ‘better Makefile’]() *for building your kernel modules !*
do read the associated PDF as well…

In-depth documentation on the Linux kernel build system (kbuild) and Makefiles can be [found here]().

NOTE-

Q. What happens if two kernel modules define the same global variable? eg. `int g;`
A. [See this SO link]().

Short answer: the global is valid and unique for each kernel module, IOW, it just works as usual. It's *not* shared between the 2 kernel modules. However, if you mark the global *as exported* with the `EXPORT_SYMBOL` macro, *then* it is truly shared!

It's important to realize that kernel modules have a dependency on the kernel source tree and machine architecture and kernel configuration that they are built against: one can *only* use a kernel module on the architecture (cpu family) and particular kernel version and configuration it has been built for!

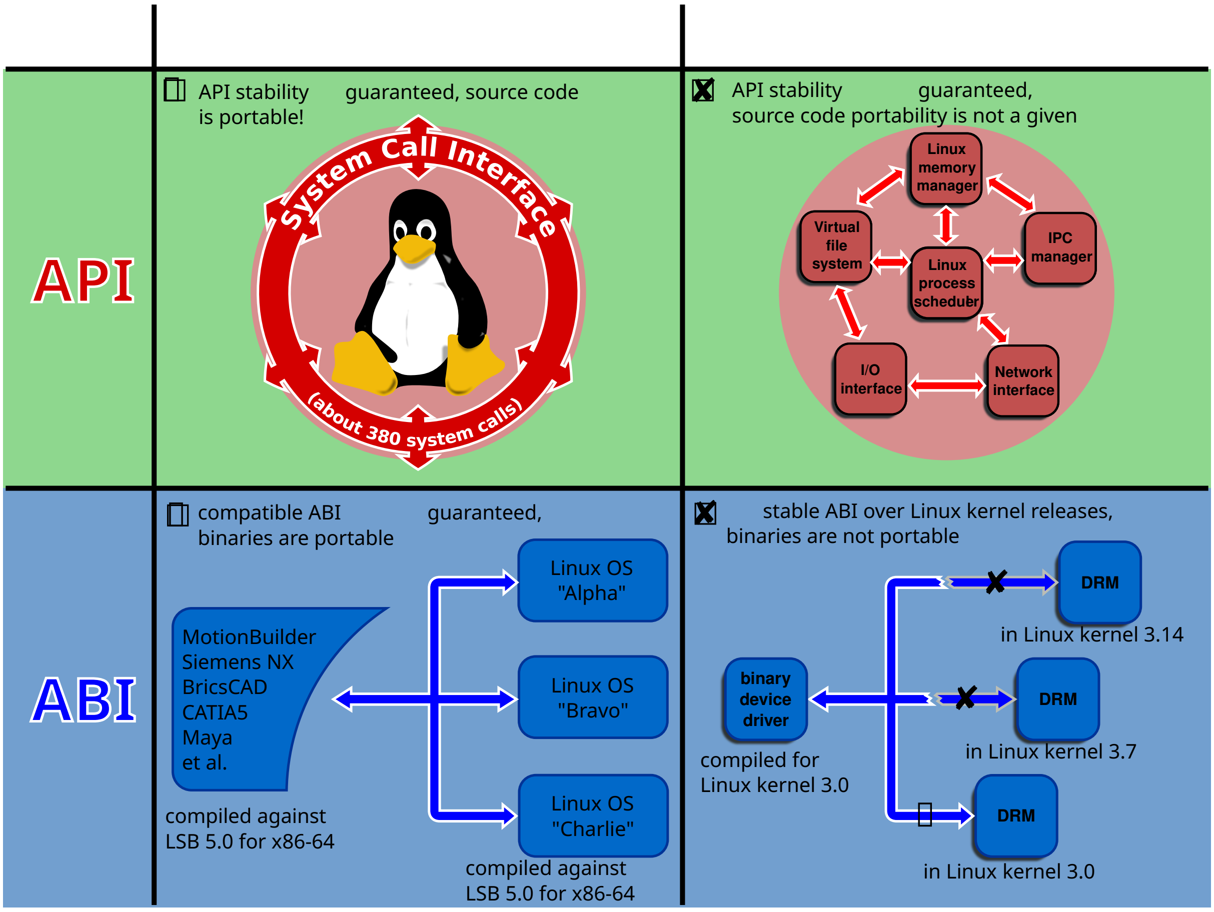
This version information can be seen with the modinfo utility; it's called the “*vermagic*”:

```
$ modinfo ./hello.ko
filename:       /home/kaiwan/src-show-2.6k/LKMs/helloworld/./hello.ko
license:        Dual BSD/GPL
srcversion:     4FB99F668096327D97C09C0
depends:
vermagic:       3.13.0-36-generic SMP mod_unload modversions 686
$
```

<<
[Source]()

“… A module will not be loaded if the "**vermagic**" string contained within the kernel module does not match the value of the currently running kernel. If it is known that the module is compatible with the current running kernel the "vermagic" check can be ignored with `modprobe --force-vermagic`.

Warning: Ignoring the version checks for a kernel module can cause a kernel to crash or a system to exhibit undefined behavior due to incompatibility. Use `--force-vermagic` only with the utmost caution.”
>>



"Linux kernel interfaces" by Shmuel Csaba Otto Traian. Licensed under CC BY-SA 3.0 via Wikimedia Commons -
https://commons.wikimedia.org/wiki/File:Linux_kernel_interfaces.svg#mediaviewer/File:Linux_kernel_interfaces.svg

For example, when attempting to run a VMware guest OS for the first time, the VMware hypervisor application arranges for the guest VM to install some required kernel modules (for guest acceleration/paravirtualization purposes); how? They're downloaded onto the target system in source form (zipped), uncompressed, built and loaded on the target system!

The 'clean' rule

<<

The 'clean' rule should be:

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

A useful point: try doing the 'make' with the verbose option set; this way you can see exactly what it's doing...

```
# make V=1
...
...
#

or

# make -d       # shows how make rules are decided (too detailed, probably)
...
```

>>

Once again, we are seeing the extended GNU make syntax in action. This makefile is read twice on a typical build. When the makefile is invoked from the command line, it notices that the KERNELRELEASE variable has not been set. It locates the kernel source directory by taking advantage of the fact that the symbolic link build in the installed modules directory points back at the kernel build tree. If you are not actually running the kernel that you are building for, you can supply a KERNELDIR= option on the command line, set the KERNELDIR environment variable, or rewrite the line that sets KERNELDIR in the makefile. Once the kernel source tree has been found, the makefile invokes the default: target, which runs a second make command (parameterized in the makefile as $(MAKE)) to invoke the kernel build system as described previously. On the second reading, the makefile sets obj-m , and the kernel makefiles take care of actually building the module.

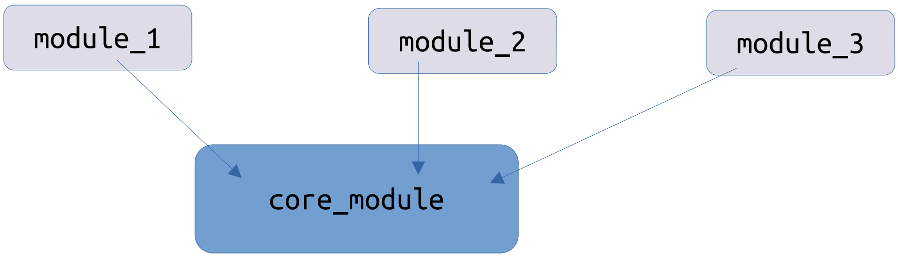
This mechanism for building modules may strike you as a bit unwieldy and obscure. Once you get used to it, however, you will likely appreciate the capabilities that have been programmed into the kernel build system. Do note that the above is not a complete makefile; a real makefile includes the usual sort of targets for cleaning up unneeded files, installing modules, etc. See the makefiles in the example source directory for a complete example.

Resource-

See the excellent article (series) by Robert PJ Day on [The Kernel Newbie Corner: Your First Loadable Kernel Module.]

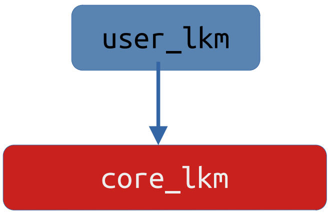
Module Stacking

Using a module as a “library” of sorts… a general picture:



Eg. (from my LKP book: code):

```
This kernel module - core_lkm - is part of the 'modstacking' POC project:
     user_lkm
        |
     core_lkm           [<--- this code]
The user_lkm kernel module calls an (exported) function that resides in the
core_lkm kernel module.
```



Implications: the *core_lkm* (the ‘library’) module:

- must be loaded first
- it’s usage count will get incremented
    - thus, it can’t be removed before first removing the module(s) stacked over it
- you must explicitly export (via `EXPORT_SYMBOL()`) the data / functions you want exposed to module stacked above it, and, in fact, to the rest of the kernel.

- (Provided they’re within the same folder; do ensure that) the Makefile now includes both modules being built:

```
PWD             := $(shell pwd)
obj-m           := core_lkm.o
obj-m           += user_lkm.o
ccflags-y       += -DDEBUG
```

- After insertion, the *lsmod* output reveals the ‘stacking’ clearly: the fourth column – of the line containing the ‘core’ (‘library’) module - is populated with the module(s) that depend upon it, that use its code/data:

```
Module                  Size  Used by
user_lkm               20480  0
core_lkm               16384  1 user_lkm
```

Using modprobe to load modules with dependencies taken into account

1. First install the modules:
`sudo make install`

(gets installed under */lib/modules/<uname -r>/extra*)

2. Perform the modprobe:
`sudo modprobe user_lkm`

It first loads the *core_lkm* module (from /lib/modules/<...>/extra/); how does it know? The dependency is generated:

```
$ modinfo /lib/modules/6.1.25-lkp-dbg/extra/user_lkm.ko.zst
filename:       /lib/modules/6.1.25-lkp-dbg/extra/user_lkm.ko.zst
license:        Dual MIT/GPL
srcversion:     12444982DE3664BD748E403
depends:        core_lkm
retpoline:      Y
name:           user_lkm
vermagic:       6.1.25-lkp-dbg SMP preempt mod_unload modversions
...
```

Easier way: simply link together multiple source files into a single kernel module. Some of the code can be common stuff – a 'library' of sorts...

An extract from the LKP book:
"… So, let's summarize: for emulating a library-like feature within the kernel module space, we explored two techniques:

- The first technique we used works by linking multiple source files together into a single kernel module.
- This is as opposed to module stacking technique, where we actually build multiple kernel modules and "stack" them on top of each other.

Not only does the first technique work well, it also has these advantages:

- We do not have to explicitly mark (via `EXPORT_SYMBOL()`) every data/function symbol that we use as exported.
- The functions are only available to the kernel module to which it is actually linked to (and not the entire kernel, including other modules).

  This is a good thing! All this at the cost of slightly tweaking the Makefile – well worth it."

Module Parameters

An example of using module parameters can be seen with the Intel e1000e network driver:

```
$ modinfo -p
/lib/modules/5.15.0-56-generic/kernel/drivers/net/ethernet/intel/e1000e/
e1000e.ko
debug:Debug level (0=none,...,16=all) (int)
copybreak:Maximum size of packet that is copied to a new buffer on receive
(uint)
TxIntDelay:Transmit Interrupt Delay (array of int)
TxAbsIntDelay:Transmit Absolute Interrupt Delay (array of int)
RxIntDelay:Receive Interrupt Delay (array of int)
RxAbsIntDelay:Receive Absolute Interrupt Delay (array of int)
InterruptThrottleRate:Interrupt Throttling Rate (array of int)
IntMode:Interrupt Mode (array of int)
SmartPowerDownEnable:Enable PHY smart power down (array of int)
KumeranLockLoss:Enable Kumeran lock loss workaround (array of int)
WriteProtectNVM:Write-protect NVM [WARNING: disabling this can lead to corrupted
NVM] (array of int)
CrcStripping:Enable CRC Stripping, disable if your BMC needs the CRC (array of
int)
$
```

Kernel modules cannot be passed parameters (arguments) the "traditional" way: there's no main() receiving argc, argv, right!

So, passing parameters to a kernel module is a bit of a trick:

- Declare a global (static) variable in the kernel module.
- Use it as a module parameter, by wrapping it in a *module_param* macro.
- Pass it to the kernel module as a <*name*>=<*value*> style list to the insmod command. The parameter (in reality, the global variable within the module), will be set to the value passed!

Trivial Example:
(all code is not shown)

```
...
static int myparam=1;
module_param(myparam, int, 0);
MODULE_PARM_DESC(myparam, "Set to the value of gold currency today (default=1)");

static u32 biggy=0x100;
module_param(biggy, uint, 0);
MODULE_PARM_DESC(biggy, "Set to the value of The Big One (default=0x100)");

static char *regname;
module_param(regname, charp, 0);
MODULE_PARM_DESC(regname, "Set to a string describing whatever...");

...
```

The module_param macro:

The first parameter to the module_param() macro is the name of the (global) parameter itself.

The second parameter is the data type.
Standard types are:
 byte, short, ushort, int, uint, long, ulong
 charp: a character pointer
 bool: a bool, values 0/1, y/n, Y/N.
 invbool: the above, only sense-reversed (N = true).

The third parameter to the module_param() macro is called “perm”.
It's to do with access to the module parameter via sysfs.
If 'perm' is 0 if the the variable is not to appear in sysfs, or 0444 for world-readable, 0644 for root-writable, etc. Note that if it is writable, you may need to use kparam_block_sysfs_write() around accesses (esp. charp, which can be kfreed when it changes).

NOTE:-
For the module parameter sysfs permission: you Cannot specify any ‘execute’ permission bit, nor a ‘write’ bit for ‘others/public’; they will trigger a compiler error.

Usage

After the kernel module (lets just call it 'mydemo') is built, on the command-line, the end-user can look up what params can be passed and additional information, using the 'modinfo' utility:

```
# modinfo ./mydemo.ko
filename:       ./mydemo.ko
license:        GPL
description:    Whatever description you choose to give...
author:         Author(s) name(s), email
srcversion:     AECC27430B585F832DDD3CF
depends:
vermagic:       2.6.38-11-generic SMP mod_unload modversions
parm:           myparam:Set to the value of gold currency today (default=1) (int)
parm:           biggy:Set to the value of The Big One (default=0x100) (uint)
parm:           regname:Set to a string describing whatever... (charp)
# insmod ./mydemo.ko myparam=3 regname="myregs"
#
```

Notice, in the above example, the module parameter 'biggy' was not passed; hence, it defaults to it's 0x100 value.

Using a more human-friendly name, rather than the original variable name for the parameter

Format:
```
module_param_named(human_friendly_name, original_var_name, <data_type>, <mode>);
```

Eg.
drivers/md/dm-bufio.c
```
...
module_param_named(max_cache_size_bytes, dm_bufio_cache_size, ulong, S_IRUGO |
S_IWUSR);
```

```
MODULE_PARM_DESC(max_cache_size_bytes, "Size of metadata cache");

module_param_named(max_age_seconds, dm_bufio_max_age, uint, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(max_age_seconds, "Max age of a buffer in seconds");
...
```

Nowadays:

- all necessary modules loading is handled automatically by **udev**
- udev is itself a part of systemd! (see systemd-udevd.service(8) for details)

Source: Kernel Modules, Arch Linux Wiki

Automatic module handling

Today, all necessary modules loading is handled automatically by **udev**, so if you do not need to use any out-of-tree kernel modules, there is no need to put modules that should be loaded at boot in any configuration file. However, there are cases where you might want to load an extra module during the boot process, or blacklist another one for your computer to function properly.

Kernel modules can be explicitly loaded during boot and are configured as a static list in files under `/etc/modules-load.d/`. Each configuration file is named in the style of `/etc/modules-load.d/<program>.conf`. Configuration files simply contain a list of kernel modules names to load, separated by newlines. Empty lines and lines whose first non-whitespace character is `#` or `;` are ignored.

```
/etc/modules-load.d/virtio-net.conf
```

```
# Load virtio-net.ko at boot
virtio-net
```

See **modules-load.d(5)** for more details.

...

Using files in /etc/modprobe.d/

Files in `/etc/modprobe.d/` directory can be used to pass module settings to **udev**, which will use `modprobe` to manage the loading of the modules during system boot. Configuration files in this directory can have any name, given that they end with the `.conf` extension. The syntax is:

```
/etc/modprobe.d/myfilename.conf
```

```
options module_name parameter_name=parameter_value
```

For example:

```
/etc/modprobe.d/thinkfan.conf
```

```
# On ThinkPads, this lets the 'thinkfan' daemon control fan speed
options thinkpad_acpi fan_control=1
```

Note: If any of the affected modules is loaded from the initramfs, then you will need to add the appropriate `.conf` file to `FILES` in **mkinitcpio.conf** or use the `modconf` **hook**, so that it will be included in the initramfs. To see the contents of the default initramfs use `lsinitcpio /boot/initramfs-linux.img` .

Using kernel command line
Iff the module is built into the kernel, you can also pass options to the module using the kernel command line. For all common bootloaders, the following syntax is correct:

```
module_name.parameter_name=parameter_value
```

For example:

```
thinkpad_acpi.fan_control=1
```

Simply add this to your bootloader's kernel-line, as described in **Kernel Parameters**.

...

Blacklisting

Blacklisting, in the context of kernel modules, is a mechanism to prevent the kernel module from loading. This could be useful if, for example, the associated hardware is not needed, or if loading that module causes problems: for instance there may be two kernel modules that try to control the same piece of hardware, and loading them together would result in a conflict.

…

Create a `.conf` file inside `/etc/modprobe.d/` and append a line for each module you want to blacklist, using the `blacklist` keyword. If for example you want to prevent the `pcspkr` module from loading:

```
/etc/modprobe.d/nobeep.conf
```

```
# Do not load the 'pcspkr' module on boot.
blacklist pcspkr
```

…

Using kernel command line

Tip: This can be very useful if a broken module makes it impossible to boot your system.

You can also blacklist modules from the bootloader.

Simply add `modprobe.blacklist=modname1,modname2,modname3` to your bootloader's kernel line, as described in **Kernel parameters**.

Note: When you are blacklisting more than one module, note that they are separated by commas only. Spaces or anything else might presumably break the syntax.

...

Modules do not load

In case a specific module does not load and the boot log (accessible with `journalctl -b`) says that the module is blacklisted, but the directory `/etc/modprobe.d/` does not show a corresponding entry, check another modprobe source folder at `/usr/lib/modprobe.d/` for blacklisting entries.

A module will not be loaded if the "vermagic" string contained within the kernel module does not match the value of the currently running kernel. If it is known that the module is compatible with the current running kernel the "vermagic" check can be ignored with `modprobe --force-vermagic`.

Warning: Ignoring the version checks for a kernel module can cause a kernel to crash or a system to exhibit undefined behavior due to incompatibility. Use `--force-vermagic` only with the utmost caution.

…

Ref / Useful

Clean module disabling, Kess Cook, Nov 2012

https://wiki.archlinux.org/index.php/udev

https://unix.stackexchange.com/questions/330186/where-does-modprobe-load-a-driver-that-udev-requests

---

Recent:

- [3.18] : Module parameters can be defined with a new "unsafe" flag; any attempt to modify such a parameter will generate a warning and taint the kernel.
  The *module_param_unsafe()* macro can be used to set up such parameters.
- Kernel modules can now be installed in compressed form by the build system.
- [3.7] Cryptographically-signed kernel modules

This release allows to optionally sign kernel modules. The kernel can optionally disable completely the load of modules that have not been signed with the correct key - even for root users. This feature is useful for security purposes, as an attacker who gains root user access will not be able to install a rootkit using the module loading routines.

Recommended LWN article: Loading signed kernel modules Code: (commit 1, 2, 3)

<<

Kernel PGP signing: https://www.kernel.org/signature.html

Kernel module signing:

https://www.kernel.org/doc/html/latest/admin-guide/module-signing.html if ! CONFIG_MODULE_SIG_FORCE

then kernel tainted with the 'E' flag

>>

Important: Signed modules must *never* be stripped (even of debug symbols); if stripped, the signature isn't parsed correctly and laoding can fail.

Security

Alexander Popov: **kconfig-hardened-check** script ! V useful: can generate a recommended config for security… draws on inputs from KSPP, Clip OS, grsec, kernel maintainers.

Install:
```
pip install git+https://github.com/a13xp0p0v/kconfig-hardened-check
```

Help screen:

```
$ kconfig-hardened-check
usage: kconfig-hardened-check [-h] [--version] [-m {verbose,json,show_ok,show_fail}] [-
c CONFIG] [-l CMDLINE]
                              [-p {X86_64,X86_32,ARM64,ARM}] [-g
{X86_64,X86_32,ARM64,ARM}]

A tool for checking the security hardening options of the Linux kernel

options:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -m {verbose,json,show_ok,show_fail}, --mode {verbose,json,show_ok,show_fail}
                        choose the report mode
  -c CONFIG, --config CONFIG
                        check the security hardening options in the kernel Kconfig file
(also supports *.gz files)
  -l CMDLINE, --cmdline CMDLINE
                        check the security hardening options in the kernel cmdline file
  -p {X86_64,X86_32,ARM64,ARM}, --print {X86_64,X86_32,ARM64,ARM}
                        print the security hardening recommendations for the selected
microarchitecture
  -g {X86_64,X86_32,ARM64,ARM}, --generate {X86_64,X86_32,ARM64,ARM}
                        generate a Kconfig fragment with the security hardening options
for the selected microarchitecture
~ $
```

So, I generated a 'regular' kconfig file : myconfig.1 (for arm64); then, I used kconfig-hardened-check to generate the recommended security conscious kconfig:

```
kconfig-hardened-check -g ARM64 > kconfig_arm64_hardened.config
```

Then **merge** the two!

```
$ ~/6.1.25/scripts/kconfig/merge_config.sh
Usage: /home/c2kp/6.1.25/scripts/kconfig/merge_config.sh [OPTIONS] [CONFIG [...]]
  -h    display this help text
```

```
  -m    only merge the fragments, do not execute the make command
  -n    use allnoconfig instead of alldefconfig
  -r    list redundant entries when merging fragments
  -y    make builtin have precedence over modules
  -O    dir to put generated output files.  Consider setting $KCONFIG_CONFIG instead.
  -s    strict mode. Fail if the fragment redefines any value.

Used prefix: 'CONFIG_'. You can redefine it with $CONFIG_ environment variable.
$

$ scripts/kconfig/merge_config.sh -m .config kconfig_arm64_hardened.config
…
…
```

Wrote the merged result into .config .

Now compare them! *Just two example kernel configs compared – show like this:*

first, the kconfig-hardened-check recommendation
next, the new merged .config (should follow the recommendation)
finally, the original myconfig kernel config file (typically insecure config)

Lines spaced for clarity...

```
$ grep MODULES kconfig_arm64_hardened.print ~/rpi_work/kernel_rpi/linux/.config
~/rpi_work/kernel_rpi/linux/myconfig.1

kconfig_arm64_hardened.print:CONFIG_MODULES                          |kconfig| is not
set |    kspp    |cut_attack_surface

/home/c2kp/rpi_work/kernel_rpi/linux/.config:CONFIG_MODULES_USE_ELF_RELA=y
/home/c2kp/rpi_work/kernel_rpi/linux/.config:# CONFIG_MODULES is not set

/home/c2kp/rpi_work/kernel_rpi/linux/myconfig.1:CONFIG_MODULES_USE_ELF_RELA=y
/home/c2kp/rpi_work/kernel_rpi/linux/myconfig.1:CONFIG_MODULES=y   << original had it
set >>
/home/c2kp/rpi_work/kernel_rpi/linux/myconfig.1:CONFIG_MODULES_TREE_LOOKUP=y

...
```

With the new security-conscious kernel config, see this:
'Only' 24 configs fail the check, 161 pass (earlier it was 91 Ok, 94 failing):

Now, you should try to follow the above recommendations as much as is possible, further hardening the kernel!
Once built and deployed, ensure you run your entire test suite…
