Project 3: Memory Encrypting Disk Driver

Andrew Chase

OS II

Spring 2015

Abstract: Describes work done to build and run a custom Memory Encrypting Disk Driver.

CONTENTS

I. Work Log

Date	Work Done
May 17	Write module, compile and test on os-class. Write helper scripts.
May 18	Fix bugs and test on os-class.

II. Questions

A. The design you plan to use to implement the algorithms.

The original plan was to use the cryptographic api exposed by Linux. The assignment gave an example device driver that contained the skeleton code needed to implement the memory disk device driver. The plan was to hook up the cryptographic primatives to that device driver such that data would be encrypted/decrypted in memory as requests were sent/received.

B. What do you think the main point of this assignment is?

I think the main point of the assignment is to use expose students to the block driver, crypto and other apis in the linux kernel and how they interact with modules. This also gives students a chance to learn about how to write custom drivers for the linux kernel. Another learning objective is exposure to the kernel memory api, which was discussed in class.

C. How did you personally approach the problem? Design decisions, algorithm, etc.

As before first I went out and sought other people's solutions to the problems. I read through them carefully and tried to figure out how they worked and what apis were used. I learned a lot from reading examples. Another thing I did was look deeply into the Linux kernel source code to try and figure out how the cryptographic primitives were supposed to be used.

I used the standard synchronous block api, with the one-block-at-a-time functions. The crypto function I used was AES.

D. How did you ensure your solution was correct? Testing details, for instance.

I inserted kprint calls and ran the kernel in the emulator. The kprint calls recorded the sector number.

Example:

```
Reading from memdiskblk --
Encrypted data: 000
Decrypted data: 001c
Reading from memdiskblk --
Encrypted data: 20ffffffa8ffffffc5
```

The testing helped because the first solution I had implemented wasn't correct and it was pretty obvious to see from the kernel messages. At first, my module didn't load, and using dmesg — grep I was able to find the error message to fix the problem.

E. What did you learn?

I learned a bit about how linux driver system works, how to compile and load modules into the linux kernel with parameters. I also learned that the linux kernel exposes a robust crypto api that can be easily integrated to add cryptographic primitives to drivers in the linux kernel. I also learned how to debug buggy kernel modules using dmesg and gdb.

III. Concurrency Exercise: Git Log

acronym	**meaning**
V	`version`
tag	`git tag`
MF	Number of `modified files`.
AL	Number of `added lines`.
DL	Number of `deleted lines`.

V	**tag**	**date**	**commit message**	**MF**	**AL**	**DL**
1		2016-05-18	init	0	191	0

IV. Code Listing

A. memdiskblk.c

```
/*
 * Andy Chase
 * CS444 -- Memory Encrypting Disk Driver
 *
 * Based on:
 *
 * http://blog.superpat.com/2010/05/04/a-simple-block-driver-for-linux-kernel-2-6-31/
 * https://github.com/davidmerrick/Classes/blob/master/CS411/project3/osurd.c
 * https://github.com/ryleyherrington/linux_kernel_411/blob/master/device_driver/device_
 * http://www.oreilly.com/openbook/linuxdrive3/book/ch16.pdf
 * http://lxr.free-electrons.com/source/net/bluetooth/smp.c?v=3.14#L54
```

```
12   */
13
14  #include <linux/module.h>
15  #include <linux/moduleparam.h>
16  #include <linux/init.h>
17
18  #include <linux/kernel.h>        /* printk() */
19  #include <linux/fs.h>                /* everything... */
20  #include <linux/errno.h>        /* error codes */
21  #include <linux/types.h>        /* size_t */
22  #include <linux/vmalloc.h>
23  #include <linux/genhd.h>
24  #include <linux/blkdev.h>
25  #include <linux/hdreg.h>
26  #include <linux/crypto.h>
27  #include <linux/scatterlist.h>
28
29  /* Module Info */
30  MODULE_LICENSE("GPL");
31  MODULE_AUTHOR("Andrew Chase");
32  MODULE_DESCRIPTION("Homework 3: Encrypted Block Device");
33  MODULE_ALIAS("membdiskblk");
34
35  /* Parameters */
36  static int major_num = 0;
37  static int logical_block_size = 512;
38  static int nsectors = 1024;        /* How big the drive is */
39  static char *key = "some_key_yo";
40  module_param(key, charp, 0000);
41
42  #define KERNEL_SECTOR_SIZE 512
43
44  static struct request_queue *Queue;
45  struct crypto_cipher *tfm;
```

```
46
47  static struct memdiskblk_device {
48          unsigned long size;
49          spinlock_t lock;
50          u8 *data;
51          struct gendisk *gd;
52  } Device;
53
54  static void
55  memdiskblk_transfer(struct memdiskblk_device *dev, unsigned long sector,
56                      unsigned long nsect, char *buffer, int write)
57  {
58          unsigned long offset = sector * KERNEL_SECTOR_SIZE;
59          unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
60          int i;
61          if ((offset + nbytes) > dev->size) {
62                  printk(KERN_NOTICE "sbd: Beyond-end write (%ld %ld)\n", offset,
63                         nbytes);
64                  return;
65          }
66          crypto_cipher_setkey(tfm, key, strlen(key));
67
68          if (write) {
69                  printk("Writing to memdiskblk -- \n");
70                  if (nbytes > 3)
71                          printk("Raw data: %x%x%x\n", (buffer)[0], (buffer)[1], (buffer)
72                  for (i = 0; i < nbytes; i += crypto_cipher_blocksize(tfm)) {
73                          memset(dev->data + offset + i, 0,
74                                     crypto_cipher_blocksize(tfm));
75                          crypto_cipher_encrypt_one(tfm, dev->data + offset + i,
76                                                    buffer + i);
77                  }
78
79                  if (nbytes > 3)
```

```
80                          printk("Encrypted data: %x%x%x\n", (dev->data + offset)[0], (dev
81          } else {
82                  printk("Reading from memdiskblk --\n");
83                  if (nbytes > 3)
84                          printk("Encrypted data: %x%x%x\n", (buffer)[0], (buffer)[1], (bu
85                  for (i = 0; i < nbytes; i += crypto_cipher_blocksize(tfm)) {
86                          crypto_cipher_decrypt_one(tfm, buffer + i,
87                                                    dev->data + offset + i);
88                  }
89                  if (nbytes > 3)
90                          printk("Decrypted data: %x%x%x\n", (dev->data + offset)[0], (dev
91          }
92  }
93
94  static void memdiskblk_request(struct request_queue *q)
95  {
96          struct request *req;
97
98          req = blk_fetch_request(q);
99          while (req != NULL) {
100                 if (req == NULL || (req->cmd_type != REQ_TYPE_FS)) {
101                         printk(KERN_NOTICE "Skip non-CMD request\n");
102                         __blk_end_request_all(req, -EIO);
103                         continue;
104                 }
105                 memdiskblk_transfer(&Device, blk_rq_pos(req),
106                                     blk_rq_cur_sectors(req), req->buffer,
107                                     rq_data_dir(req));
108                 if (!__blk_end_request_cur(req, 0)) {
109                         req = blk_fetch_request(q);
110                 }
111         }
112 }
113
```

```
114 int memdiskblk_getgeo(struct block_device *block_device,
115                         struct hd_geometry *geo)
116 {
117         long size;
118
119         size = Device.size * (logical_block_size / KERNEL_SECTOR_SIZE);
120         geo->cylinders = (size & ~0x3f) >> 6;
121         geo->heads = 4;
122         geo->sectors = 16;
123         geo->start = 0;
124         return 0;
125 }
126
127 static struct block_device_operations memdiskblk_ops = {
128         .owner = THIS_MODULE,
129         .getgeo = memdiskblk_getgeo
130 };
131
132 static int __init memdiskblk_init(void)
133 {
134         tfm = crypto_alloc_cipher("aes", 0, 0);
135         /* Error checking for crypto */
136         if (IS_ERR(tfm)) {
137                 printk(KERN_ERR "memdiskblk -- cipher allocation failed");
138                 return PTR_ERR(tfm);
139         }
140
141         Device.size = nsectors * logical_block_size;
142         spin_lock_init(&Device.lock);
143         Device.data = vmalloc(Device.size);
144         if (Device.data == NULL)
145                 return -ENOMEM;
146
147         Queue = blk_init_queue(memdiskblk_request, &Device.lock);
```

```
148         if (Queue == NULL)
149                 goto out;
150         blk_queue_logical_block_size(Queue, logical_block_size);
151 
152         major_num = register_blkdev(major_num, "memdiskblk");
153         if (major_num < 0) {
154                 printk(KERN_WARNING "memdiskblk: unable to get major number\n");
155                 goto out;
156         }
157 
158         Device.gd = alloc_disk(16);
159         if (!Device.gd)
160                 goto out_unregister;
161         Device.gd->major = major_num;
162         Device.gd->first_minor = 0;
163         Device.gd->fops = &memdiskblk_ops;
164         Device.gd->private_data = &Device;
165         strcpy(Device.gd->disk_name, "memdiskblk0");
166         set_capacity(Device.gd, nsectors);
167         Device.gd->queue = Queue;
168         add_disk(Device.gd);
169 
170         return 0;
171 
172  out_unregister:
173         unregister_blkdev(major_num, "memdiskblk");
174  out:
175         crypto_free_cipher(tfm);
176         vfree(Device.data);
177         return -ENOMEM;
178 }
179 
180 static void __exit memdiskblk_exit(void)
181 {
```

```
182         del_gendisk(Device.gd);
183         put_disk(Device.gd);
184         unregister_blkdev(major_num, "memdiskblk");
185         blk_cleanup_queue(Queue);
186         vfree(Device.data);
187 }
188
189 module_init(memdiskblk_init);
190 module_exit(memdiskblk_exit);
```