CS 444 Project 3: Encypted Block Device

Morgan Patch, Mark Bereza

Abstract

For this project, we created a simple block device driver for RAM that maintains its data in an encrypted state.

November 15, 2017

Contents

1 Design

Although the details of implementation were messy and involved a lot of debugging, the overall design of the block device driver was fairly simple. We would use the example code for a simple block driver from the "LWM.net Porting Drivers to 2.6 Series" and update it to work on the 3.19.2 version of the Linux Kernel. From there, we would utilize the Linux crypto API to perform EBC encryption on all data to be written and perform a symmetric decryption on all data to be read. The one caveat would be for data being read from the disk that has not yet been written to, which should be read as-is without being decrpyted. Finally, the key for the encryption would be used as a module parameter so that it may be defined by the user when the module is loaded.

2 Questions

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

The point of this assignment seems to be three-fold:

1. The first point is for us to familiarize ourselves with and gain experience implementing block device drivers for the Linux kernel.
2. The second point is for us to familiarize ourselves with and gain experience utilizing the Linux crypto API to perform encryption and decryption. Related to this goal is the one involving learning how cryptography may be utilized to secure data on an arbitrary digital storage medium.
3. The third point is for us to familiarize ourselves with and gain experience creating modules for the Linux kernel that can be attached at runtime.

2.2 How did you personally approach the problem?

We began by first trying to get a working implementation of the simple block driver without cryptography. The one provided in the aforementioned source was created for version 2.6 of the kernel and several of the function calls, header files, and request struct members were deprecated. Specifically, here is a list of the deprecated references and their updated equivalents:

request_t	→	struct request
elv_next_request()	→	blk_fetch_request
req->sector	→	blk_rq_pos()
req->current_nr_sectors	→	blk_rq_cur_sectors()
req->buffer	→	bio_data(request->bio)
end_request()	→	__blk_end_request_cur()

Beyond these changes, several additional header files had to be included due to certain references being moved since the 2.6 version of the kernel. After all this was done, the block driver finally compiled successfully.

After verifying that it worked correctly, the next step was to implement cryptography. Research was done on the Linux crypto API and several pieces were added onto the existing simple block driver to implement encryption/decryption. First, encrypt/decrpyt keys were defined and made to be a module parameter. Second, members were added to the sbd_device struct in order to hold the

crypto_ciphers and an array to keep track of what blocks have been written to and what blocks have not. The latter member was created to help prevent attempts to decrypt uninitialized data. Next, code was added to the sbd_init() function to allocate the ciphers and associate the appropriate keys with them. Any keys not specified by the user through the module would be set to a default value. Finally, the sbd_transfer() function was changed to encrypt the contents of the write buffer block by block before writing the data to the device. Areas of the device that were written to were flagged as such so the driver knew that reads from those locations should first be decrypted. As for reading, a check was first done to see if the requested memory location was initialized or not. Uninitialized data would be copied into the read buffer as-is, whereas encrypted data would first be decrypted block by block.

Beyond that, several printk() statements were added in order to verify that the driver was functioning correctly and to aid in debugging. These included printing when functions started and ended, the type of operation being done (read vs. write), the location and number of bytes for each operation, and a hex dump of the data before/after cryptography for each operation.

2.3 How did you ensure your solution was correct?

In order to prove that the device worked correctly, we tested:

1. That a file system could be created and mounted on the device
2. That the data was being read correctly
3. That it was stored in a different form on the disk than in the requests
4. That the individual blocks were encrypted correctly.

To prove the data was being read correctly (i.e. it worked as a block device), we first formatted the drive using mkfs.ext4 /dev/sbd0, then mounted the drive with mount /dev/sbd0 /mnt, then wrote several files onto the drive, and read them back out to confirm they were unchanged; we then unmounted the drive, remounted it, and checked that the data was still there. In order to prove the data was being transformed, we added a function to our code which would, each time sbd_transfer() was called to read or write data, print out both the plaintext and encrypted forms of that block of data as a series of hexidecimal pairs; the fact that the plaintext and encrypted data were different proved the existence of a transformation. Finally, to prove the encryption was correct, we copied several of these blocks in both their plaintext and encrypted form, from both reads and writes; by performing the encryption/decryption ourselves using available online implementations of AES, we could confirm that they provided the same data as our code did, proving that our implementation of AES was correct.

Additionally, to ensure that the encryption was sensitive to the key being used, we loaded the module using a decrypt key different from the encrypt key and verified that the data being read was garbage through the hexdumps. In fact, this prevented the device from being able to properly mount a filesystem.

2.4 What did you learn?

By doing this assignment, we learned several things:

1. We learned how to update deprecated kernel code to work on newer versions. In particular, some of the deprecated symbols used in the 2.6 simple block driver could only be replaced with their modern equivalents by determining which version dropped said deprecated symbol and looking at commits to the relevant file for that version. This approach allowed us to determine how to replace the req->buffer code.

2. We learned how to create a Linux module and plug it in during runtime.

3. We learned the differences between different transformation algorithms provided by the Linux crypto API. In particular, we first attempted to use CBC, but found that it was extremely difficult to correctly implement for operations of varying sizes and also required the use of scatterlists, which themsleves are complicated constructs. In light of this, we instead opted to use EBC encryption.

4. We learned how to use scatterlists whilst flirting with the idea of using CBC-AES as our encryption/decryption algorithm.

2.5 How should the TA evaluate your work?

In order to evaluate our work, the TA should take the following steps:

1. Examine our source code, provided below, to see if our approach to encryption makes sense.

2. Clone the linux yocto repository:
   `git clone git://git.yoctoproject.org/linux-yocto-3.19`

3. Switch to the v3.19.2 tag:
   `cd linux-yocto-3.19/`
   `git checkout tags/v3.19.2`

4. Apply the patch file provided with this submission:
   `git apply assignment3.patch`

5. Copy the configuration file from scratch:
   `cp /scratch/files/config-3.19.2-yocto-standard ./.config`

6. Copy the core image file from scratch:
   `cp /scratch/files/core-image-lsb-sdk-qemux86.ext4 .`

7. Setup environment variables:
   `source /scratch/files/environment-setup-i586-poky-linux` (if using bash/zsh)
   OR
   `source /scratch/files/environment-setup-i586-poky-linux.csh` (if using tcsh/csh)

8. Edit the configuration to use our scheduler:
   `make menuconfig`
   Then go to Device drivers→Block devices. From there, highlight "Simple Block Device" and hit 'M' to tell the kernel to build it as a module. Save this configuration to .config and exit menuconfig.

9. Build the kernel:
   `make all -j4`

10. Run the kernel using QEMU:
    `qemu-system-i386 -nographic -kernel arch/x86/boot/bzImage -drive file=core-image-lsb-sdk-qemux86.ext4,if=virtio -enable-kvm -net user,hostfwd=tcp::5540-:22 -net nic -usb -localtime -no-reboot -append "root=/dev/vda rw console=ttyS0 debug"`

11. Verify that the kernel boots and you can log in to root.

12. Open a new terminal at the root kernel directory and copy the sbd.ko kernel module onto the emulated machine:
    `scp -P 5540 drivers/blocksbd.ko root@localhost:sbd.ko`

13. In QEMU, load the driver module with some encrypt key. By not specifying a decrypt key, the decrypt key will default to be the same as the encrypt key:
    `insmod ./sbd.ko encrypt_key=121,53,161,29,3,65,9,196,167,167,221,203,208,40,190,186`

14. Create a filesystem on the sbd0 device that should now be available:
    `mkfs.ext4 /dev/sbd0`

15. Mount the filesystem:
    `mount /dev/sbd0 /mnt`

16. Create a file:
    ```
    cd /mnt
    touch sample
    ```

17. Write some bytes to the file:
    `dd if=/dev/urandom of=sample count=8 bs=4096`

18. Print some of the contents and save them somewhere not on the device:
    `head sample`

19. Unmount the filesystem:
    ```
    cd /
    umount /mnt
    ```

20. Now we remount the filesystem to verify that the data remained the same:
    ```
    mount /dev/sbd0 /mnt
    cd /mnt
    head sample
    ```
    Compare this output with the output you saved earlier.

21. Now we need to verify that the encryption is working correctly. To do this, first find a write operation in the dmesg output (should start with "sbd: Operation type: write") and copy one or more lines from the before-encryption hexdump (should start with "sbd: Data before encryption:"). Additionally, find the same line(s) in the after-encryption hexdump (should start with "sbd: Data after encryption") and copy them, as well. Then, enter the unenceypted lines (along with the key provided to the module earlier) into a tool capable of AES ECB encryption/decryption, such as `http://aes.online-domain-tools.com/`. Verify that the output produced by this tool matches the encrypted output from the hexdump.

22. Now we do essentially the same thing, but this time for a read operation. Make sure that the before-decryption hexdunp for the read operation selected isn't a bunch of zeros; this indicates that the data is not encrypted. Verify using the same AES ECB tool that decryption is being performed correctly.

23. Finally, we must verify that using the incorrect key will not work. To to do this, we must unload the module and reload it, specifiying a decrypt key different from the encrypt key:
```
cd /
umount /mnt
rmmod sbd
insmod ./sbd.ko encrypt_key=121,53,161,29,3,65,9,196,167,167,221,203,208,40,190,
186 decrypt_key=71,144,180,225,211,54,25,234,222,104,210,145,47,167,159,107

mkfs.ext4 /dev/sbd0
mount /dev/sbd0 /mnt
```
The mount operation should fail, giving an error message along the lines of:
```
mount:  wrong fs type, bad option, bad superblock on /dev/sbd0, missing codepage
or helper program, or other error
```
This is expected, since using the wrong decrypt key causes the kernel to incorrectly interpret the filesystem's metadata.

3 Version Control 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		2017-10-19	Add Linux Yocto at tag 3.19.2.	48441	19142068	0
2		2017-11-07	Added sbd.c using code from LVM.net	1	189	0
3		2017-11-07	made changes to drivers/block Makefile	1	1	0
4		2017-11-07	drivers/block/Kconfig now references BLK_DEV_SBD	1	3	0
5		2017-11-07	Fixed some bugs in sbd.c	1	2	7
6		2017-11-07	Changes to sbd.c	1	1	0
7		2017-11-09	Add include line to SBD.	1	3	2
8		2017-11-12	got sbd.c to compile	1	115	123
9		2017-11-12	Add crypto setup and free.	3	42	8
10		2017-11-12	Add writing encrypted data.	1	27	4
11		2017-11-12	Add reading decrypted data.	1	16	2
12		2017-11-12	Correct sizes.	1	2	2
13		2017-11-12	Added kernel messages	1	19	0
14		2017-11-12	Added kernel messages	1	0	1
15		2017-11-12	Fixed compiler warnings	1	18	3
16		2017-11-12	Minor change to sbd.c	1	4	4
17		2017-11-12	Move crypto above registration in init.	1	31	31
18		2017-11-12	Use correct algorithm.	1	1	1
19		2017-11-12	Add 16-byte default key.	1	3	2
20		2017-11-12	minor changes	1	36	27

V	tag	date	commit message	MF	AL	DL
21		2017-11-12	Re-add key and make arrays on heap.	1	16	10
22		2017-11-12	Move malloc and free outside if.	1	7	18
23		2017-11-12	fixed compiler warnings/errors	1	4	4
24		2017-11-12	Now prints data before/after encryption/decryption	1	36	4
25		2017-11-14	Fix some merge issues.	1	6	6
26		2017-11-14	Skip uninitialized memory.	1	18	0
27		2017-11-14	now does not decrypt uninitialized blocks	1	19	10
28		2017-11-14	Rename uninit function to match.	1	2	2
29		2017-11-15	code cleanup and IV is not freed	1	58	63
30		2017-11-15	new approach to checking for uninitialized blocks	1	63	57
31		2017-11-15	Move to EBC.	1	29	43
32		2017-11-15	Move error checking into place.	1	2	2
33		2017-11-15	Remove unnecessary module parameters.	1	0	3
34		2017-11-15	Added writeup files	2	626	0
35		2017-11-15	Remove unnecessary temp buffers.	1	9	24
36		2017-11-15	Use two keys if given.	1	68	24
37		2017-11-15	more work done to writeup	1	112	192
38		2017-11-15	fleshed out more sections of writeup	1	10	56
39		2017-11-15	Fix key lengths.	1	2	2

4 Work Log

- Tuesday, 2017-11-07
  - Looked at assignment description.
  - Copied sbd.c code from LVM.net.
  - Edited kconfig and make files to recognize the driver.
  - Began work on editing the sbd code to compile for 3.19.2.
- Thursday, 2017-11-09
  - Continued to work on getting sbd.c to compile
  - Researched Linux crypto API.
- Sunday, 2017-11-12
  - Got sbd.c to compile.
  - Added crypto setup to sbd_init()
  - Added decryption to reads.
  - Added encryption to writes.
  - Added crypto key.
  - Added a bunch of printk() messages.
- Tuesday, 2017-11-14
  - Determined why CBC was only correctly decrypting first request.

  - Added logic to prevent decryption of uninitialized blocks.
  - Started writeup.

- Wednesday, 2017-11-15
  - Switched to EBC encryption.
  - Fixed logic for determining uninitialized blocks.
  - Removed unneeded intermediate buffers.
  - Now can handle two keys: one for encrypting and the other for decrypting.
  - Loaded driver as module successfully.
  - Successfully mounted filesystem on device and wrote to/read from files.
  - Completed write up.
  - Created patch file.
  - Turned in assignment.

Appendix 1: Source Code

4.1 sbd.c

```
/*
* A sample, extra-simple block driver.
*
* Copyright 2003 Eklektix, Inc.  Redistributable under the terms
* of the GNU GPL.
*/

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/blkdev.h>
#include <linux/elevator.h>
#include <linux/bio.h>

#include <linux/kernel.h> /* printk() */
#include <linux/fs.h>     /* everything... */
#include <linux/errno.h>  /* error codes */
#include <linux/types.h>
#include <linux/vmalloc.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>
#include <linux/hdreg.h>

#include <linux/crypto.h>
#include <linux/err.h>
#include <linux/scatterlist.h>
#include <linux/random.h>
```

```
#include <linux/mm.h>
#include <linux/sysinfo.h>

#define SBD_ENCRYPTED 0
#define SBD_PLAINTEXT 1

MODULE_LICENSE("Dual BSD/GPL");
static char *Version = "1.3";

static int major_num = 0;
static int logical_block_size = 512;
static int nsectors = 1024;  /* How big the drive is */
static u8 default_key[] = {0x26, 0xe7, 0x0a, 0x0c, 0xbc, 0xce, 0x4a, 0x5c, 0x0b,
        0x0b, 0xd1, 0xd8, 0xf7, 0xce, 0x2d, 0xf7};
static const unsigned int default_keylen = 16;
static u8 encrypt_key[] = {};
static unsigned int ekeylen = 0;
module_param_array(encrypt_key, byte, &ekeylen, 0);
static u8 decrypt_key[] = {};
static unsigned int dkeylen = 0;
module_param_array(decrypt_key, byte, &dkeylen, 0);


/*
* We can tweak our hardware sector size, but the kernel talks to us
* in terms of small sectors, always.
*/
#define KERNEL_SECTOR_SIZE 512

/*
* Our request queue.
*/
static struct request_queue *Queue;

/*
* The internal representation of our device.
*/
static struct sbd_device {
        unsigned long size;
        spinlock_t lock;
        u8 *data;
        u8 *written;
        struct gendisk *gd;
        struct crypto_cipher *encrypt;
        struct crypto_cipher *decrypt;
} Device;

static void sbd_print_buffer(u8 *buffer,
                             unsigned long len,
                             unsigned long offset,
                             int type)
```

```
{
        int i;
        for (i = 0; i < len; ++i) {
                if ((i % 16) == 0) {
                        if (type == SBD_PLAINTEXT)
                                printk("\nPLAINTEXT, BYTES %06lu-%06lu: ",
                                        i + offset, i + offset + 15);
                        else
                                printk("\nENCRYPTED, BYTES %06lu-%06lu: ",
                                        i + offset, i + offset + 15);
                }
                printk("%02x ", (unsigned int) *(buffer + i));
        }
        printk("\n");
}

/*
* Handle an I/O request.
*/
static void sbd_transfer(struct sbd_device *dev, unsigned long sector,
                         unsigned long nsect, char *buffer, int write)
{
        int i;
        unsigned long offset = sector * logical_block_size;
        unsigned long nbytes = nsect  * logical_block_size;
        // Encrypt and decrypt have the same block size
        unsigned int crypt_blksize = crypto_cipher_blocksize(dev->encrypt);
        u8 *memloc = dev->data + offset;

        printk(KERN_NOTICE "sbd: Starting sbd_transfer()...\n");
        printk(KERN_NOTICE "sbd: offset        = %lu\n", offset);
        printk(KERN_NOTICE "sbd: nbytes        = %lu\n", nbytes);
        printk(KERN_NOTICE "sbd: crypt_blksize = %u\n", crypt_blksize);

        if ((offset % crypt_blksize) != 0) {
                printk(KERN_ERR
                        "sbd: Offset not aligned to crypto block size\n");
                return;
        } else if (nbytes % crypt_blksize != 0) {
                printk(KERN_ERR
                        "sbd: Nbytes not aligned to crypto block size\n");
                return;
        } else if ((offset + nbytes) > dev->size) {
                printk(KERN_ERR "sbd: Beyond-end write (%ld %ld)\n",
                                offset, nbytes);
                return;
        }

        if (write) {
                int i;
```

```
                printk(KERN_NOTICE "sbd: Operation type: write\n");

                for (i = 0; i < nbytes; i += crypt_blksize)
                        crypto_cipher_encrypt_one(dev->encrypt, memloc + i,
                                                  buffer + i);

                for (i = (offset / crypt_blksize);
                     i < ((offset + nbytes) / crypt_blksize);
                     ++i)
                        (dev->written)[i] = 1;

                printk(KERN_NOTICE "sbd: Data before encryption:");
                sbd_print_buffer(buffer, nbytes, offset, SBD_PLAINTEXT);
                printk(KERN_NOTICE "sbd: Data after encryption:");
                sbd_print_buffer(memloc, nbytes, offset, SBD_ENCRYPTED);
        } else {
                printk(KERN_NOTICE "sbd: Operation type: read\n");

                for (i = 0; i < nbytes; i += crypt_blksize) {
                        if ((dev->written)[(offset + i) / crypt_blksize]) {
                                crypto_cipher_decrypt_one(dev->decrypt,
                                                          buffer + i,
                                                          memloc + i);
                        } else {
                                memcpy(buffer + i, memloc + i,
                                       crypt_blksize);
                        }
                }
                printk(KERN_NOTICE "sbd: Data before decryption:");
                sbd_print_buffer(memloc, nbytes, offset, SBD_ENCRYPTED);
                printk(KERN_NOTICE "sbd: Data after decryption:");
                sbd_print_buffer(buffer, nbytes, offset, SBD_PLAINTEXT);
        }

        printk(KERN_NOTICE "sbd: Ending sbd_transfer()...\n");
}

static void sbd_request(struct request_queue *q)
{
        struct request *req;
        printk(KERN_NOTICE "sbd: Starting sbd_request()...\n");
        req = blk_fetch_request(q);
        while (req != NULL) {
                if (req == NULL || (req->cmd_type != REQ_TYPE_FS)) {
                        printk(KERN_NOTICE "Skip non-CMD request\n");
                        __blk_end_request_all(req, -EIO);
                        continue;
                }
                sbd_transfer(&Device, blk_rq_pos(req), blk_rq_cur_sectors(req),
                             bio_data(req->bio), rq_data_dir(req));
                if (!__blk_end_request_cur(req, 0)) {
```

```
                        req = blk_fetch_request(q);
                }
        }
        printk(KERN_NOTICE "sbd: Ending sbd_request()...\n");
}

/*
 * The HDIO_GETGEO ioctl is handled in blkdev_ioctl(), which
 * calls this. We need to implement getgeo, since we can't
 * use tools such as fdisk to partition the drive otherwise.
 */
int sbd_getgeo(struct block_device * block_device, struct hd_geometry * geo)
{
        long size;

        printk(KERN_NOTICE "sbd: Starting sbd_getgeo()...\n");

        /* We have no real geometry, of course, so make something up. */
        size = Device.size * (logical_block_size / KERNEL_SECTOR_SIZE);
        geo->cylinders = (size & ~0x3f) >> 6;
        geo->heads = 4;
        geo->sectors = 16;
        geo->start = 0;

        printk(KERN_NOTICE "sbd: Ending sbd_getgeo()...\n");

        return 0;
}

/*
* The device operations structure.
*/
static struct block_device_operations sbd_ops = {
        .owner = THIS_MODULE,
        .getgeo = sbd_getgeo
};

static int __init sbd_init(void)
{
        struct crypto_cipher *encrypt, *decrypt;
        unsigned int encrypt_blksize, decrypt_blksize;
        unsigned int ret;

        printk(KERN_NOTICE "sbd: Starting sbd_init()...\n");

        /*
         * First, init our crypto.
         */

        // If both keys are given, use them; if one is given, it is used for
        // both; if no keys are given, use the default for both
```

```
if (ekeylen != 0 && dkeylen != 0 && ekeylen != dkeylen) {
        printk(KERN_WARNING "sbd: keys must be the same length\n");
        goto out;
}

encrypt = crypto_alloc_cipher("aes", 0, 0);
if (IS_ERR(encrypt)) {
        printk(KERN_WARNING "sbd: unable to allocate cipher\n");
        goto out;
}
encrypt_blksize = crypto_cipher_blocksize(encrypt);

if (ekeylen == 0 && dkeylen == 0) {
        ret = crypto_cipher_setkey(encrypt, default_key, default_keylen);
} else if (ekeylen == 0) {
        ret = crypto_cipher_setkey(encrypt, decrypt_key, dkeylen);
} else {
        ret = crypto_cipher_setkey(encrypt, encrypt_key, ekeylen);
}
if (ret) {
        printk(KERN_NOTICE "sbd: crypto_blkcipher_setkey() failed\n");
        goto out;
}

Device.encrypt = encrypt;

decrypt = crypto_alloc_cipher("aes", 0, 0);
if (IS_ERR(decrypt)) {
        printk(KERN_WARNING "sbd: unable to allocate cipher\n");
        goto out;
}
decrypt_blksize = crypto_cipher_blocksize(decrypt);

if (ekeylen == 0 && dkeylen == 0) {
        ret = crypto_cipher_setkey(decrypt, default_key, default_keylen);
} else if (dkeylen == 0) {
        ret = crypto_cipher_setkey(decrypt, encrypt_key, ekeylen);
} else {
        ret = crypto_cipher_setkey(decrypt, decrypt_key, dkeylen);
}
if (ret) {
        printk(KERN_NOTICE "sbd: crypto_blkcipher_setkey() failed\n");
        goto out;
}

Device.decrypt = decrypt;

if (encrypt_blksize != decrypt_blksize) {
        printk(KERN_WARNING "sbd: block lengths must be the same\n");
        goto out;
```

```
        }

        /*
        * Set up our internal device.
        */
        Device.size = nsectors * logical_block_size;
        spin_lock_init(&Device.lock);
        Device.data    = vzalloc(Device.size);
        Device.written = vzalloc(Device.size / encrypt_blksize);
        if (Device.data == NULL)
                return -ENOMEM;
        /*
        * Get a request queue.
        */
        Queue = blk_init_queue(sbd_request, &Device.lock);
        if (Queue == NULL)
                goto out;
        blk_queue_logical_block_size(Queue, logical_block_size);
        /*
        * Get registered.
        */
        major_num = register_blkdev(major_num, "sbd");
        if (major_num <= 0) {
                printk(KERN_WARNING "sbd: unable to get major number\n");
                goto out;
        }
        /*
        * And the gendisk structure.
        */
        Device.gd = alloc_disk(16);
        if (!Device.gd)
                goto out_unregister;
        Device.gd->major        = major_num;
        Device.gd->first_minor  = 0;
        Device.gd->fops         = &sbd_ops;
        Device.gd->private_data = &Device;
        strcpy(Device.gd->disk_name, "sbd0");
        set_capacity(Device.gd, nsectors);
        Device.gd->queue        = Queue;
        add_disk(Device.gd);

        printk(KERN_NOTICE "sbd: Ending sbd_init()...\n");

        return 0;

out_unregister:
        unregister_blkdev(major_num, "sbd");
out:
        vfree(Device.written);
        vfree(Device.data);
        crypto_free_cipher(Device.encrypt);
```

```
        crypto_free_cipher(Device.decrypt);
        printk(KERN_NOTICE "sbd: Ending sbd_init()...\n");

        return -ENOMEM;
}

static void __exit sbd_exit(void)
{
        printk(KERN_NOTICE "sbd: Starting sbd_exit()...\n");

        del_gendisk(Device.gd);
        put_disk(Device.gd);
        unregister_blkdev(major_num, "sbd");
        blk_cleanup_queue(Queue);
        vfree(Device.written);
        vfree(Device.data);
        crypto_free_cipher(Device.encrypt);
        crypto_free_cipher(Device.decrypt);

        printk(KERN_NOTICE "sbd: Ending sbd_exit()...\n");
}

module_init(sbd_init);
module_exit(sbd_exit);
```