CS411 Operating Systems II Fall 2008
Project #5
Due 6:00 am, Saturday, 6 December 2008

Objectives:

1. Write a device driver for the Linux kernel
2. Understand the Linux block device interface
3. Be able to use the Linux kernel's Crypto API
4. Practice kernel coding skills

Description:

A device driver is a piece of code that enables the operating system to "talk" to a particular piece of hardware. Device drivers account for more than half of the code in the Linux kernel, which is a good thing, because, as Greg Kroah-Hartman boasts in all of his talks, "the Linux Kernel supports more devices than any other operating system in history." One of the most common ways for a developer to get his code into the Linux kernel is by writing a driver for a currently unsupported device. This is one also of the most common first tasks for those hired by a company to do Linux kernel development. In this project, we will write a device driver for the Linux kernel which allocates a chunk of memory and presents it as a block device, which is a device that allows random access to data in fixed-size blocks. The most familiar example of a block device is a disk drive.

As an added twist, your device driver must encrypt and decrypt data when it is written and read, using the Linux kernel's *Crypto* API.

Tasks:

1. The first version of your device driver should not encrypt data that passes through it.
   a. Make a new branch in your *svn* repository based on the original vanilla version of the kernel. Name the branch **`linux-2.6.23.17-osurd`**. Put your ramdisk at **`drivers/block/osurd.c`**
   b. Edit **`drivers/block/Kconfig`** as follows: Right after the **`menu "Block Devices"`** line, add
      ```
      config OSU_RD
              tristate "Oregon State U Ramdisk"
              ---help---
              <some appropriate help>
      ```
   c. Edit **`drivers/block/Makefile`** as follows: Add
      ```
      Obj-$(CONFIG_OSU_RD)    += osurd.o
      ```
   d. Follow through Chapter 16 of Linux Device Drivers (available online at http://lwn.net/Kernel/LDD3/) and write code for the bio-aware ramdisk driver module described there. Call your device *osurd*, not *sbull*, and name all structures and functions accordingly. Allow the user to set the disk size in bytes using a module parameter **`disksize`**. (See the notes on **`module_param`** at the end of the second chapter of Linux Device Drivers at http://lwn.net/Kernel/LDD3/ .)

2. Insert your module and make a device node and some minor nodes as follows (you can find the major number of your device in **`/proc/devices`**):

```
> insmod osurd.ko
> mknod /dev/osurda b <major> 0 # b is for 'block device'
> # repeat below for each partition you want to make;
> # <minor> should start at 1 and increase by 1 for each new minor
> mknod /dev/osurda<minor> b <major> <minor>
```

   Try to partition your disk with **`cfdisk`**. Hopefully, you will notice that this fails because **`cfdisk`** cannot read the number of cylinders on your disk. We will fix this next.

3. In newer kernels, the block layer intercepts the **`HDIO GETGEO`** *ioctl* command and calls a **`getgeo()`** method that each block driver must implement. This new scheme relieves driver authors from doing any copies to userspace, as Chapter 16 suggests you must do. If you're curious, check out **`blkdev_ioctl()`** in **`block/ioctl.c`**. The result of this is that you must take the geometry detecting code out of **`ioctl()`** and put it into a new function **`osurd_getgeo()`**. See **`fd_getgeo()`** in **`drivers/block/floppy.c`** for a simple example of how this should work. Add a new line to the initialization of your **`osurd_ops`** structure so the kernel can find your new function:

```
.getgeo = osurd_getgeo
```

   You should now be able to test your ramdisk module by partitioning it, making filesystems on the partitions, mounting the filesystems, and reading from and writing to them normally:

```
> cfdisk /dev/osurda # make all the partitions you want
> mkfs.ext2 /dev/osurda1
> mkdir /mnt/osurd
> mount /dev/osurda1 /mnt/osurd
> # read and write in /mnt/osurd
```

   Put **`printk()`** statements at strategic locations in your code to make sure it is working as you expect it to.

4. Once your device driver is working correctly without encryption …
   a. Create a new branch in your *svn* repository named

      `linux-2.6.23.17-osurd_encrypt`.

   b. Rewrite the appropriate functions (or replace the generic ones with your own) so that data is encrypted when written to a file and decrypted when read. Use the Linux kernel's *Crypto* API as defined in **`crypto/`** for this purpose. You may use any encryption algorithm you like from the **`crypto/`** directory. **Important:** Be sure that the block size of your ramdisk is divisible by the block size of the encryption algorithm you select.
   c. Your encryption key should be read as a module parameter `key` using **`module_param_array()`**. (See the end of the second chapter of Linux Device Drivers at http://lwn.net/Kernel/LDD3/ for more information on how this is done. You can also use the search function of the LUG's LXR cross-reference to find instances of the use of **`module_param_array()`**.)
   d. The kernel's *Crypto* API is designed to operate directly on pages of memory. This is extremely useful in practice, but figuring out how to do it may be difficult. Fortunately, included in the kernel is a test suite for the crypto routines which nicely demonstrates the use of the *Crypto* API. This test suite can be found in **`crypto/tcrypt.c`** and **`crypto/tcrypt.h`**. Note in particular the **`test_cypher()`** function.

5. Use `svn diff` to make a patch file called `osurd_encrypt-teamXX.patch` representing the changes between your encrypted ramdisk branch and version 2.6.23.17 of the kernel. This will make it easy for the TAs to merge your changes into their own kernel for testing.

Additional Requirements, Notes and Caveats:

1. ***Do not code directly on your VM!***
2. Use version 2.6.23.17 of the kernel.
3. As is stated directly in Chapter 16, code for the *sbull* driver is available online. However, it is in your best interest to write your own code and not just to download it.
4. Use the course mailing list to help each other figure out how to use the appropriate data structures, etc.
5. Follow Kroah-Hartman's kernel coding style guidelines (slides in Lecture #10 on the course wiki).
6. Frequently commit your changes to your source code repository. Be sure that each member of your team commits changes to the repository at least once.
7. The evaluation criteria will be posted at least one week before the due date. Be sure to check this posting before making your final submission.

Useful resources:

The following resources will be particularly useful:

- Chapter 16 in LDD3: http://lwn.net/Kernel/LDD3/
- http://www.linuxjournal.com/article/6451
- `drivers/block/floppy.c`
- `Documentation/crypto/api-intro.txt`
- `crypto/tcrypt.c, crypto/tcrypt.h`
- cs411-f08@engr.orst.edu and the archives

Check the course wiki for other resources.

What to hand in:

Create a gzipped tar archive containing the items below and submit it at http://engr.oregonstate.edu/teach by 6:00 am, Saturday, December 6, 2008

- The source code for your device driver.
- The patch file created in task #5
- The changelog from your source code repository.
- Your compiled kernel image, named `vmlinuz-osurd_encrypt-teamXX`
- Your compiled encrypted ramdisk module, named `osurd.ko`

Each member of your team must individually write and submit a separate "review" document (see the course wiki for the requirements). This document must be submitted to http://engr.oregonstate.edu/teach before midnight on Monday, December 8.

Your group must also submit, by 12:00 noon on Tuesday, December 9, a signed hardcopy of your completed Credit Distribution Agreement.

Extra Credit:

Chapter 16 suggests using a timer to simulate ejecting our disk and reinserting a new one. As an extra-credit extension, we will allow the user to run the eject command on our disk to simulate ejection and reinsertion of a new disk. Take out all the code associated with the timer, leaving the **`media_changed()`** and **`revalidate()`** functions. Write a function that checks if the device is being used (the user count will be greater than 1 if the device is being used). If it isn't, the function should signal a media change by setting the **`media_changed`** flag to 1 and returning success. Otherwise, it should return **`-EBUSY`**. Note that you should hold your device's lock throughout this function so the user count can't change unexpectedly. Call this new function from **`ioctl()`** when you get the **`CDROMEJECT`** command. Test out your new functionality using **`eject`**:

```
> umount /dev/osurda1
> eject /dev/osurda
```

Now if you want to remount your partition, you should have to again go through the process described in task 3 to wind up with a fresh, blank disk. If you do try to remount, you should get an error:

```
> mount /dev/osurda1 /mnt/osurd
mount: /dev/osurda1 is not a valid block device
```