OSU
Oregon State
UNIVERSITY
College of Engineering

Operating Systems II

May 27, 2018
Spring Term

Prepared for

Oregon State University

Kevin McGrath

Prepared by

Austin Row

Lazar Sharipoff

Benjamin Richards

Abstract

This document describes the steps used by group 17 to complete Project 3 in which we built a block driver with added crypto support for the Linux kernel for Operating Systems II. It also answers the questions given in the assignment description regarding the assignment itself.

CONTENTS

1 Steps To Setup Environment

Note: This assumes that the files **yocto_config**, **look_scheduler.patch**, and **sstf-iosched.c** which were included in this submission have been copied to your home directory on os2.engr.oregonstate.edu.

1) SSH on to the OS2 server: *ssh os2.engr.oregonstate.edu.*
2) Clone the yocto repo: *git clone git://git.yoctoproject.org/linux-yocto*
3) Change to the correct yocto version with *cd linux-yocto* followed by *git checkout v3.19.2*
4) *cp ~/driver.patch ./*
5) *git apply look_scheduler.patch*

2 Steps To Build and Run Kernel

1) Split the terminal into two windows using tmux. We will refer to these windows as T1 and T2.
2) T1: *source /scratch/files/environment-setup-i586-poky-linux*
   It will appear that nothing happened, but it actually ran commands to configure the shell so that its ready to build the kernel.
3) T1: *make -j4 all*
   This is the command that actually builds the kernel and generates the kernel image that will used to boot the operating system within the VM. If it asks you if you want to build the os2 driver enter "M". After doing this, this command may take several minutes. Wait until it is done before going on to the next step.
4) T1: *qemu-system-i386 -gdb tcp::5517 -S -nographic -kernel ./arch/x86/boot/bzImage -drive file=/scratch/spring2018/17/core-image-lsb-sdk-qemux86.ext4 -enable-kvm -usb -localtime –no-reboot –append "root=/dev/hda rw console=ttyS0 elevator=look"*
   The terminal should hang now and do nothing. You have booted the VM using the bzImage file that you generated in the previous step. The VMs CPU is halted which is why nothing is happening. Proceed to the next step.
5) T2: *gdb -tui*
6) T2: *target remote :5517*
   This attaches the current GDB process to the process which is running on port 5517 which is the process running the halted VM.
7) T2 : *continue*
   Now the VM in T1 will no longer be hanging. Switch back to T1 and wait for it to ask you to login.
8) T1: *root*
   You are now within the VM logged in as the root user
9) T1: *scp (Your server):(path to linux-yocto)/drivers/block/os2driver.ko /home/*
   This copies the .ko file we made during running make earlier to the linux kernel.
10) T1: *insmod /home/os2driver.ko*
    This loads the module.
11) T1: *fdisk /dev/os20*
    This partitions the module.
12) T1: *n* T1: *p* T1: *1* T1: *1* T1: (Press enter) T1: *w*
    This series of settings sets up the partition.

13) T1: *mkfs.ext2 /dev/os20p1*

    This creates a file system on the partition.

14) T1: *mount /dev/os20p1 /mnt/*

    This mounts the device to the /mnt/ folder. Now writing or reading anything to the /mnt/ folder should be using our driver with it's encryption. You can check this by using "dmesg" to see the kernel alerts our driver makes.

3 BLOCK DRIVER DESIGN

We worked primarily in one .c file which contained all the functions needed to run a basic block driver. This file was kept in the linux-yocto/drivers/block directory. Our driver contains os2_exit and os2_init functions that register the device and it's structure to the operating system, as well as set up the crypto api for future read/write use. We also have a getgeo function which allows us to use fdisk on this version of the operating system. And we have transfer and request functions for handling the movement of data into the driver.

```
/*
* A sample, extra-simple block driver. Updated for kernel 2.6.31.
*
* (C) 2003 Eklektix, Inc.
* (C) 2010 Pat Patterson <pat at superpat dot com>
* Redistributable under the terms of the GNU GPL.
*/



/*
* https://www.kernel.org/doc/html/v4.13/crypto/api-skcipher.html#single-block-cipher-api
* https://stackoverflow.com/questions/8927827/encryption-and-decryption-using-aes-256
*/

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

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

MODULE_LICENSE("Dual BSD/GPL");
```

```
static int major_num = 0;
module_param(major_num, int, 0);
static int logical_block_size = 512;
module_param(logical_block_size, int, 0);
static int nsectors = 1024; /* How big the drive is */
module_param(nsectors, int, 0);
static unsigned int crypto_key_length = 16;
module_param(crypto_key_length, int, 0);
static char *crypto_key = "abcdefghijklmnop";
module_param(crypto_key, charp, 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

struct crypto_cipher *tfm = NULL;
unsigned int cipher_blocksize;
/*
* Our request queue.
*/
static struct request_queue *Queue;

/*
* The internal representation of our device.
*/
static struct os2_device {
unsigned long size;
spinlock_t lock;
u8 *data;
struct gendisk *gd;
} Device;

static void print_memory(char *buffer, unsigned int length) {
int i = 0;
for(i = 0; i < length; i++) {
printk(KERN_CONT "%02X", (unsigned)buffer[i]);
}
}
```

```
static void print_cipher_result(char *from, char *to, unsigned int transformation_size, in
transformation_size = 50 < transformation_size ? 50 : transformation_size;
printk(KERN_DEBUG "**OS2: \"");
print_memory(from, transformation_size);
if (encrypting) {
printk(KERN_CONT "\" encrypted to \"");
} else {
printk(KERN_CONT "\" decrypted to \"");
}
print_memory(to, transformation_size);
printk(KERN_CONT "\" (showing first %d characters)\n", transformation_size);
}

/*
* Handle an I/O request.
*/
static void os2_transfer(struct os2_device *dev, sector_t sector,
unsigned long nsect, char *buffer, int write) {
unsigned long offset = sector * logical_block_size;
unsigned long nbytes = nsect * logical_block_size;
unsigned int ciphered_bytes = 0;
char temp_buffer[cipher_blocksize + 1];

printk(KERN_INFO "**OS2: Inside os2_transfer\n");
if ((offset + nbytes) > dev->size) {
printk (KERN_NOTICE "os2: Beyond-end write (%ld %ld)\n", offset, nbytes);
return;
}

while(ciphered_bytes + cipher_blocksize <= nbytes) {
if (write) {
crypto_cipher_encrypt_one(tfm, dev->data + offset + sizeof(char)*ciphered_bytes, buffer +
} else {
crypto_cipher_decrypt_one(tfm, buffer + sizeof(char)*ciphered_bytes, dev->data + offset +
}
ciphered_bytes += cipher_blocksize;
}
```

```
if (ciphered_bytes < nbytes) {
if (write) {
printk(KERN_ALERT "**OS2: encrypting remaining %d bytes\n", (int)(nbytes-ciphered_bytes));
crypto_cipher_encrypt_one(tfm, temp_buffer, buffer + sizeof(char)*ciphered_bytes);
memcpy(dev->data + offset + sizeof(char)*ciphered_bytes, temp_buffer, nbytes-ciphered_byte
} else {
printk(KERN_ALERT "**OS2: decrypting remaining %d bytes\n", (int)(nbytes-ciphered_bytes));
crypto_cipher_decrypt_one(tfm, temp_buffer, dev->data + offset + sizeof(char)*ciphered_byt
memcpy(buffer, temp_buffer, nbytes-ciphered_bytes);
}
}

if (write) {
print_cipher_result(buffer, dev->data + offset, nbytes, 1);
} else {
print_cipher_result(dev->data + offset, buffer, nbytes, 0);
}
}

static void os2_request(struct request_queue *q) {
struct request *req;

req = blk_fetch_request(q);
while (req != NULL) {
if (req == NULL || (req->cmd_type != REQ_TYPE_FS)) {
printk (KERN_NOTICE "**OS2: Skip non-CMD request\n");
__blk_end_request_all(req, -EIO);
continue;
}
os2_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);
}
}
}

/*
* 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 os2_getgeo(struct block_device * block_device, struct hd_geometry * geo) {
long size;

/* 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;
return 0;
}

/*
* The device operations structure.
*/
static struct block_device_operations os2_ops = {
.owner  = THIS_MODULE,
.getgeo = os2_getgeo
};

static int __init os2_init(void) {
/*
* Set up our internal device.
*/
printk(KERN_ALERT "**OS2: In OS2 Device Driver INIT\n");
Device.size = nsectors * logical_block_size;
spin_lock_init(&Device.lock);
Device.data = vmalloc(Device.size);
if (Device.data == NULL)
return -ENOMEM;

tfm = crypto_alloc_cipher("aes", 0, 0);
if (IS_ERR(tfm)) {
printk(KERN_ALERT "**OS2: Error while initializing crypto cipher\n");
goto out;
}
```

```
if (crypto_cipher_setkey(tfm, crypto_key, crypto_key_length)) {
printk(KERN_ALERT "**OS2: Error while setting crypto key\n");
goto out;
}

cipher_blocksize = crypto_cipher_blocksize(tfm);

/*
* Get a request queue.
*/
Queue = blk_init_queue(os2_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, "os2");
if (major_num < 0) {
printk(KERN_WARNING "os2: 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 = &os2_ops;
Device.gd->private_data = &Device;
strcpy(Device.gd->disk_name, "os20");
set_capacity(Device.gd, nsectors);
Device.gd->queue = Queue;
add_disk(Device.gd);

return 0;
```

```
out_unregister:
unregister_blkdev(major_num, "os2");
out:
vfree(Device.data);
if (tfm) {
crypto_free_cipher(tfm);
}
return -ENOMEM;
}

static void __exit os2_exit(void)
{
printk(KERN_ALERT "**OS2: In OS2 Device Driver EXIT\n");
del_gendisk(Device.gd);
put_disk(Device.gd);
unregister_blkdev(major_num, "os2");
blk_cleanup_queue(Queue);
vfree(Device.data);
if (tfm) {
crypto_free_cipher(tfm);
}
}

module_init(os2_init);
module_exit(os2_exit);
```

4 QUESTIONS ABOUT ASSIGNMENT

1) What do you think the main point of this assignment is?
   The main point of this assignment is to get a glimpse of how the linux operating system treats drivers and to get experience on making one ourselves.
2) How did you personally approach the problem?
   First we looked for sample block driver code templates by using Google. We discovered that several functions in the templates we found initially were depreciated and were no longer working. In addition there were several functions that we didn't need for our assignment. Eventually we found a working driver template that we were able to properly mount. The next step was to implement encryption to all incoming traffic and decryption to all out outgoing traffic using the crypto api.

3) How did you ensure your solution was correct?

   The way we checked to see if our driver was working was by seeing if the driver would properly show up and function in the linux kernel. This required finding the .ko device, successfully partitioning it, adding a file system to it and mounting it. Once that was done the way we checked encryption was through kernel messages that would print out to dmesg whenever an encryption or decryption even happened.

4) What did you learn?

   Throughout the course of this assignment we learned:

   - The process of how drivers are partitioned and mounted to the system.
   - That drivers have multiple ways you can utilize them, including multiple types of file systems depending on the version.
   - How to use the API for crypto.
   - How to see kernel messages printed from the driver using dmesg.
   - How devices are registered to the os with exit and init. Additionally we learned of the default driver functions the os uses.

5) How should the TA evaluate your work?

   Running dmesg once the device is properly mounted (As described how to do earlier) will show all of the kernel messages printed during the encryption and decryption of data written to and read from the device. To trigger these messages you must be sure to write a file to the /mnt/ folder. Additionally whether or not the device properly mounts is another way to evaulate our work. You can see a list of running modules by using the lsmod command.

5 Version Control Log

Commit	**Author**	**Description**
7b736f8	Benjamin Richards	added driver folder
f57302f	Benjamin Richards	added patch file

6 Work Log

5/21/2018	Began reading about block devices from online sources such as [1] started implementing it.
5/25/2018	We eventually found this source to be outdated and instead went with an updated but more barebones template[**?**]. We also added crypto by looking at the Linux crypto API[2] and reading up on how to use that with blocks[3] as well as AES-256[4]

References

[1] G. Kroa-Hartman. Linux device drivers, 3rd edition. https://www.safaribooksonline.com/library/view/linux-device-drivers/0596005903/ch16.html.

[2] S. Mueller and M. Vasut. Linux kernel crypto api. https://www.kernel.org/doc/html/v4.13/crypto/index.html.

[3] ——. Single block cipher api. https://www.kernel.org/doc/html/v4.13/crypto/api-skcipher.html#single-block-cipher-api.

[4] Encryption and decryption using aes-256. https://stackoverflow.com/questions/8927827/encryption-and-decryption-using-aes-256.