Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
7ο εξάμηνο, Ακαδημαϊκή περίοδος 2021 - 2022
http://www.cslab.ece.ntua.gr/

Εργαστήριο Λειτουργικών Συστημάτων

Ομάδα 20

Μέλη Ομάδας: Βρεττός Βασίλειος el18126
Τσάφος Αλέξανδρος el18211

3η Εργαστηριακή Άσκηση: VirtIO - Cryptodev

Σκοπός της άσκησης ήταν η υλοποίηση:

1. Μιας απλής υπηρεσίας chat μεταξύ 2 ατόμων (server και client) με απλό σύστημα socket χρησιμοποιώντας TCP/IP.
2. Συστήματος κρυπτογράφησης της παραπάνω υπηρεσίας με χρήση του Cryptodev character device.
3. Επέκταση του συστήματος κρυπτογράφησης σε εικονική συσκευή προτύπου VirtIO για το περιβάλλον QEMU-KVM.

Η υπηρεσία που έτρεχε μέσα στο QEMU (VM) είχε πρόσβαση στην συσκευή /dev/cryptodev του host μηνύματος και μπορούσε να κρυπτογραφησει τα μηνύματα εντός του VM μέσω εικονικών συσκευών cryptodev.

1. Υπηρεσία chat με χρήση BSD Sockets τύπου TCP/IP

Ο σκελετός της υπηρεσίας chat δινόταν έτοιμος.

Οι δικές μας προσθήκες ήταν στο infinite loop κομμάτι του κώδικα που φαίνεται η επιλογή των **socket_fd** (SD) με την χρήση της **select()**. Ουσιαστικά, επιλέγουμε κατάλληλα το socket επικοινωνίας αναλόγως με το εάν θέλουμε να διαβάσουμε αυτά που έστειλε το απέναντι μέρος επικοινωνίας ή εάν θελήσουμε να στείλουμε εμείς μήνυμα στο απέναντι μέρος.

Ο κώδικας που προστέθηκε παραλείπεται σε αυτό το κομμάτι για να μην υπάρχουν επαναλήψεις στην αναφορά, φαίνεται κάτω στην διαδικασία κρυπτογράφησης της υπηρεσίας.

2. Κρυπτογράφηση της υπηρεσίας chat με χρήση του Cryptodev Module

Σαν σκελετό του κρυπτογραφημένου chat είχαμε το απλό socket chat.

Το κλειδί κρυπτογράφησης είναι hard set στο πρόγραμμα και δεν παρουσιάζει ρεαλιστική υλοποίηση ενός κρυπτογραφημένου chat καθώς αποτελεί μεγάλο security flaw.

Η διαδικασία κρυπτογράφησης ξεκινάει αρχικοποιώντας το ***struct crypt_op*** **cryp** και εκτελείται η επεξεργασία με χρήση της εντολής **ioctl()** με το όρισμα **CIOCCRYPT** (και crypt.op **COP_ENCRYPT**). Το αποτέλεσμα της κρυπτογράφησης αποθηκεύεται στον ***buffer crypt.dst.*** (crypt destination). Η αντίθετη διαδικασία ξεκινάει πάλι με αρχικοποίηση του crypt session και μετά με την κλήση της **ioctl()** πάλι με το όρισμα **CIOCCRYPT** (αλλά με ***crypt.op*** **COP_DECRYPT**). Το αποτέλεσμα της αποκρυπτογράφησης αποθηκεύεται στον buffer crypt.dst.

Στο κομμάτι της επιλογής του **socket_fd** η μόνη αλλαγή είναι ότι καλούμε την **encrypt_data()** όταν θέλουμε να στείλουμε μήνυμα στο άλλο μέρος και καλούμε την **decrypt_data()** όταν είναι να λάβουμε μήνυμα από το άλλο μέρος.

crypto-chat.c - Socket Select portion

```
sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = key;

if (ioctl(crypto_fd, CIOCGSESSION, &sess)) {
	perror("Crypto session init failed");
	return 1;
}
fd_set readfds;
struct timeval timeout;
timeout.tv_sec = 0;
timeout.tv_usec = 0;
//Timeout with 0.0 timer is basically Polling

for (;;) {
	memset(buf, '\0', sizeof(buf));
	//readfds set init
	FD_ZERO(&readfds);

	//add the conflicting fd's
	FD_SET(0, &readfds);
	FD_SET(sd, &readfds);

	//nfds = max_fd + 1
	select(sd+1, &readfds, NULL, NULL, &timeout);
```

```
	//If something is ready to be read from socket_fd
	if (FD_ISSET(sd, &readfds)){
		n = read(sd, buf, sizeof(buf));

		if (n < 0) {
			perror("Client read from peer");
			exit(1);
		}

		if (n <= 0)
			break;

		//decrypt data that was received from socket <-- server
		decrypt_data(crypto_fd);
		if (insist_write(1, buf, sizeof(buf)) != sizeof(buf)) {
			perror("Client wrote to stdout");
			exit(1);
		}
	}
	//If something is ready to be sent through socket --> server
	if (FD_ISSET(0, &readfds)){
		n = read(0, buf, sizeof(buf));

		if (n < 0) {
			perror("Client read from itself");
			exit(1);
		}

		if (n <= 0)
			break;

		//encrypt data that will be sent through socket --> server
		encrypt_data(crypto_fd);
		if (insist_write(sd, buf, sizeof(buf)) != sizeof(buf)) {
			perror("Client wrote to peer");
			exit(1);
		}
	}
}
fprintf(stderr, "\nDone.\n");

if (ioctl(crypto_fd, CIOCFSESSION, &sess.ses)) { //Send IO command to end session
	perror("Error on ending Crypto Session");
	exit(1);
}

if (close(crypto_fd) < 0){ //close Cryptodev module
```

```
        perror("Erron on closing Cryptodev module FD");
        exit(1);
}

return 0;
```

crypto-server.c - Socket Select portion

```
sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = key;

if (ioctl(crypto_fd, CIOCGSESSION, &sess)) {
        perror("Crypto session init failed");
        return 1;
}
if (listen(sd, TCP_BACKLOG) < 0) {
        perror("listen");
        exit(1);
}

/* Loop forever, accept()ing connections */
for (;;) {
        fprintf(stderr, "Waiting for an incoming connection...\n");

        /* Accept an incoming connection */
        len = sizeof(struct sockaddr_in);
        if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
                perror("accept");
                exit(1);
        }
        //convert IPv4 and IPv6 addresses from binary to text form
        if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr))) {
                perror("could not format IP address");
                exit(1);
        }
        fprintf(stderr, "Incoming connection from %s:%d\n",
        addrstr, ntohs(sa.sin_port));

        fd_set readfds;
        struct timeval timeout;
        timeout.tv_sec = 0;
        timeout.tv_usec = 0;

        /* We break out of the loop when the remote peer goes away */
```

```
	for (;;) {
		memset(buf, '\0', sizeof(buf));

		//readfds set init
		FD_ZERO(&readfds);

		//add the conflicting fd's
		FD_SET(0, &readfds);
		FD_SET(newsd, &readfds);

		//see which fd has something to say
		select(newsd+1, &readfds, NULL, NULL, &timeout);

		if (FD_ISSET(newsd, &readfds)){
			n = read(newsd, buf, sizeof(buf));
			if (n <= 0) {
				if (n < 0)
				perror("read from remote peer failed");
				else
				fprintf(stderr, "Peer went away\n");
				break;
			}
			//	toupper_buf(buf, n);
			decrypt_data(crypto_fd);
			if (insist_write(1, buf, sizeof(buf)) != sizeof(buf)) {
				perror("write to remote peer failed");
				break;
			}
		}
		if (FD_ISSET(0, &readfds)){

			n = read(0, buf, sizeof(buf));
			if (n <= 0) {
				if (n < 0)
				perror("read from remote peer failed");
				else
				fprintf(stderr, "Peer went away\n");
				break;
			}

			encrypt_data(crypto_fd);

			if (insist_write(newsd, buf, sizeof(buf)) != sizeof(buf)) {
				perror("write to remote peer failed");
				break;
			}
		}
```

```
        }
        /* Make sure we don't leak open files */
        if (close(newsd) < 0)
        perror("close");
}

if (ioctl(crypto_fd, CIOCFSESSION, &sess.ses)) {
        perror("Error on ending Crypto Session");
        exit(1);
}
if (close(crypto_fd) < 0){
        perror("Erron on closing Cryptodev module FD");
        exit(1);
}
/* This will never happen */
return 1;
```

Encryption/Decryption Functions

```
int encrypt_data(int cfd)
{
        int i;
        struct crypt_op cryp;
        struct {
                unsigned char       in[DATA_SIZE],
                encrypted[DATA_SIZE];
        } data;


memset(&cryp, 0, sizeof(cryp));

//Initialize crypto struct with crypto session info
cryp.ses = sess.ses;
cryp.len = sizeof(buf);
cryp.src = buf;
cryp.dst = data.encrypted;
cryp.iv = iv;
cryp.op = COP_ENCRYPT;

//send IO command to encrypt from buf to data.encrypted
if (ioctl(cfd, CIOCCRYPT, &cryp)) {
        perror("Error encrypting");
        return 1;
}
//for buffer-size fill it with encrypted data
```

```
for (i = 0; i < sizeof(buf); i++){
	buf[i] = data.encrypted[i];
}
//Function returns 0 on success
return 0;
}


int decrypt_data(int cfd)
{
	int i;
	struct crypt_op cryp;
	struct {
		unsigned char	in[DATA_SIZE],
		decrypted[DATA_SIZE];
	} data;

memset(&cryp, 0, sizeof(cryp));
//Initialize crypto struct with crypto session info
cryp.ses = sess.ses;
cryp.len = sizeof(buf);
cryp.src = buf;
cryp.dst = data.decrypted;
cryp.iv = iv;
cryp.op = COP_DECRYPT;


//send IO command to decrypt from buf to data.decrypted
if (ioctl(cfd, CIOCCRYPT, &cryp)) {
	perror("Error decrypting");
	return 1;
}

for (i = 0; i < sizeof(buf); i++) {
	buf[i] = data.decrypted[i];
}
//Function returns 0 on success
return 0;
}
```

3. QEMU Cryptodev-VirtIO module

Πλέον, έπρεπε να υλοποιηθεί μια εικονική συσκευή αρχιτεκτονικής ***PCI*** και προτύπου ***VirtIO*** η οποία θα εξυπηρετεί τα αντίστοιχα ***system calls*** που χρησιμοποιούμε στην υπηρεσία chat μας.

Δηλαδή, πρέπει να υλοποιήσουμε τα system calls **open(), ioctl(), release()** (και την **init()** αλλά είναι ήδη έτοιμη).

Frontend:

crypto_chrdev_open():

Καλούμε από το user space την **open*()*** για να ανοίξουμε το εικονικό αρχείο μιας συσκευής ***cryptodev***. Τότε, αναγνωρίζοντας το ***major number*** των crypto devices, το kernel εκτελεί την **crypto_chrdev_open()** η οποία ξεκινάει λαμβάνοντας το ***minor number*** της συσκευής που θέλουμε να ανοίξουμε, συνεχίζει στέλνοντας το ***host_fd*** και τον τύπο του ***syscall*** στο ***backend*** κομμάτι (QEMU) το οποίο αναλαμβάνει την εκτέλεση της **open()**. Η αποστολή στο ***backend*** γίνεται μέσω "***scatterlists***" (**sg_init_one()** για όσα δεδομένα στέλνουμε και μετά **virtqueue_add_sgs()**) και η ενημέρωση ότι υπάρχουν νέα δεδομένα γίνεται με την συνάρτηση **virtqueue_kick()**. Αφού επιστραφεί ο ***host_fd*** από το ***backend***, ο πυρήνας του VM μπορεί πλέον να επικοινωνεί με την συσκευή ***cryptodev*** του host μέσω του ***host_fd***.

```
static int crypto_chrdev_open(struct inode *inode, struct file *filp)
{
	int ret = 0;
	int err;
	unsigned int len;
	struct crypto_open_file *crof;
	struct crypto_device *crdev;
	unsigned int *syscall_type;
	int *host_fd;
	struct scatterlist sgs_syscall_type, sgs_host_fd, *sgs[2];

	debug("Entering");

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

	/* Associate this open file with the relevant crypto device. */
	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;
	}
```

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


	/**
	* We need two sg lists, one for syscall_type and one to get the
	* file descriptor from the host.
	**/


	sg_init_one(&sgs_syscall_type, syscall_type, sizeof(syscall_type));
	sgs[0] = &sgs_syscall_type;
	sg_init_one(&sgs_host_fd, host_fd, sizeof(host_fd));
	sgs[1] = &sgs_host_fd;

	/**
	* Wait for the host to process our data.
	**/
	/* ?? */
	if (down_interruptible(&crdev->lock)){
		return -ERESTARTSYS;
	}

	virtqueue_add_sgs(crdev->vq, sgs, 1, 1, &sgs_syscall_type, GFP_ATOMIC); //
TO ADD: TOKEN BUFFER
	debug("open: sent sgs to backend");
	virtqueue_kick(crdev->vq);
	debug("open: notified host that new data has been sent (KICK)");

	while(!virtqueue_get_buf(crdev->vq, &len));

	up(&crdev->lock);

	crof->host_fd = *host_fd;

	//kaneis wait_event_interruptible() -> kaneis add sto queue -> kaneis kick


	/* If host failed to open() return -ENODEV. */
```

```
	if (crof->host_fd < 0){
		debug("Failed to open crypto device");
		ret = -ENODEV;
	}

	debug("Crypto device opened successfully");

	fail:
	debug("Leaving");
	return ret;
}
```

crypto_chrdev_release():

Η **crypto_chrdev_release()** δουλεύει με τον ίδιο τρόπο αλλά αντί να καλείται η **open()** στο ***backend*** κομμάτι, καλείται η **close()**. Η επικοινωνία μεταξύ ***frontend*** και ***backend*** γίνεται πάλι μέσω "***scatterlist***".

```
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;
	unsigned int *syscall_type;
	//int *host_fd;
	unsigned int len;
	struct scatterlist sgs_syscall_type, sgs_host_fd, *sgs[2];

	debug("Entering crypto-release");

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

	sg_init_one(&sgs_syscall_type, syscall_type, sizeof(syscall_type));
	sgs[0] = &sgs_syscall_type;
	sg_init_one(&sgs_host_fd, &crof->host_fd, sizeof(crof->host_fd));
	sgs[1] = &sgs_host_fd;

	/**
	* Wait for the host to process our data.
	**/

	if (down_interruptible(&crdev->lock)){
		return -ERESTARTSYS;
	}
```

```
        virtqueue_add_sgs(crdev->vq, sgs, 2, 0, &sgs_syscall_type, GFP_ATOMIC);
        debug("open: sent sgs to backend");
        virtqueue_kick(crdev->vq);
        debug("open: notified host that new data has been sent (KICK)");

        while(!virtqueue_get_buf(crdev->vq, &len));

        up(&crdev->lock);

        debug("Succesfully close cryptodev file");

        kfree(crof);
        debug("Leaving");
        return ret;

}
```

crypto_chrdev_ioctl():

Η **ioctl()** στο μέρος του ***frontend*** πρέπει να λάβει από το ***userspace*** όλα τα απαραίτητα δεδομένα ώστε να εκτελεστεί στο μέρος του ***backend***. Τα δεδομένα αυτά (**arg, sess_key, crypt->src, crypt->iv**, κ,λ.π.) λαμβάνονται από το ***userspace*** στο ***kernelspace*** του ***VM*** με χρήση της **copy_from_user()** (αφού πρώτα γίνει allocated σωστά μνήμη με χρήση της **kzalloc()**).

Με χρήση switch-case ορίζουμε αναλόγως την **ioctl()** (**CIOCGSESSION**, **CIOCCRYPT**, **CIOCFSESSION**) και στέλνουμε τα αντίστοιχα δεδομένα στο ***backend***. Προσέχουμε ιδιαίτερα ότι η **copy_from_user()** πρέπει να εκτελεστεί για τα ***structs*** που χρησιμοποιούμε (**session_op** και **crypt_op**) αλλά και για τα περιεχόμενα τους τα οποία επεξεργαζόμαστε.

Τα δεδομένα που χρειάζεται η **ioctl()** για να εκτελεστεί, τα “στέλνουμε” στο ***backend*** πάλι με χρήση “***scatterlist***” και ενημέρωση με **virtqueue_kick()**. Μετά την επιστροφή τους στο ***frontend***, πάλι με switch-case στέλνουμε στο ***userspace*** του VM τα δεδομένα επιστροφής της κλήσης **ioctl()** (**sess_op, sess_id, crypt->dst**). Τέλος, ελευθερώνουμε την μνήμη για τα δεδομένα που αξιοποιήσαμε με την **kfree()**.

```
static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,unsigned long arg)
{
long ret = 0;
        int err, *host_fd;
        long *host_return_val;

        struct crypto_open_file *crof = filp->private_data;
        struct crypto_device *crdev = crof->crdev;
        struct virtqueue *vq = crdev->vq;
```

```
struct scatterlist sgs_syscall_type, sgs_host_fd,
sgs_ioctl_cmd, sgs_key, sgs_sess_op, sgs_host_return_val,
sgs_sess_id, sgs_crypt_op, sgs_src, sgs_iv, sgs_dst, *sgs[8];
unsigned int num_out, num_in, len, *syscall_type, *ioctl_cmd;


//Need to be initialized to NULL or else kfree() complains.
uint32_t *sess_id = NULL;
struct crypt_op *crypt = NULL;
struct session_op *sess_op = NULL;
unsigned char *sess_key = NULL;
unsigned char *src = NULL, *dst = NULL, *iv = NULL;

debug("Entering frontend ioctl");

/**
* Allocate all data that will be sent to the host.
**/
syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
if (!syscall_type){
       ret = -ENOMEM;
       printk(KERN_ERR "Failed to allocate memory for syscall_type\n");
}
*syscall_type = VIRTIO_CRYPTODEV_SYSCALL_IOCTL;

host_fd = kzalloc(sizeof(*host_fd), GFP_KERNEL);
if (!host_fd){
       ret = -ENOMEM;
       printk(KERN_ERR "Failed to allocate memory for host_fd\n");
}
*host_fd = crof->host_fd;

ioctl_cmd = kzalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
if (!ioctl_cmd){
       ret = -ENOMEM;
       printk(KERN_ERR "Failed to allocate memory for ioctl_cmd\n");
}
//write sgs
host_return_val = kzalloc(sizeof(*host_return_val), GFP_KERNEL);
if (!host_return_val){
       ret = -ENOMEM;
       printk(KERN_ERR "Failed to allocate memory for host_return_val\n");
}

num_out = 0;
num_in = 0;

/**
*  These are common to all ioctl commands.
**/
sg_init_one(&sgs_syscall_type, syscall_type, sizeof(*syscall_type));
sgs[num_out++] = &sgs_syscall_type;
```

```
        sg_init_one(&sgs_host_fd, host_fd, sizeof(host_fd));
        sgs[num_out++] = &sgs_host_fd;

        /**
        *  Add all the cmd specific sg lists.
        **/
        switch (cmd) {
                case CIOCGSESSION:
                debug("frontend CIOCGSESSION");
                *ioctl_cmd = VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION;
                sg_init_one(&sgs_ioctl_cmd, ioctl_cmd, sizeof(ioctl_cmd));
                sgs[num_out++] = &sgs_ioctl_cmd;
                debug("ioctl scatterlist init successful");

                //arg will contain the session we want to open
                sess_op = kzalloc(sizeof(sess_op), GFP_KERNEL);
                if (!sess_op){
                        ret = -ENOMEM;
                        printk(KERN_ERR "Failed to allocate memory for session_op\n");
                        goto fail;
                }

                if (copy_from_user(sess_op, (struct session_op*)arg, sizeof(*sess_op)))
                {
                        debug("CIOCGSESSION: copy_from_user failed (session)");
                        ret = -EFAULT;
                        goto fail;
                }

                sess_key = kzalloc(sess_op->keylen*sizeof(char), GFP_KERNEL);
                if (!sess_key){
                        ret = -ENOMEM;
                        printk(KERN_ERR "Failed to allocate memory for sess_key\n");
                }

                if (copy_from_user(sess_key, sess_op->key, sess_op->keylen*sizeof(char)))
                {
                        debug("CIOCGSESSION: copy_from_user failed (session key)");
                        ret = -EFAULT;
                        goto fail;
                }
                //unsigned char session_key -> Read Flag

                sg_init_one(&sgs_key, sess_key, sizeof(sess_key));
                sgs[num_out++] = &sgs_key;
                debug("sess key scatterlist init successful");
                //struct session_op session_op -> Write Flag
                sg_init_one(&sgs_sess_op, sess_op, sizeof(sess_op));
                sgs[num_in++ + num_out] = &sgs_sess_op;
                debug("sess op scatterlist init successful");
                //int host_return_val -> Write Flag
                sg_init_one(&sgs_host_return_val, host_return_val, sizeof(host_return_val));
```

```
sgs[num_in++ + num_out] = &sgs_host_return_val;
debug("host_ret_val scatterlist init successful");
break;

case CIOCFSESSION:
debug("CIOCFSESSION");


*ioctl_cmd = VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION;
sg_init_one(&sgs_ioctl_cmd, ioctl_cmd, sizeof(ioctl_cmd));
sgs[num_out++] = &sgs_ioctl_cmd;

sess_id = kzalloc(sizeof(uint32_t), GFP_KERNEL);
if (!sess_id){
        ret = -ENOMEM;
        printk(KERN_ERR "Failed to allocate memory for sess_id\n");
}
if (copy_from_user(sess_id, (uint32_t *)arg, sizeof(sess_id)))
{
        debug("CIOCFSESSION: copy_from_user failed (session ID)");
        ret = -EFAULT;
        goto fail;
}
//u32 ses id -> Read flag
sg_init_one(&sgs_sess_id, sess_id, sizeof(sess_id));
sgs[num_out++] = &sgs_sess_id;

//int host_return_val -> Write Flag
sg_init_one(&sgs_host_return_val, host_return_val, sizeof(host_return_val));
sgs[num_in++ + num_out] = &sgs_host_return_val;

break;

case CIOCCRYPT:
debug("CIOCCRYPT");


*ioctl_cmd = VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT;
sg_init_one(&sgs_ioctl_cmd, ioctl_cmd, sizeof(ioctl_cmd));
sgs[num_out++] = &sgs_ioctl_cmd;

crypt = kzalloc(sizeof(*crypt), GFP_KERNEL);
if (!crypt){
        ret = -ENOMEM;
        printk(KERN_ERR "Failed to allocate memory for crypt\n");
}
if (copy_from_user(crypt, (struct crypt_op *)arg, sizeof(*crypt)))
{
        debug("CIOCCRYPT: copy_from_user failed (crypto)");
        ret = -EFAULT;
        goto fail;
}
```

```
            sg_init_one(&sgs_crypt_op, crypt, sizeof(crypt));
            sgs[num_out++] = &sgs_crypt_op;

            src = kzalloc(sizeof(char)*crypt->len, GFP_KERNEL);
            if (!src){
                    ret = -ENOMEM;
                    printk(KERN_ERR "Failed to allocate memory for src\n");
            }
            if (copy_from_user(src, crypt->src, sizeof(char)*crypt->len))
            {
                    debug("CIOCCRYPT: copy_from_user failed (source)");
                    ret = -EFAULT;
                    goto fail;
            }
            sg_init_one(&sgs_src, src, sizeof(src));
            sgs[num_out++] = &sgs_src;

            iv = kzalloc(BLOCK_SIZE * sizeof(char), GFP_KERNEL);
            if (!iv){
                    ret = -ENOMEM;
                    printk(KERN_ERR "Failed to allocate memory for iv\n");
            }

            if (copy_from_user(iv, crypt->iv, BLOCK_SIZE * sizeof(char)))
            {
                    debug("CIOCCRYPT: copy_from_user failed (iv)");
                    ret = -EFAULT;
                    goto fail;
            }
            sg_init_one(&sgs_iv, iv, sizeof(iv));
            sgs[num_out++] = &sgs_iv;

            dst = kzalloc(crypt->len * sizeof(char), GFP_KERNEL);
            if (!dst){
                    ret = -ENOMEM;
                    printk(KERN_ERR "Failed to allocate memory for dst\n");
            }
            sg_init_one(&sgs_dst, dst, sizeof(dst));
            sgs[num_in++ + num_out] = &sgs_dst;

            //int host_return_val -> Write Flag
            sg_init_one(&sgs_host_return_val, host_return_val, sizeof(host_return_val));
            sgs[num_in++ + num_out] = &sgs_host_return_val;


            break;

            default:
            debug("Unsupported ioctl command");

            break;
    }
```

```
	/**
	* Wait for the host to process our data.
	**/
	if (down_interruptible(&crdev->lock)){
		return -ERESTARTSYS;
	}
	debug("locked before sending to backend (ioctl)");
	err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
		&sgs_syscall_type, GFP_ATOMIC);
		debug("sent queue to backend (ioctl)");
		virtqueue_kick(vq);
		debug("informed backend for change in VQ (kick_ioctl)");
		while (virtqueue_get_buf(vq, &len) == NULL)
		/* do nothing */;

		up(&crdev->lock);
		debug("unlocked (ioctl)");
		switch (cmd) {
			case CIOCGSESSION:
			debug("before copy_to_user in CIOCGSESSION ioctl");
			if (copy_to_user((struct session_op*) arg, sess_op,
sizeof(*sess_op))){
				debug("CIOCGSESSION: copy_to_user failed (session)");
				ret = -EFAULT;
				goto fail;
			}
			break;

			case CIOCFSESSION:
			if (copy_to_user((uint32_t *) arg, sess_id, sizeof(*sess_id))) {
				debug("CIOCFSESSION: copy_to_user failed (sess_id)");
				ret = -EFAULT;
				goto fail;
			}
			break;

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

			default: break;
		}
		fail:
		debug("about to free memory");
		kfree(crypt);
		kfree(dst);
```

```
                kfree(iv);
                kfree(src);
                kfree(host_return_val);
                kfree(syscall_type);
                kfree(ioctl_cmd);
                kfree(sess_key);
                kfree(sess_id);
                kfree(sess_op);


                //DO NOT kfree(host_fd);!!!!!

                debug("Leaving frontend ioctl");


                return ret;
        }
```

Backend:

Την στιγμή που εκτελείται η **virt_queue_kick()**, “αναλαμβάνει” το ***backend***. Εκτελείται η συνάρτηση **vq_handle_output()**, η οποία δεν κάνει τίποτα παραπάνω από το να διαβάζει τις ***scatterlists*** από την ουρά, και να εκτελεί την εκάστοτε εντολή. Πιο συγκεκριμένα, κάνει **virt_queue_pop()** την λίστα από την ουρά και, ανάλογα με τα περιεχόμενά της, εκτελεί την εντολή που ζήτησε το ***frontend*** και βάζει την απάντηση σε μια καινούρια ***scatterlist***, την οποία στέλνει ξανά μέσω της ***virt_queue*** με την συνάρτηση **virt_queue_push()**. Τελικά “ειδοποιεί” το ***frontend*** μέσω της **virt_queue_notify()**. Σε αυτό το σημείο, αξίζει να σημειωθεί πως στην προκειμένη υλοποίηση, η χρήση της τελευταίας συνάρτησης δεν είναι απαραίτητη, αφού το ***frontend*** ελέγχει για νέα δεδομένα στην ουρά με μέθοδο ***polling***. Για να γίνει “σωστά” η χρήση της ***virt_queue,*** στο ***frontend*** θα έπρεπε να έχουμε χρησιμοποίησει την **wait_event_interruptible()**, η οποία κοίμιζε την διεργασία του ***guest***, μέχρι ο ***host*** να την ειδοποιήσει και εκείνη να ξυπνήσει.

Backend output handler

```
static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
{
    VirtQueueElement *elem;
    unsigned int *syscall_type, *ioctl_cmd;
    int *crdev_fd;
    unsigned char *sess_key;
    long *host_ret;
    struct session_op *sess_op;
    struct crypt_op *crypt;
    uint32_t *sess_id;
    unsigned char *src, *dst, *iv;

    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");
    /* ?? */
    crdev_fd = elem->in_sg[0].iov_base;
    *crdev_fd = open("/dev/crypto", O_RDWR);
    if (*crdev_fd < 0) {
      DEBUG("Error opening cryptodev module");
    }
    printf("Successfully opened cryptodev module, fd = %d\n", *crdev_fd);
    break;

case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
    /* ?? */
    crdev_fd = elem->out_sg[1].iov_base;
    if (close(*crdev_fd) < 0) {
      DEBUG("Error closing cryptodev module");
    }
    printf("Successfully closed cryptodev module, fd = %d\n", *crdev_fd);
    break;

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

    switch (*ioctl_cmd) {
      case VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION:
        DEBUG("Entering CIOCGSESSION");
        sess_key = elem->out_sg[3].iov_base;
        sess_op = elem->in_sg[0].iov_base;
        host_ret = elem->in_sg[1].iov_base;

        sess_op->key = sess_key;
        *host_ret = ioctl(*crdev_fd, CIOCGSESSION, sess_op);
```

```
            if (*host_ret)
              perror("Crypto session init failed");

            printf("Backend sent CIOCGSESSION command successfully");
            break;
          case VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION:
            DEBUG("Entering CIOCFSESSION");
            sess_id = elem->out_sg[3].iov_base;
            host_ret = elem->in_sg[0].iov_base;
            *host_ret = ioctl(*crdev_fd, CIOCFSESSION, sess_id);
            if(*host_ret)
              perror("Ending crypto session failed");

            printf("Backend sent CIOCFSESSION command succesfully");
            break;
          case VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT:
            DEBUG("Entering CIOCCRYPT");
            crypt = elem->out_sg[3].iov_base;
            src = elem->out_sg[4].iov_base;
            iv = elem->out_sg[5].iov_base;
            dst = elem->in_sg[0].iov_base;
            host_ret = elem->in_sg[1].iov_base;

            crypt->src = src;
            crypt->dst = dst;
            crypt->iv = iv;
            *host_ret = ioctl(*crdev_fd, CIOCCRYPT, crypt);
            if(*host_ret)
              perror("Error encrypting/decrypting from cryptodev module");

            printf("Backend sent CIOCCRYPT command succesfully");
            break;

          default:
            DEBUG("Unknown ioctl command");
            break;
        }
        break;

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

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

```
    DEBUG("BACKEND PUSHED TO FRONTEND SUCCESSFULLY");
}
```