Driving Me Nuts - Things You Never Should Do in the Kernel

Apr 06, 2005 By Greg Kroah-Hartman (/user/800887)
in

(/issue/133)

From Issue #133
May 2005 (/issue/133)

How do you read and write files from a kernel module? Wait, make that "how would you read and write files from a kernel module if that weren't a bad thing to do?"

I Want to Do This Anyway

Now that you understand the reasoning behind forbidding the ability to read a file from a kernel module, you of course can skip the rest of this article. It does not concern you, as you are off busily converting your kernel module to use sysfs.

Still here? Okay, so you still want to know how to read a file from a kernel module, and no amount of persuading can convince you otherwise. You promise never to try to do this in code that will be submitted for inclusion into the main kernel tree and that I never described how to do this, right?

Actually, reading a file is quite simple, once one minor issue is resolved. A number of the kernel system calls are exported for module use; these system calls start with sys_. So, for the read system call, the function sys_read should be used.

The common approach to reading a file is to try code that looks like the following:

```
fd = sys_open(filename, O_RDONLY, 0);
if (fd >= 0) {
  /* read the file here */
  sys_close(fd);
}
```

However, when this is tried within a kernel module, the sys_open() call usually returns the error -EFAULT. This causes the author to post the question to a mailing list, which elicits the "don't read a file from the kernel" response described above.

The main thing the author forgot to take into consideration is the kernel expects the pointer passed to the sys_open() function call to be coming from user space. So, it makes a check of the pointer to verify it is in the proper address space in order to try to convert it to a kernel pointer that the rest of the kernel can use. So, when we are trying to pass a kernel pointer to the function, the error -EFAULT occurs.

Fixing the Address Space

To handle this address space mismatch, use the functions get_fs() and set_fs(). These functions modify the current process address limits to whatever the caller wants. In the case of sys_open(), we want to tell the kernel that pointers from within the kernel address space are safe, so we call:

```
set_fs(KERNEL_DS);
```

The only two valid options for the set_fs() function are KERNEL_DS and USER_DS, roughly standing for kernel data segment and user data segment, respectively.

To determine what the current address limits are before modifying them, call the get_fs() function. Then, when the kernel module is done abusing the kernel API, it can restore the proper address limits.

So, with this knowledge, the proper way to write the above code snippet is:

```
old_fs = get_fs();
set_fs(KERNEL_DS);

fd = sys_open(filename, O_RDONLY, 0);
if (fd >= 0) {
  /* read the file here */
  sys_close(fd);
}
set_fs(old_fs);
```

An example of an entire module that reads the file /etc/shadow and dumps it out to the kernel system log, proving that this can be a dangerous thing to do, can be seen below:

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/fcntl.h>
#include <asm/uaccess.h>
```

```
static void read_file(char *filename)
{
  int fd;
  char buf[1];

  mm_segment_t old_fs = get_fs();
  set_fs(KERNEL_DS);

  fd = sys_open(filename, O_RDONLY, 0);
  if (fd >= 0) {
    printk(KERN_DEBUG);
    while (sys_read(fd, buf, 1) == 1)
      printk("%c", buf[0]);
    printk("\n");
    sys_close(fd);
  }
  set_fs(old_fs);
}

static int __init init(void)
{
  read_file("/etc/shadow");
  return 0;
}

static void __exit exit(void)
{ }

MODULE_LICENSE("GPL");
module_init(init);
module_exit(exit);
```

But What about Writing?

Now, armed with this newfound knowledge of how to abuse the kernel system call API and annoy a kernel programmer at the drop of a hat, you really can push your luck and write to a file from within the kernel. Fire up your favorite editor, and pound out something like the following:

```
old_fs = get_fs();
set_fs(KERNEL_DS);

fd = sys_open(filename, O_WRONLY|O_CREAT, 0644);
```

```
if (fd >= 0) {
  sys_write(data, strlen(data);
  sys_close(fd);
}
set_fs(old_fs);
```

The code seems to build properly, with no compile time warnings, but when you try to load the module, you get this odd error:

```
insmod: error inserting 'evil.ko': -1 Unknown symbol in module
```

This means that a symbol your module is trying to use has not been exported and is not available in the kernel. By looking at the kernel log, you can determine what symbol that is:

```
evil: Unknown symbol sys_write
```

So, even though the function sys_write is present in the syscalls.h header file, it is not exported for use in a kernel module. Actually, on three different platforms this symbol is exported, but who really uses a parisc architecture anyway? To work around this, we need to take advantage of the kernel functions that are available to kernel modules. By reading the code of how the sys_write function is implemented, the lack of the exported symbol can be thwarted. The following kernel module shows how this can be done by not using the sys_write call:

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/file.h>
#include <linux/fs.h>
#include <linux/fcntl.h>
#include <asm/uaccess.h>

static void write_file(char *filename, char *data)
{
  struct file *file;
  loff_t pos = 0;
  int fd;

  mm_segment_t old_fs = get_fs();
  set_fs(KERNEL_DS);

  fd = sys_open(filename, O_WRONLY|O_CREAT, 0644);
  if (fd >= 0) {
    sys_write(fd, data, strlen(data));
    file = fget(fd);
    if (file) {
      vfs_write(file, data, strlen(data), &pos);
      fput(file);
    }
    sys_close(fd);
  }
  set_fs(old_fs);
}

static int __init init(void)
{
  write_file("/tmp/test", "Evil file.\n");
  return 0;
}

static void __exit exit(void)
{ }

MODULE_LICENSE("GPL");
module_init(init);
module_exit(exit);
```

As you can see, by using the functions fget, fput and vfs_write, we can implement our own sys_write functionality.

Comments

Comment viewing options

Threaded list - expanded | Date - newest first | 50 comments per page | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

I think there is a typo on the write code... (/article/8110#comment-351786)

Submitted by Anonymous on May 12, 2010.

I think there's a typo on the final write code. The first call after checking for the file descriptor should be vfs_write and no the sys_write call.

```
  fd = sys_open(filename, O_WRONLY|O_CREAT, 0644);

  if (fd >= 0) {

    vfs_write(fd, data, strlen(data));
```

about usb drivers (/article/8110#comment-349878)

Submitted by Anonymous on Mar 24, 2010.

I am writing usb device driver and i want some help.
OS that i am using is suse10.2. If i write down complete usb driver that is mentioned in your documents, what are the extra things that i need to do apart from inserting that module in to kernel (insmod).If i insert my usb driver module in to kernel how can we stop kernel from using previous usb driver and now kernel should use the driver(module) that i have inserted.
Similarly i have also written mouse device driver for ps2 mouse.But i faced the simiar problem.How we should tell to kernel that dont use previous driver for mouse use that i have inserted.
Please Help Me.If i could get way for this i will be able to run both the drivers.

Yeah, the kernel has (/article/8110#comment-343026)

Submitted by Anonymous (not verified) on Sep 17, 2009.

Yeah, the kernel has internal API to open files: filp_open(), filp_close(), vfs_read(), ...
Consult for example with sound/sound_firmware.c.

In proprietary spec I have (/article/8110#comment-321044)

Submitted by Anonymous (not verified) on Apr 18, 2008.

In proprietary spec I have

int ioctl(int fd, int req=DMX_SET_SOURCE, int *frontend_fd);
where frontend_fd is the pointer to file descriptor of another driver opened previousely.

The logic here - the user is responsible for which driver to connect for internal communication

Anyway I need sys_ioctl to call one driver from another. I can not parse user file descriptor to determine which driver I need.

Don't use sys_*() (/article/8110#comment-313407)

Submitted by Anonymous (not verified) on Feb 03, 2008.

If you do read/write files from the kernel, then please do it without too much hacks, meaning you use `filp_open`, `vfs_read` and `vfs_write` instead of `sys_open`, `sys_read` and `sys_write`, respectively. Thanks!

Hey you said that you can (/article/8110#comment-336626)
Submitted by Saurabh Chokshi (not verified) on May 01, 2009.

Hey you said that you can use filp_open , vfs_read and vfs_write. I tried that but I could not write and read into the file.

```
static void write_file(char *filename, char *data)
{
struct file *file;
file = filp_open(filename, O_WRONLY|O_CREAT, 0777);
ssize_t wc;
wc = vfs_write(file, data, strlen(data),0);
if(wc < strlen(data))
{
printk(KERN_INFO "Problem in Writing Data\n");
}
filp_close(file,0);

}
```

This is my code.

Please let me know where I am wrong.

Thanks,

Translation (/article/8110#comment-334414)
Submitted by Anonymous (not verified) on Mar 11, 2009.

Wow... Can someone translate this into English for me?

How very clever (/article/8110#comment-280118)
Submitted by C. J. Stiller (http://www.linuxjournal.com) (not verified) on Sep 03, 2007.

Greg: thanks for this summary. And how clever of you to obscure your advice concerning file writes from kernel space by including the sys_write() call in your sample, misdirecting readers into ignoring the vfs_write() and thinking that they would still need that syscall.

---

err (/article/8110#comment-280109)
Submitted by Anonymous (not verified) on Sep 03, 2007.

Your final example still uses sys_write and the snippet above that has a syntax error in it.

Kernel (/article/8110#comment-280107)
Submitted by Vanessa Kay (http://www.vanessakay.org) (not verified) on Sep 03, 2007.

God, I hate re-writing kernels. I used to use suse linux. Everytime I had to get my wireless card to work, I would have to re-write the kernel and that shit takes forever.

Rides and Whips (http://www.ridesnwhips.com/)

Clearly, you know a lot (/article/8110#comment-280112)
Submitted by Anonymous (not verified) on Sep 03, 2007.

Clearly, you know a lot about linux. You don't re-write a kernel. You recompile it with drivers or code for your device. If you were rewriting a kernel that shit would take forever. Recompiling a kernel takes me only a few minutes on my machine, granted, it's faster than most. Keep anti-linux comments on topic and try to back them up with fact. The simple fact is, you don't need to make any thing up to criticize any operating system, there is plenty wrong with any given one to be criticized.

You had to re-write the (/article/8110#comment-280110)
Submitted by Anonymous (not verified) on Sep 03, 2007.

You had to re-write the kernel everything you wanted your wifi card to work?

vince

everytime* (/article/8110#comment-280111)
Submitted by Anonymous (not verified) on Sep 03, 2007.

everytime*

Not using sys_write (/article/8110#comment-280086)
Submitted by Anonymous (not verified) on Sep 03, 2007.

"how this can be done by not using the sys_write call"

But you've used it right there!

Stacked Drivers Concept (/article/8110#comment-244319)
Submitted by Ilya. I (not verified) on May 16, 2007.

I know one case when this is the cleanest practical solution.

Linux kernel is supposed to be build on "stacked drivers" concept. In reality, kernel supports it in very limited number of specially designed core drivers. Try to write a translation driver for a device which already has a generic driver. Good example is having a USB-to-serial chip talking to a custom hardware. Now, anyone trying to create a driver for that custom hardware is facing a huge uphill battle with the kernel, unless you can just open device file created by generic driver and talk to it in vfs. However, vfs_read/vfs_write are user-space restricted. (Unacceptable alternatives: user-space daemon, full-blown rewrite of USB-to-serial driver, etc.)

Using technique in this article proves to be the only way for given situation.

You've failed to explain why (/article/8110#comment-280087)

Submitted by Anonymous (not verified) on Sep 03, 2007.

You've failed to explain why you can't solve the problem in userspace in your example.

What's wrong with a libmycustomhardware.so?

Problem with sys_write and sys_open (/article/8110#comment-236568)
Submitted by Sarang (not verified) on Apr 25, 2007.

hi,
information given in this article is very useful.
i have tried above codes on kernel 2.6.9, but getting error with sys_open and sys_write.
it gives error in system log file:
fileops: Unknown symbol sys_open
fileops: Unknown symbol sys_write
Please suggest me how can i overcome this problem.
I welcome your suggestions.
Thank you

dont use sys_read or (/article/8110#comment-344115)
Submitted by Anonymous (not verified) on Oct 15, 2009.

dont use sys_read or sys_write or sys_open or sys_open calls.. instead u can use vfs_read, vfs_write, filp_open, filp_close calls..

don't do that! (/article/8110#comment-280113)
Submitted by Anonymous (not verified) on Sep 03, 2007.

don't do that!

(sorry)

I think using sys_open and (/article/8110#comment-349522)
Submitted by Anonymous on Mar 13, 2010.

I think using sys_open and sys_read is more general than vfs_open vfs_read.
In my case, I need to create to module to read a file from SD card (fat),
vfs_read always give me many errors.

In thread (/article/8110#comment-222839)
Submitted by iPAS (not verified) on Feb 13, 2007.

Could this function be called by Thread ?

Used this to interface a stack with the DVBS drivers of Linux (/article/8110#comment-195660)
Submitted by GeorgeJ (not verified) on Nov 17, 2006.

Used this to interface a stack with the DVBS drivers of Linux.
Good Job!!
Hope to see you in person some day!!!

i guess filpopen should work (/article/8110#comment-311786)
Submitted by Anonymous (not verified) on Jan 31, 2008.

i guess filpopen should work ...

Did u look into copyfromuser() and copytouser()

These are useful for getting data back and forth between kernel and user space.

netlink sockets are the very (/article/8110#comment-344116)
Submitted by Anonymous (not verified) on Oct 15, 2009.

netlink sockets are the very efficient way to communicate between user, kernel modules and vice-versa.. Its very easy to use also..

Post new comment

Please note that comments may not appear immediately, so there is no need to repost your comment.

Your name:
Anonymous

E-mail:

The content of this field is kept private and will not be shown publicly.

Homepage:

Subject:

Comment: *

Allowed HTML tags: <a> <em> <strong> <cite> <code> <pre><tt> <ul> <ol> <li> <dl> <dt> <dd> <i> <b><blockquote>
Lines and paragraphs break automatically.
Use to create page breaks.

More information about formatting options (/filter/tips)

☑ Notify me when new comments are posted

[✓ All comments] Replies to my comment

Preview