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

Linux Journal
by Greg Kroah-Hartman on April 6, 2005

On Linux kernel programming mailing lists oriented toward new developers (see the on-line Resources), a number of common questions are asked. Almost every time one of these questions is asked, the response always is, “Don't do that!”, leaving the bewildered questioner wondering what kind of strange development community they have stumbled into. This is the first in an occasional series of articles that attempts to explain why it generally is not a good idea to do these kinds of things. Then, in order to make up for the chastising, we break all of the rules and show you exactly how to do them anyway.

Read a File

The most common question asked in this don't-do-that category is, “How do I read a file from within my kernel module?” Most new kernel developers are coming from user-space programming environments or other operating systems where reading a file is a natural and essential part of bringing configuration information into a program. From within the Linux kernel, however, reading data out of a file for configuration information is considered to be forbidden. This is due to a vast array of different problems that could result if a developer tries to do this.

The most common problem is interpreting the data. Writing a file interpreter from within the kernel is a process ripe for problems, and any errors in that interpreter can cause devastating crashes. Also, any errors in the interpreter could cause buffer overflows. These might allow unprivileged users to take over a machine or get access to protected data, such as password files.

Trying to protect the kernel from dumb programming errors is not the most important reason for not allowing drivers to read files. The biggest issue is policy. Linux kernel programmers try to flee from the word policy as fast as they can. They almost never want to force the kernel to force a policy on to user space that can possibly be avoided. Having a module read a file from a filesystem at a specific location forces the policy of the location of that file to be set. If a Linux distributor decides the easiest way to handle all configuration files for the system is to place them in the /var/black/hole/of/configs, this kernel module has to be modified to support this change. This is unacceptable to the Linux kernel community.

Another big issue with trying to read a file from within the kernel is trying to figure out exactly where the file is. Linux supports filesystem namespaces, which allow every process to contain its own view of the filesystem. This allows some programs to see only portions of the entire filesystem, while others see the filesystem in different locations. This is a powerful feature, and trying to determine that your module lives in the proper filesystem namespace is an impossible task.

If these big issues are not enough, the final problem of how to get the configuration into the kernel is also a policy decision. By forcing the kernel module to read a file every time, the author is forcing that decision. However, some distributions might decide it is better to store system configurations in a local database and have helper programs funnel that data into the kernel at the proper time. Or, they might want to connect to an external machine in some manner to determine the proper configuration at that moment. Whatever method the user decides to employ to store configuration data, by forcing it to be in a specific file, he or she is forcing that policy decision on the user, which is a bad idea.

But How Do I Configure Things?

After finally understanding the Linux kernel programmer's aversion to policy decisions and thinking that those idealists are out of their mind, you still are left with the real problem of how to get configuration data into a kernel module. How can this be done without incurring the wrath of an angry e-mail flame war?

A common way of sending data to a specific kernel module is to use a char device and the ioctl system call. This allows the author to send almost any kind of data to the kernel, with the user-space program sending the data at the proper time in the initialization process. The ioctl command, however, has been determined to have a lot of nasty side affects, and creating new ioctls in the kernel generally is frowned on. Also, trying properly to handle a 32-bit user-space program making an ioctl call into a 64-bit kernel and converting all of the data types in the correct manner is a horrible task to undertake.

Because ioctls are not allowed, the /proc filesystem can be used to get configuration data into the kernel. By writing data to a file in the filesystem created by the kernel module, the kernel module has direct access to it. Recently, though, the proc filesystem has been clamped down on by the kernel developers, as it was horribly abused by programmers over time to contain almost any type of data. Slowly this filesystem is being cleaned up to contain

only process information, such as the names of filesystem states.

For a more structured filesystem, the sysfs filesystem provides a way for any device and any driver to create files to which configuration data may be sent. This interface is preferred over ioctls and using /proc. See previous articles in this column for how to create and use sysfs files within a kernel module.

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.

I Never Told You about This

In conclusion, reading and writing a file from within the kernel is a bad, bad thing to do. Never do it. Ever. Both modules from this article, along with a Makefile for compiling them, are available from the *Linux Journal* FTP site, but we expect to see no downloads in the logs. And, I never told you how to do it either. You picked it up from someone else, who learned it from his sister's best friend, who heard about how to do it from her coworker.

Resources for this article: /article/8130.

Greg Kroah-Hartman is one of the authors of *Linux Device Drivers, 3rd edition* and is the kernel maintainer for more driver subsystems than he likes to admit. He works for SuSE Labs, doing various kernel-specific things and can be reached at greg@kroah.com for issues unrelated to this article.