Εργαστήριο Λειτουργικών Συστήματων
3η Άσκηση

Αλέξιος Ζαμάνης
03115010
Μιχαήλ Μεγγίσογλου
03115014

10 Μαρτίου 2019

1 Σχόλια

1.1 Chat

Η κατασκευή του Chat διαιρέθηκε, όπως προτεινόταν, σε δύο φάσεις: μία για την επικοινωνία client-server μέσω BSD sockets και μία για την κρυπτογράφηση αυτής μέσω Cryptodev.

Ως προς το κομμάτι της επικοινωνίας, αυτή ξεκινά από τον server, που ανοίγει ένα ορισμένο TCP socket, στο οποίο ακούει για αιτήματα. Εν συνεχεία, συνδέεται ο client στο γνωστό socket, και δεδομένης επιτυχούς χειραψίας, ο server αποδέχεται την επικοινωνία και η συζήτηση ξεκινά.

Η συζήτηση καθεαυτή αποτελείται από έναν αέναο βρόχο εναλλασσόμενων εγγραφών και αναγνώσεων, ήτοι ο εκάστοστε χρήστης είτε γράφει είτε περιμένει μπλοκάροντας για να διαβάσει. Ο βρόχος και επομένως η συζήτηση λήγει με αποστολή του μηνύματος bye από εναν εκ των δύο συνομιλητών. Στην πραγματικότητα ο αποστολέας του μηνύματος λήξης αποστέλει shutdown με όρισμα SHUT_WR, ήτοι πως παύει να γράφει, προκαλώντας εμμέσως το κλείσιμο του socket σε αμφότερες τις πλευρές.

Αξίζει να σημειωθεί η χρήση των συναρτήσεων insist_read και insist_write, που επιβάλλουν την αποστολή ολόκληρου του μηνύματος και την πλήρη ανάγνωσή του, ώστε να διατηρείται η εμπερία του chat από τους χρήστες.

Όσον αφορά το κομμάτι της κρυπτογράφησης, αυτό προστέθηκε εμβόλιμα στις προαναφερθείσες λειτουργίες. Συγκεκριμένα, πριν την έναρξη του ατέρμονος βρόχου ξεκινά ένα session με το Cryptodev, το οποίο λήγει ύστερα από το πέρας του βρόχου. Η ύπαρξη κοινού session, ήτοι με τις ίδιες κρυπτογραφικές σταθερές, είναι καίρια για την αποκρυπτογράφηση των δεδομένων. Στην ουσία τα μόνα δεδομένα εισόδου που μεταβάλλονται στο cryptodev είναι τα ίδια τα μηνύματα.

Η κρυπτογράφηση γίνεται πριν την αποστολή του μηνύματος και η αποκρυπτογράφηση κατόπιν της παραλαβής του. Στο τέλος του κρυπρογραφημένου μηνύματος προστίθεται ένας χαρακτήρας newline, που χρησιμοποιείται για την αναγνώριση της λήξης της αποστολής.

Κάθε κλήση για έναρξη ή λήξη session, καθώς και για (απο)κρυπτογράφηση αντιστοιχεί σε μια αντίστοιχει κλήση συστήματος στον οδηγό Cryptodev. Σε αυτή τη φάση της άσκησης οι κλήσεις εξυπηρετούνται απευθείας από τη συσκευή.

1.2 VirtIO

Στόχος του συγκεκριμένου πρωτοκόλλου είναι ο ορισμός μίας κοινής διεπαφής για επικοινωνία μεταξύ host και guest λειτουργικού συστήματος σε περιπτώσεις paravirtualization. Εν προκειμένω, το αξιοποιούμε ώστε να προωθήσουμε τις προαναφερθείσες κλήσεις συστήματος στο Cryptodev που εδράζει σε ένα host μηχάνημα, όταν

αυτές αποστέλονται από τον quest.

Συγκεκριμένα, υλοποιούμε τις τρεις κλήσεις συστήματος, ήτοι τις τρεις περιπτώσεις της ioctl, ώστε να βάζουν τα δεδομένα τους, μαζί με την αντίστοιχη εντολή, σε μια δομή, ονόματι scatter-gather list, που, ως κοινή μνήμη μεταξύ πραγματικού και εικονικού μηχανήματος, μπορεί να γράφεται και να διαβάζεται από αμφότερα.

Σε κάθε κλήση της ioctl, ο guest φτιάχνει μια διαφορετική sg για το κάθε δεδομένο που θέλει να γράψει ή διαβάσει (μάλιστα πρώτα όλα τα προς ανάγνωση δεδομένα, και ύστερα όλα τα προς εγγραφή), τις προωθεί όλες μαζί σε μια δεδομένη ουρά που του έχει δοθεί, και κάνει busy wait, ενώ αναμένει τα αποτελέσματα. Κατόπιν λαμβάνει τα δεδομένα του Cryptodev και επιστρέφει αναλόγως στο χρήστη.

Κατά τη διάρκεια της αναμονής, ο guest προφυλάσσεται από έναν σημαφόρο, ουσιαστικά ένα mutex, που δεν επιτρέπει πρόσβαση από άλλες διεργασίες στην υπο χρησιμοποίηση ουρά. Η χρήση σημαφόρου, έναντι spinlock είναι προτιμητέα, καθώς τα δεύτερα μονοπωλούν το χρόνο της CPU, εν προκειμένω καθόλη την ανταλλαγή των δεδομένων. Μάλιστα, δεδομένου ότι ο κώδικάς μας πρόκειται να εκτελείται μόνο σε process context, η χρήση spinlock είναι περιττή, καθώς δεν υπάρχει η απαίτηση να μην μπορεί να κοιμηθεί.

Από την πλευρά του host μηχανήματος, και συγκεκριμένα στον πηγαίο κώδικα του QEMU, υλοποιείται η παραλαβή των δεδομένων από τις ουρές, σε πλήρη, ένα προς ένα αντιστοιχία με την σειρά τοποθέτησής τους από τον guest. Από εκεί εκτελούνται, ωσάν δια αντιπροσώπου, οι αντίστοιχες κλήσεις συστήματος στο Cryptodev. Εν τέλει, στέλνονται τα αντίστοιχα αποτελέσματα.

Κατά πλήρη αναλογία με τα παραπάνω, υλοποιούνται και οι συναρτήσεις open και release, που καλούνται στο άνοιγμα μίας συγκεκριμένης εικονικής συσκευής Cryptodev και στο κλείσιμό της αντίστοιχα. Εδώ δεσμεύονται ή απελευθερώνονται οι απαραίτητες δομές για την συσκευή και στέλνοται όπως πάνω στο host μηχάνημα. Σημειώνουμε πως η επιλογή της κατάλληλης συσκευής μέσω inodes, καθώς και οι απαραίτητες αρχικοποιήσεις και καταστροφές αυτών, γίνονται όπως στην προηγούμενη άσκηση, οπότε δε χρήζουν ιδιαίτερης μνείας.

2 Παράρτημα: Πηγαίος κώδικας

2.1 Chat

2.1.1 mycrypto.h

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>

#include <sys/types.h>
#include <sys/stat.h>

#include "cryptodev.h"

#define KEY (unsigned char *)"hellhellhellhell"
#define KEY_SIZE 16
#define DATA_SIZE 256
#define BUF_SIZE 257
#define IV (unsigned char *)"llehllehllehlleh"

void auxcrypt(int cfd, unsigned int ses, unsigned short
   op, unsigned char *src, unsigned char *dst, unsigned
   char *iv)
{
     struct crypt_op cryp;
     memset(&cryp, 0, sizeof(cryp));
     cryp.ses = ses;
     cryp.op = op;
     cryp.len = DATA_SIZE;
     cryp.src = src;
     cryp.dst = dst;
     cryp.iv = iv;

     ioctl(cfd, CIOCCRYPT, &cryp);
}
```

```
void encrypt(int cfd, unsigned int ses, unsigned char *
   src, unsigned char *dst, unsigned char *iv)
{
    auxcrypt(cfd, ses, COP_ENCRYPT, src, dst, iv);
}

void decrypt(int cfd, unsigned int ses, unsigned char *
   src, unsigned char *dst, unsigned char *iv)
{
    auxcrypt(cfd, ses, COP_DECRYPT, src, dst, iv);
}

void beginSession(int cfd, struct session_op *sess)
{
    memset(sess, 0, sizeof(*sess));
    sess->cipher = CRYPTO_AES_CBC;
    sess->keylen = KEY_SIZE;
    sess->key = KEY;
    ioctl(cfd, CIOCGSESSION, sess);
}

void endSession(int cfd, struct session_op sess)
{
    ioctl(cfd, CIOCFSESSION, &sess.ses);
}
```

2.1.2 mysocket-common.h

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```c
#include <arpa/inet.h>
#include <netinet/in.h>

#include "mycrypto.h"

#define TCP_PORT 35001

void insist_read(int fd, void *buf, size_t nbytes)
{
    memset(buf, 0, BUF_SIZE);
    while (nbytes > 0)
    {
        ssize_t ret = read(fd, buf, nbytes);
        if (ret == 0)
            exit(0);
        if (strchr(buf, '\n'))
            break;
        buf += ret;
        nbytes -= ret;
    }
}

void insist_write(int fd, void *buf, size_t n)
{
    while (n > 0)
    {
        ssize_t ret = write(fd, buf, n);
        buf += ret;
        n -= ret;
    }
    memset(buf, 0, BUF_SIZE);
}

void print(unsigned char *array, ssize_t len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("%c", array[i]);
    printf("\n");
}
```

```
void mywrite(int sd, int cfd, struct session_op sess,
   unsigned char *buf, unsigned char *auxcrypted)
{
    printf("me:    ");
    fflush(0);
    insist_read(1, buf, DATA_SIZE);
    if (!strcmp((const char *)buf, "bye\n"))
    {
        shutdown(sd, SHUT_WR);
        exit(0);
    }
    encrypt(cfd, sess.ses, buf, auxcrypted, IV);
    auxcrypted[BUF_SIZE - 1] = '\n';
    insist_write(sd, auxcrypted, BUF_SIZE);
}

void myread(int sd, int cfd, struct session_op sess,
   unsigned char *buf, unsigned char *auxcrypted)
{
    insist_read(sd, buf, BUF_SIZE);
    decrypt(cfd, sess.ses, buf, auxcrypted, IV);
    printf("other: ");
    print(auxcrypted, (ssize_t)(strchr((char *)
       auxcrypted, '\n') - (char *)auxcrypted));
    fflush(0);
}
```

2.1.3 mysocket-client.c

```
#include "mysocket-common.h"

int main(int argc, char **argv)
{
    char *hostname = "localhost";

    int sd = socket(PF_INET, SOCK_STREAM, 0);

    struct hostent *hp = gethostbyname(hostname);

    struct sockaddr_in sa;
```

```
    sa.sin_family = AF_INET;
    sa.sin_port = htons(TCP_PORT);
    memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(
       struct in_addr));
    connect(sd, (struct sockaddr *)&sa, sizeof(sa));
    printf("connected\n");

    /*
       ************************************************************
       */

    int cfd = open(argv[1] == NULL ? "/dev/crypto" :
       argv[1], O_RDWR);

    struct session_op sess;
    beginSession(cfd, &sess);

    unsigned char buf[DATA_SIZE];
    unsigned char auxcrypted[BUF_SIZE];
    for (;;)
    {
        mywrite(sd, cfd, sess, buf, auxcrypted);
        myread(sd, cfd, sess, buf, auxcrypted);
    }

    endSession(cfd, sess);

    close(cfd);
}
```

2.1.4 mysocket-server.c

```
#include "mysocket-common.h"

int main(int argc, char **argv)
{
    signal(SIGPIPE, SIG_IGN);

    int sd = socket(PF_INET, SOCK_STREAM, 0);

    struct sockaddr_in sa;
```

```
    sa.sin_family = AF_INET;
    sa.sin_port = htons(TCP_PORT);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(sd, (struct sockaddr *)&sa, sizeof(sa));

    listen(sd, 0);

    socklen_t len = sizeof(struct sockaddr_in);
    int newsd = accept(sd, (struct sockaddr *)&sa, &len)
       ;
    printf("accepted\n");

    /*
       ************************************************************
       */

    int cfd = open(argv[1] == NULL ? "/dev/crypto" :
       argv[1], O_RDWR);

    struct session_op sess;
    beginSession(cfd, &sess);

    unsigned char buf[DATA_SIZE];
    unsigned char auxcrypted[BUF_SIZE];
    for (;;)
    {
        myread(newsd, cfd, sess, buf, auxcrypted);
        mywrite(newsd, cfd, sess, buf, auxcrypted);
    }

    endSession(cfd, sess);

    close(cfd);

    close(newsd);
}
```

2.2 VirtIO

2.2.1 crypto.h

```
#ifndef _CRYPTO_H
```

```
#define _CRYPTO_H

#define VIRTIO_CRYPTODEV_BLOCK_SIZE 16

#define VIRTIO_CRYPTODEV_SYSCALL_OPEN 0
#define VIRTIO_CRYPTODEV_SYSCALL_CLOSE 1
#define VIRTIO_CRYPTODEV_SYSCALL_IOCTL 2

/* The Virtio ID for virtio crypto ports */
#define VIRTIO_ID_CRYPTODEV 30

struct crypto_driver_data
{
    /* The list of the devices we are handling. */
    struct list_head devs;
    /* The minor number that we give to the next device.
        */
    unsigned int next_minor;
    spinlock_t lock;
};
extern struct crypto_driver_data crdrvdata;

struct crypto_device
{
    /* Next crypto device in the list, head is in the
       crdrvdata struct */
    struct list_head list;
    struct virtio_device *vdev;
    struct virtqueue *vq;
    struct semaphore vq_sem;
    unsigned int minor;
};

struct crypto_open_file
{
    struct crypto_device *crdev;
    int host_fd;
};

#endif
```

2.2.2 crypto-module.c

```
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/spinlock.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>

#include "crypto.h"
#include "crypto-chrdev.h"
#include "debug.h"

struct crypto_driver_data crdrvdata;

static void vq_has_data(struct virtqueue *vq) {}

static struct virtqueue *find_vq(struct virtio_device *
   vdev)
{
    int err;
    struct virtqueue *vq;

    vq = virtio_find_single_vq(vdev, vq_has_data, "
       crypto-vq");
    if (IS_ERR(vq))
    {
        debug("Could not find vq");
        vq = NULL;
    }

    return vq;
}

// This function is called each time the kernel finds a
   virtio device that we are associated with.
static int virtcons_probe(struct virtio_device *vdev)
{
    int ret = 0;
    struct crypto_device *crdev;
```

```
    crdev = kzalloc(sizeof(*crdev), GFP_KERNEL);
    if (!crdev)
    {
        ret = -ENOMEM;
        goto out;
    }

    crdev->vdev = vdev;
    vdev->priv = crdev;

    crdev->vq = find_vq(vdev);
    if (!crdev->vq)
    {
        ret = -ENXIO;
        goto out;
    }

    sema_init(&crdev->vq_sem, 1);

    spin_lock_irq(&crdrvdata.lock);
    crdev->minor = crdrvdata.next_minor++;
    list_add_tail(&crdev->list, &crdrvdata.devs);
    spin_unlock_irq(&crdrvdata.lock);
    debug("Got minor = %u", crdev->minor);

out:
    return ret;
}

static void virtcons_remove(struct virtio_device *vdev)
{
    struct crypto_device *crdev = vdev->priv;

    spin_lock_irq(&crdrvdata.lock);
    list_del(&crdev->list);
    spin_unlock_irq(&crdrvdata.lock);

    vdev->config->reset(vdev);
    vdev->config->del_vqs(vdev);
```

```
    kfree(crdev);
}

static struct virtio_device_id id_table[] = {
    {VIRTIO_ID_CRYPTODEV, VIRTIO_DEV_ANY_ID},
    {0},
};

static unsigned int features[] = {
    0};

static struct virtio_driver virtio_crypto = {
    .feature_table = features,
    .feature_table_size = ARRAY_SIZE(features),
    .driver.name = KBUILD_MODNAME,
    .driver.owner = THIS_MODULE,
    .id_table = id_table,
    .probe = virtcons_probe,
    .remove = virtcons_remove,
};

// The function that is called when our module is being
   inserted in the running kernel.
static int __init init(void)
{
    int ret;

    ret = crypto_chrdev_init();
    if (ret < 0)
    {
        printk(KERN_ALERT "Could not initialize
           character devices.\n");
        return ret;
    }

    INIT_LIST_HEAD(&crdrvdata.devs);
    spin_lock_init(&crdrvdata.lock);

    ret = register_virtio_driver(&virtio_crypto);
    if (ret < 0)
```

```
    {
        printk(KERN_ALERT "Failed to register virtio
           driver.\n");
        crypto_chrdev_destroy();
        return ret;
    }

    return ret;
}

// The function that is called when our module is being
   removed.
static void __exit fini(void)
{
    crypto_chrdev_destroy();
    unregister_virtio_driver(&virtio_crypto);
}

module_init(init);
module_exit(fini);

MODULE_DEVICE_TABLE(virtio, id_table);
MODULE_DESCRIPTION("Virtio crypto driver");
MODULE_LICENSE("GPL");
```

2.2.3 crypto-chrdev.c

```
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/wait.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>

#include "crypto.h"
#include "crypto-chrdev.h"
#include "debug.h"
#include "cryptodev.h"

struct cdev crypto_chrdev_cdev;
```

```
static struct crypto_device *get_crypto_dev_by_minor(
   unsigned int minor)
{
    struct crypto_device *crdev;
    unsigned long flags;

    spin_lock_irqsave(&crdrvdata.lock, flags);
    list_for_each_entry(crdev, &crdrvdata.devs, list)
    {
        if (crdev->minor == minor)
            goto out;
    }
    crdev = NULL;

out:
    spin_unlock_irqrestore(&crdrvdata.lock, flags);

    return crdev;
}

static int crypto_chrdev_open(struct inode *inode,
   struct file *filp)
{
    int ret;
    struct crypto_open_file *crof;
    struct crypto_device *crdev;
    struct virtqueue *vq;
    struct scatterlist syscall_type_sg, host_fd_sg, *sgs
       [2];
    unsigned int num_out = 0, num_in = 0, len;
    unsigned int *syscall_type;
    int *host_fd;

    syscall_type = kzalloc(sizeof(*syscall_type),
       GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_OPEN;
    host_fd = kzalloc(sizeof(*host_fd), GFP_KERNEL);
    *host_fd = -1;
```

```
ret = nonseekable_open(inode, filp);
if (ret < 0)
    goto fail;

crdev = get_crypto_dev_by_minor(iminor(inode));
if (!crdev)
{
    debug("Could not find crypto device with %u
       minor", iminor(inode));
    ret = -ENODEV;
    goto fail;
}
vq = crdev->vq;

crof = kzalloc(sizeof(*crof), GFP_KERNEL);
if (!crof)
{
    ret = -ENOMEM;
    goto fail;
}
crof->crdev = crdev;
crof->host_fd = -1;
filp->private_data = crof;

sg_init_one(&syscall_type_sg, syscall_type, sizeof(*
   syscall_type));
sgs[num_out++] = &syscall_type_sg;
sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
sgs[num_out + num_in++] = &host_fd_sg;

down(&crdev->vq_sem);
virtqueue_add_sgs(vq, sgs, num_out, num_in, &
   syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(vq);
while (virtqueue_get_buf(vq, &len) == NULL)
    ;
up(&crdev->vq_sem);
crof->host_fd = *host_fd;

if (*host_fd < 0)
```

```
    {
        debug("host failed to open()");
        ret = -ENODEV;
    }

fail:
    kfree(syscall_type);
    kfree(host_fd);

    return ret;
}

static int crypto_chrdev_release(struct inode *inode,
   struct file *filp)
{
    int ret = 0;
    struct crypto_open_file *crof = filp->private_data;
    struct crypto_device *crdev = crof->crdev;
    struct virtqueue *vq = crdev->vq;
    struct scatterlist syscall_type_sg, host_fd_sg, *sgs
       [2];
    unsigned int num_out = 0, num_in = 0, len;
    unsigned int *syscall_type;

    syscall_type = kzalloc(sizeof(*syscall_type),
       GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_CLOSE;

    sg_init_one(&syscall_type_sg, syscall_type, sizeof(*
       syscall_type));
    sgs[num_out++] = &syscall_type_sg;
    sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof
       ->host_fd));
    sgs[num_out++] = &host_fd_sg;

    down(&crdev->vq_sem);
    virtqueue_add_sgs(vq, sgs, num_out, num_in, &
       syscall_type_sg, GFP_ATOMIC);
    virtqueue_kick(vq);
    while (virtqueue_get_buf(vq, &len) == NULL)
```

```
        ;
    up(&crdev->vq_sem);

    kfree(syscall_type);
    kfree(crof);
    return ret;
}

static long crypto_chrdev_ioctl(struct file *filp,
   unsigned int cmd, unsigned long arg)
{
    long ret = 0;
    struct crypto_open_file *crof = filp->private_data;
    struct crypto_device *crdev = crof->crdev;
    struct virtqueue *vq = crdev->vq;
    struct scatterlist syscall_type_sg, host_fd_sg,
       ioctl_cmd_sg, session_key_sg, session_op_sg,
       host_return_val_sg, ses_id_sg, crypt_op_sg,
       src_sg, iv_sg, dst_sg, *sgs[8];
    unsigned int num_out = 0, num_in = 0, len;
    unsigned int *syscall_type, *ioctl_cmd, *ses_id;
    int *host_return_val;
    unsigned char *session_key = NULL, *src, *iv, *dst =
        NULL;
    struct session_op *session_op = NULL;
    struct crypt_op *crypt_op = NULL;

    syscall_type = kzalloc(sizeof(*syscall_type),
       GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_IOCTL;
    ioctl_cmd = kzalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
    *ioctl_cmd = cmd;
    host_return_val = kzalloc(sizeof(*host_return_val),
       GFP_KERNEL);

    sg_init_one(&syscall_type_sg, syscall_type, sizeof(*
       syscall_type));
    sgs[num_out++] = &syscall_type_sg;
    sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof
       ->host_fd));
```

```
sgs[num_out++] = &host_fd_sg;
sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(cmd));
sgs[num_out++] = &ioctl_cmd_sg;

switch (cmd)
{
case CIOCGSESSION:
    debug("CIOCGSESSION");
    session_op = (struct session_op *)kzalloc(sizeof
       (*session_op), GFP_KERNEL);
    if (copy_from_user(session_op, (void *)arg,
       sizeof(*session_op)))
    {
        debug("copy_from_user");
        ret = -EFAULT;
        goto fail;
    }
    session_key = session_op->key;
    session_op->key = (unsigned char *)kzalloc(
       session_op->keylen * sizeof(*session_op->key)
       , GFP_KERNEL);
    if (copy_from_user(session_op->key, session_key,
        session_op->keylen * sizeof(*session_key)))
    {
        debug("Copy from user failed!");
        ret = -EFAULT;
        return ret;
    }

    sg_init_one(&session_key_sg, session_op->key,
       session_op->keylen * sizeof(*session_op->key)
       );
    sgs[num_out++] = &session_key_sg;
    sg_init_one(&session_op_sg, session_op, sizeof(*
       session_op));
    sgs[num_out + num_in++] = &session_op_sg;
    sg_init_one(&host_return_val_sg, host_return_val
       , sizeof(*host_return_val));
    sgs[num_out + num_in++] = &host_return_val_sg;
```

```
        break;

    case CIOCFSESSION:
        debug("CIOCFSESSION");
        ses_id = (unsigned int *)kzalloc(sizeof(*ses_id)
            , GFP_KERNEL);
        if (copy_from_user(ses_id, (void *)arg, sizeof(*
            ses_id)))
        {
            debug("copy_from_user");
            ret = -EFAULT;
            goto fail;
        }

        sg_init_one(&ses_id_sg, ses_id, sizeof(*ses_id))
            ;
        sgs[num_out++] = &ses_id_sg;
        sg_init_one(&host_return_val_sg, host_return_val
            , sizeof(*host_return_val));
        sgs[num_out + num_in++] = &host_return_val_sg;
        break;

    case CIOCCRYPT:
        debug("CIOCCRYPT");
        crypt_op = (struct crypt_op *)kzalloc(sizeof(*
            crypt_op), GFP_KERNEL);
        if (copy_from_user(crypt_op, (void *)arg, sizeof
            (*crypt_op)))
        {
            debug("copy_from_user");
            ret = -EFAULT;
            goto fail;
        }
        src = crypt_op->src;
        crypt_op->src = (unsigned char *)kzalloc(
            crypt_op->len * sizeof(*crypt_op->src),
            GFP_KERNEL);
        if (copy_from_user(crypt_op->src, src, crypt_op
            ->len * sizeof(*crypt_op->src)))
        {
```

```
    debug("copy_from_user");
    ret = -EFAULT;
    goto fail;
}
iv = crypt_op->iv;
crypt_op->iv = (unsigned char *)kzalloc(
   VIRTIO_CRYPTODEV_BLOCK_SIZE * sizeof(*
   crypt_op->iv), GFP_KERNEL);
if (copy_from_user(crypt_op->iv, iv,
   VIRTIO_CRYPTODEV_BLOCK_SIZE * sizeof(*
   crypt_op->iv)))
{
    debug("copy_from_user");
    ret = -EFAULT;
    goto fail;
}
crypt_op->dst = (unsigned char *)kzalloc(
   crypt_op->len * sizeof(*crypt_op->dst),
   GFP_KERNEL);
dst = crypt_op->dst;

sg_init_one(&crypt_op_sg, crypt_op, sizeof(
   struct crypt_op));
sgs[num_out++] = &crypt_op_sg;
sg_init_one(&src_sg, crypt_op->src, crypt_op->
   len * sizeof(*crypt_op->src));
sgs[num_out++] = &src_sg;
sg_init_one(&iv_sg, crypt_op->iv,
   VIRTIO_CRYPTODEV_BLOCK_SIZE * sizeof(*
   crypt_op->iv));
sgs[num_out++] = &iv_sg;
sg_init_one(&dst_sg, dst, crypt_op->len * sizeof
   (*dst));
sgs[num_out + num_in++] = &dst_sg;
sg_init_one(&host_return_val_sg, host_return_val
   , sizeof(*host_return_val));
sgs[num_out + num_in++] = &host_return_val_sg;

break;
```

```
default:
    debug("Unsupported ioctl command");

    break;
}

down(&crdev->vq_sem);
virtqueue_add_sgs(vq, sgs, num_out, num_in, &
   syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(vq);
while (virtqueue_get_buf(vq, &len) == NULL)
    ;
up(&crdev->vq_sem);

switch (cmd)
{
case CIOCGSESSION:
    if (copy_to_user((void *)arg, session_op, sizeof
       (*session_op)))
    {
        debug("copy_to_user");
        ret = -EFAULT;
        goto fail;
    }
    ((struct session_op *)arg)->key = session_key;

    break;

case CIOCCRYPT:
    if (copy_to_user(((struct crypt_op *)arg)->dst,
       dst, crypt_op->len * sizeof(*crypt_op->dst)))
    {
        debug("copy_to_user");
        ret = -EFAULT;
        goto fail;
    }

    break;
}
```

```
    ret = *host_return_val;

fail:
    kfree(syscall_type);
    kfree(ioctl_cmd);

    return ret;
}

static ssize_t crypto_chrdev_read(struct file *filp,
   char __user *usrbuf, size_t cnt, loff_t *f_pos)
{
    return -EINVAL;
}

static struct file_operations crypto_chrdev_fops =
    {
        .owner = THIS_MODULE,
        .open = crypto_chrdev_open,
        .release = crypto_chrdev_release,
        .read = crypto_chrdev_read,
        .unlocked_ioctl = crypto_chrdev_ioctl,
};

int crypto_chrdev_init()
{
    int ret;
    dev_t dev_no;
    unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

    debug("Initializing character device...");
    cdev_init(&crypto_chrdev_cdev, &crypto_chrdev_fops);
    crypto_chrdev_cdev.owner = THIS_MODULE;

    dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
    ret = register_chrdev_region(dev_no,
       crypto_minor_cnt, "crypto_devs");
    if (ret < 0)
    {
        debug("failed to register region, ret = %d", ret
```

```
            );
        return ret;
    }
    ret = cdev_add(&crypto_chrdev_cdev, dev_no,
       crypto_minor_cnt);
    if (ret < 0)
    {
        debug("failed to add character device");
        unregister_chrdev_region(dev_no,
           crypto_minor_cnt);
        return ret;
    }

    debug("Completed successfully");
    return 0;
}

void crypto_chrdev_destroy()
{
    dev_t dev_no;
    unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

    dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
    cdev_del(&crypto_chrdev_cdev);
    unregister_chrdev_region(dev_no, crypto_minor_cnt);
}
```

2.2.4 virtio-cryptodev.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <crypto/cryptodev.h>

#include "qemu/osdep.h"
#include "qemu/iov.h"
#include "hw/qdev.h"
#include "hw/virtio/virtio.h"
#include "standard-headers/linux/virtio_ids.h"
#include "hw/virtio/virtio-cryptodev.h"
```

```
static uint64_t get_features(VirtIODevice *vdev,
   uint64_t features, Error **errp)
{
    DEBUG_IN();
    return features;
}

static void get_config(VirtIODevice *vdev, uint8_t *
   config_data)
{
    DEBUG_IN();
}

static void set_config(VirtIODevice *vdev, const uint8_t
    *config_data)
{
    DEBUG_IN();
}

static void set_status(VirtIODevice *vdev, uint8_t
   status)
{
    DEBUG_IN();
}

static void vser_reset(VirtIODevice *vdev)
{
    DEBUG_IN();
}

static void vq_handle_output(VirtIODevice *vdev,
   VirtQueue *vq)
{
    VirtQueueElement *elem;
    unsigned int *syscall_type, *ioctl_cmd, *ses_id;
    int *host_fd;
    struct session_op *session_op;
    int *host_return_val;
    struct crypt_op *crypt_op;
```

```
DEBUG_IN();

elem = virtqueue_pop(vq, sizeof(VirtQueueElement));
if (!elem)
{
    DEBUG("No item to pop from VQ :(");
    return;
}

DEBUG("I have got an item from VQ :)");

syscall_type = elem->out_sg[0].iov_base;
switch (*syscall_type)
{
case VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN");
    host_fd = elem->in_sg[0].iov_base;
    *host_fd = open("/dev/crypto", O_RDWR);
    break;

case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
    host_fd = elem->out_sg[1].iov_base;
    if (close(*host_fd) < 0)
        DEBUG("close");
    break;

case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
    host_fd = elem->out_sg[1].iov_base;
    ioctl_cmd = elem->out_sg[2].iov_base;

    switch (*ioctl_cmd)
    {
    case CIOCGSESSION:
        DEBUG("CIOCGSESSION");
        session_op = (struct session_op *)elem->
           in_sg[0].iov_base;
        session_op->key = elem->out_sg[3].iov_base;
```

```
            host_return_val = elem->in_sg[1].iov_base;
            *host_return_val = ioctl(*host_fd,
               CIOCGSESSION, session_op);
            break;

        case CIOCFSESSION:
            DEBUG("CIOCFSESSION");
            ses_id = (unsigned int *)elem->out_sg[3].
               iov_base;
            host_return_val = elem->in_sg[0].iov_base;
            *host_return_val = ioctl(*host_fd,
               CIOCFSESSION, ses_id);
            break;

        case CIOCCRYPT:
            DEBUG("CIOCCRYPT");
            crypt_op = (struct crypt_op *)elem->out_sg
               [3].iov_base;
            crypt_op->src = elem->out_sg[4].iov_base;
            crypt_op->iv = elem->out_sg[5].iov_base;
            crypt_op->dst = elem->in_sg[0].iov_base;
            host_return_val = elem->in_sg[1].iov_base;
            *host_return_val = ioctl(*host_fd, CIOCCRYPT
               , crypt_op);
            break;

        default:
            DEBUG("Unknown syscall_type");
            break;
        }
    }

    virtqueue_push(vq, elem, 0);
    virtio_notify(vdev, vq);
    g_free(elem);
}

static void virtio_cryptodev_realize(DeviceState *dev,
   Error **errp)
{
```

```
    VirtIODevice *vdev = VIRTIO_DEVICE(dev);

    DEBUG_IN();

    virtio_init(vdev, "virtio-cryptodev",
       VIRTIO_ID_CRYPTODEV, 0);
    virtio_add_queue(vdev, 128, vq_handle_output);
}

static void virtio_cryptodev_unrealize(DeviceState *dev,
    Error **errp)
{
    DEBUG_IN();
}

static Property virtio_cryptodev_properties[] = {
    DEFINE_PROP_END_OF_LIST(),
};

static void virtio_cryptodev_class_init(ObjectClass *
   klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    VirtioDeviceClass *k = VIRTIO_DEVICE_CLASS(klass);

    DEBUG_IN();
    dc->props = virtio_cryptodev_properties;
    set_bit(DEVICE_CATEGORY_INPUT, dc->categories);

    k->realize = virtio_cryptodev_realize;
    k->unrealize = virtio_cryptodev_unrealize;
    k->get_features = get_features;
    k->get_config = get_config;
    k->set_config = set_config;
    k->set_status = set_status;
    k->reset = vser_reset;
}

static const TypeInfo virtio_cryptodev_info = {
    .name = TYPE_VIRTIO_CRYPTODEV,
```

```
    .parent = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtCryptodev),
    .class_init = virtio_cryptodev_class_init,
};

static void virtio_cryptodev_register_types(void)
{
    type_register_static(&virtio_cryptodev_info);
}

type_init(virtio_cryptodev_register_types)
```