Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Λειτουργικών Συστημάτων
7ο Εξάμηνο, Ακ. Έτος 2021-2022

Γεώργιος Παπαδούλης: 03118003
Χριστίνα Προεστάκη: 03118877

Κρυπτογραφική συσκευή VirtIO για QEMU-KVM

Ζ1. Εργαλείο chat πάνω από TCP/IP sockets

Για το πρώτο ζητούμενο υλοποιήθηκε εργαλείο με χρήση BSD Sockets API που επιτρέπει την επικοινωνία πάνω από TCP/IP.
Τα sockets είναι ένα προγραμματιστικό interface ώστε ένα πρόγραμμα να καταναλώσει υπηρεσίες δικτυακών πρωτοκόλλων (TCP). Σε κάθε socket συνδέεται ένας server και clients. Για να επιτευχθεί η επικοινωνία μεταξύ του client και του server, ο καθένας οφείλει να καλέσει τα κατάλληλα system calls.

Server

- **socket(int domain, int type, int protocol);**
  Δημιουργεί το socket που θα χρησιμοποιηθεί για την επικοινωνία. Σε επιτυχή κλήση θα επιστραφεί ο fd που θα χαρακτηρίζει το socket.
  domain: οικογένεια πρωτοκόλλων επικοινωνίας - για IPv4 χρησιμοποιείται το PF_INET.
  type: στυλ επικοινωνίας. Επιλέχθηκε SOCK_STREAM για σειριακή, αξιόπιστη και αμφίδρομη μεταφορά δεδομένων.
  protocol: πρωτόκολλο προς χρήση. Επιλέχθηκε η τιμή 0, ώστε να χρησιμοποιηθεί η default τιμή.

```
/* Create TCP/IP socket, used as main chat channel */
if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
}
fprintf(stderr, "Created TCP socket\n");
```

- **bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);**
  Θέτει το socket με fd sockfd στη διεύθυνση addr.

```
        /* Bind to a well-known port */
        memset(&sa, 0, sizeof(sa)); //fills the sizeof(sa) with 0
        sa.sin_family = AF_INET; //IPv4
        sa.sin_port = htons(TCP_PORT); //htons -> convert TCP_PORT to network byte
order
        sa.sin_addr.s_addr = htonl(INADDR_ANY); //This is an IP address that is used when
we don't want to bind a socket to any specific IP.
        if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) { //αναθετει διευθυνση στο
socket
                perror("bind");
                exit(1);
```

- **listen(int sockfd, int backlog)**
  Ορίζει το socket sockfd σαν listening socket ώστε να μπορεί να δεχθεί αιτήματα σύνδεσης. Το backlog είναι το μέγιστο μήκος ουράς για αιτήσεις σύνδεσης.

```
/* Listen for incoming connections */
        if (listen(sd, TCP_BACKLOG) < 0) {
        perror("listen");
        exit(1);
        }
```

- **accept (int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen)**
  Αφαιρεί το πρώτο αίτημα για σύνδεση στο listening socket sockfd, στην περίπτωση μας το sd και δημιουργεί ένα καινούριο socket - οχι listening και επιστρέφει τον fd του, newsd, ώστε να επιτευχθεί η επικοινωνία με το client που έκανε αίτημα. Αν δεν υπάρχει αίτημα, μπλοκάρει μέχρι να κληθεί.

```
/* Accept an incoming connection */
len = sizeof(struct sockaddr_in);
if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) { //newsd -> new connected
socket
        perror("accept");
        exit(1);
}
```

- **read()** και **write()** στο newsd
- **close(int newsd)** - κλείνει το connection που έχει δημιουργηθεί με την accept.

Client

- **socket(int domain, int type, int protocol);**
  Δημιουργεί το socket που θα χρησιμοποιηθεί για την επικοινωνία.
- **connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)**
  Συνδέει το fd του socket, στην προκειμένη περίπτωση sd, στη διεύθυνση addr.

```
/* Connect to remote TCP port */
        sa.sin_family = AF_INET;
        sa.sin_port = htons(port);
        memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(struct in_addr));
        fprintf(stderr, "Connecting to remote host... "); fflush(stderr);
        if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
                perror("connect");
                exit(1);
        }
        fprintf(stderr, "Connected.\n");
```

Αξίζει να σημειωθεί η χρήση της select(). Για την ορθή αμφίδρομη επικοινωνία μέσω socket απαιτείται η ανάγνωση και η εγγραφή τόσο από το sd, τον descriptor του socket, όσο και από το standard input, ταυτόχρονα. Συγκεκριμένα, θέλουμε να μπορούν να ελεχθούν και οι δύο αυτοί file descriptors ώστε όταν είναι έτοιμοι να ανιχνευθεί και να μη χαθούν δεδομένα. Σε περίπτωση δεν μπορούσαμε να ελέγξουμε ταυτόχρονα, θα προσπαθούσαμε να διαβάσουμε από αυτόν με την read. Επειδή η read είναι blocking θα περιμέναμε μέχρι να υπάρξουν δεδομένα για ανάγνωση, ενώ ενδεχομένως να υπήρχαν έτοιμα στον άλλον fd. Για να αποφευχθεί αυτό, πρέπει να ελέγχουμε ταυτόχρονα όλους τους fds.
Ο έλεγχος αυτός πραγματοποιείται με τη βοήθεια της select(). Αναλυτικότερα, γίνεται χρήση 3 sets readfds, writefds, exceptfds, τα οποία περιέχουν τους fds που είναι έτοιμοι για ανάγνωση, εγγραφή ή κάποια ειδική περίπτωση αντίστοιχα. Όταν επιστρέψει η select(), τα sets αυτά θα περιέχουν τους κατάλληλους fds, οπότε για να ελέγξουμε την κάθε περίπτωση αρκεί να δούμε αν και σε ποιο set περιέχονται οι fds.
Στην προκειμένη περίπτωση, δημιουργούμε το readfdsset στο οποίο βάζουμε τους file descriptor για το stdin και για το socket μας, sd. Καλούμε την select και έπειτα καλούμε την FD_ISSET η οποία επιστρέφει 0 σε περίπτωση που ο fd που ελέγχουμε δεν περιέχεται στο set, ενώ μη μηδενική τιμή σε περίπτωση που περιέχεται.

```
                fd_set readfds;
```

```
int retval;
for (;;) {
        FD_ZERO(&readfds);
        FD_SET(0, &readfds);
        FD_SET(newsd, &readfds);
        retval = select(newsd+1, &readfds, NULL, NULL, NULL);
        if(retval == -1){
                perror("select()");
                break;
        }
        if(FD_ISSET(0, &readfds)){
                n = read(0, buf, sizeof(buf));
                if(n < 0){
                        perror("read from input failed");
                        break;
                }

                if(n-1 >= 0)
                        buf[n - 1] = '\0';
                if (insist_write(newsd, buf, strlen(buf)) != strlen(buf)) {
                        perror("write");
                        exit(1);
                }
                if(n==0) break;
        }
        if(FD_ISSET(newsd, &readfds)){
                n = read(newsd, buf, sizeof(buf));
                if(n<0){
                        perror("read from remote peer failed");
                        break;
                }
                if(n==0) break;
```

Ζ2. Κρυπτογραφημένο chat πάνω από TCP/IP

Η κρυπτογράφηση/αποκρυπτογράφηση των μηνυμάτων της σύνδεσης που έγινε στο 1ο ζητούμενο επιτυγχάνεται μέσω της χρήσης του οδηγού συσκευής cryptodev. Δηλαδή, πριν την αποστολή (κρυπτογράφηση) ή την απεικόνιση

(αποκρυπτογράφηση) των μηνυμάτων εκτελούνται κατάλληλες κλήσεις ioctl() προς το ειδικό αρχείο /dev/crypto του οδηγού cryptodev.
Οι κλήσεις ioct() που χρησιμοποιούνται είναι οι εξής:

- CIOCGSESSION
- CIOCCRYPT
- CIOCFSESSION

Αρχικά αφού κληθεί η open στο ειδικό αρχείο /dev/crypto της συσκευής κρυπτογράφησης, χρειάζεται να ξεκινήσουμε ένα session με τη συσκευή το οποίο θα περιέχει όλες πληροφορίες για τις κλήσεις κρυπτογράφησης/αποκρυπτογράφησης, όπως τον αλγόριθμο που θα χρησιμοποιηθεί (cipher), το κλειδί κρυπτογράφησης (key) , το μήκος του (keylen).
Αυτό επιτυγχάνεται με την ioctl() κλήση CIOCGSESSION η οποία δέχεται σαν όρισμα ένα session struct (δομή που περιέχει τις παραπάνω πληροφορίες) και αρχίζει ένα session δίνοντας του ένα χαρακτηριστικό session_identification (ses).

```
/*
     * Get crypto session for AES128
     */
    nfd = open("key.txt", O_RDONLY);
    read(nfd, data.key, KEY_SIZE);
    nfd = open("iv.txt", O_RDONLY);
    read(nfd, data.iv, BLOCK_SIZE);

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

    if (ioctl(cfd, CIOCGSESSION, &sess)){
            perror( "ioctl(CIOCGSESSION)");
            return 1;
    }
```

Σημειώνεται ότι για να λειτουργήσει η κρυπτογράφηση και αποκρυπτογράφηση των μηνυμάτων από τον client και τον server θα πρεπει να έχουν συμφωνήσει στην χρήση κοινού αλγορίθμου και κλειδιού (στην εργασία μας είναι προκαθορισμένο (διαβάζουν ένα ίδιο αρχείο) και για τους δύο το κοινό κλειδί που θα χρησιμοποιήσουν για αυτό δεν υπάρχει ανταλλαγή κάποιας τέτοιου είδους πληροφορίας).

Έπειτα, τόσο για την κρυπτογράφηση όσο και για αποκρυπτογράφηση χρησιμοποιείται η κλήση ioct() CIOCCRYPT, όπου δέχεται σαν όρισμα ένα crypt_op struct, το οποίο περιέχει τις παρακάτω πληροφορίες:

```
struct crypt_op {
        __u32  ses;             /* session identifier */
        __u16  op;              /* COP_ENCRYPT or COP_DECRYPT */
        __u16  flags;           /* see COP_FLAG_* */
        __u32  len;             /* length of source data */
        __u8   __user *src;     /* source data */
        __u8   __user *dst;     /* pointer to output data */
        /* pointer to output data for hash/MAC operations */
        __u8   __user *mac;
        /* initialization vector for encryption operations */
        __u8   __user *iv;
};
```

Ανάλογα με την τιμή του πεδίου op (COP_ENCRYPT / COP_DECRYPT) εκτελεί κρυπτογράφηση ή αποκρυπτογράφηση των δεδομένων του src buffer αντίστοιχα και αποθηκεύει το αποτέλεσμα στον dst buffer.
Υλοποιήθηκαν οι παρακάτω συναρτήσεις encrypt και decrypt που εκτελούνται πριν την αποστολή ή την εμφάνιση δεδομένων, οι οποίες δέχονται σαν ορίσματα τον fd του ειδικού αρχείου /dev/crypto που επιστρέφει η open πάνω σε αυτό το αρχείο, ένα struct session_op, και ένα buffer iv που είναι απαραίτητα για την κλήση της ioct() CIOCCRYPT, καθώς και δύο buffers in και enc_msg/dec_msg όπου ο πρώτος δέχεται το μήνυμα προς κρυπτογράφηση/αποκρυπτογράφηση και στον δεύτερο αποθηκεύεται το αποτέλεσμα :

```
int encrypt(int cfd, struct session_op sess, unsigned char *iv, unsigned char *in, unsigned char *enc_msg){
        struct crypt_op cryp;
        memset(&cryp, 0, sizeof(cryp));

        cryp.ses = sess.ses;
        cryp.len = DATA_SIZE;
        cryp.src = in;
        cryp.dst = enc_msg;
        cryp.iv = iv;
        cryp.op = COP_ENCRYPT;
```

```
        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                perror("encrypt_ioctl(CIOCCRYPT)");
                return 1;
        }
        return 0;
}

int decrypt(int cfd, struct session_op sess, unsigned char *iv, unsigned char *in, unsigned char *dec_msg){
        struct crypt_op cryp;
        memset(&cryp, 0, sizeof(cryp));

        cryp.ses = sess.ses;
        cryp.len = DATA_SIZE;
        cryp.src = in;
        cryp.dst = dec_msg;
        cryp.iv = iv;
        cryp.op = COP_DECRYPT;

        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                perror("decrypt_ioctl(CIOCCRYPT)");
                return 1;
        }
        return 0;
}
```

Τέλος όταν πλέον δεν χρειαζόμαστε τη συσκευή /dev/crypto κλείνουμε την σύνδεση που δημιουργήθηκε με την ioct() κλήση CIOCGSESSION. Αυτό γίνεται με την ioct() κλήση CIOCFSESSION η οποία χρειάζεται σαν όρισμα το session_identification (ses).

```
/* Finish crypto session */
        if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
                perror("ioctl(CIOCFSESSION)");
                return 1;
        }

        return 0;
}
```

Ζ3. Υλοποίηση συσκευής cryptodev με VirtIO

Στη συνέχεια, επεκτείνουμε την κρυπτογράφηση του ζητούμενου 2 χρησιμοποιώντας παραεικονικοποίηση αντί για πλήρη εικονικοποίηση. Με τη χρήση της παραεικονικοποίησης θα επιτευχθεί καλύτερη επίδοση. Θα πρέπει ωστόσο να γραφτεί κατάλληλος κώδικας στον οδηγό της συσκευής ώστε να επικοινωνει ο host με τον guest και επίσης χρειάζεται να υλοποιηθεί στην πλατφόρμα εικονοποίησης το κατάλληλο εικονικό υλικό που θα εξυπηρετεί τα αιτήματα του guest.
Αναλυτικότερα, οι κλήσεις open και ioctl() προς τη συσκευή /dev/crypto θα αντικατασταθούν από κλήσεις open και ioctl() στην πραγματική συσκευή που βρίσκεται στον host, οι οποίες θα μεταφέρονται στον hypervisor μέσω του πρότυπου virtio και πιο συγκεκριμένα από μέσω της δομής Virtqueue. Με τη σειρά του τότε ο hypervisor θα καλεί την αντίστοιχη κλήση ioctl και θα επιστρέφει τα δεδομένα πίσω στο VM που εκτελείται από το QEMU, μέσω των virtqueues. Η διαδικασία αυτή μεταφοράς των δεδομένων από τον host στον guest και ανάποδα γίνεται μέσω από τη κλήση συναρτήσεων που προσφέρει η διεπαφή του πρωτοκόλλου μεταφοράς virtio_ring.
Πιο συγκεκριμένα, παρακάτω παρατίθεται ο κώδικας για τον χώρο πυρήνα της εικονικής μηχανής (frontend) και για τον χώρο χρήστη του host (backend), οι οποίοι επικοινωνούν μέσω VirtQueues.

FRONTEND

static int crypto_chrdev_open(struct inode *inode, struct file *filp)

Η κλήση της παραπάνω συνάρτησης έχει ως στόχο να μεταφερθεί το αίτημα για άνοιγμα ενός αρχείου της κρυπτογραφικής συσκευής /dev/crypto στο backend. Αφού, συσχετίσουμε το open file με το crypto device που έχει τον minor number του αρχείου, στη συνέχεια, κλειδώνουμε το spinlock για να επιτευχθεί ο σωστός συγχρονισμός και χρησιμοποιούμε scatter-gather lists. Μέσω των λιστών αυτών θα μεταφερθούν τα δεδομένα από και προς το backend.
Μέσω της εντολής **sg_init_one(struct scatterlist *sg, const void *buf, unsigned int buflen)** αρχικοποιούμε μία scatter gather list. Στο buf αποθηκεύουμε την εικονική διεύθυνση της λιστας. Αρχικοποιούμε 2 τέτοιες λίστες, μία για το πέρασμα της κλήσης συστήματος που θα πραγματοποιηθεί στο backend syscall_type_sg (readable) και μία στην οποία θα μας επιστραφεί το host_fd, δηλαδή ο fd της συσκευής που άνοιξε στο backend (writable). Αποθηκεύουμε τις λίστες αυτές στον πίνακα sg και κρατάμε τον αριθμό των readable και writable λιστών (num_out, num_in) ώστε να τα περάσουμε στην παρακάτω συνάρτηση.

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

```
         **/
        num_out=0;
        num_in=0;

        spin_lock_irqsave(&(crdev->spinlock), flags);

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

Στη συνέχεια καλούμε την **virtqueue_add_sgs(struct virtqueue *vq, struct scatterlist sg[ ], unsigned int num_out, unsigned int num_int, void *data, gfp_t gfp),** η οποία προσθέτει τις λίστες στην ουρά VirtQueue, ώστε να περαστούν στο backend. Περνάμε σαν ορίσματα, το virtqueue που έχουμε αποθηκεύσει στην δομή crypto_device *crdev, το πίνακα με τις λίστες, τον αριθμό των readable και writable λιστών και τη διεύθυνση του πρώτου scatter gather list.

```
        //add data in VirtQueue
        retval = virtqueue_add_sgs(crdev->vq, sg, num_out, num_in, &syscall_type_sg,
GFP_ATOMIC);
        if (retval < 0){
                debug("virtqueue_add error");
                ret = retval;
                goto fail;
        }
```

Έπειτα ενημερώνουμε το backend ότι προσθέσαμε λίστες στο virtqueue εκτελώντας την **virtqueue_kick(struct virtqueue *vq).**

```
        //kick
        virtqueue_kick(crdev->vq);
```

Έπειτα, πραγματοποιείται busy wait καλώντας την **virtqueue_get_buf(struct virtqueue *vq, unsigned int *len),** η οποία επιστρέφει όταν το backend έχει επεξεργαστεί τις λίστες.

Όταν επιστρέψει η παραπάνω συνάρτηση, ξεκλειδώνουμε το spinlock και αποθηκεύουμε την τιμή που επέστρεψε το backend.

Ο λόγος που χρησιμοποιείται συγχρονισμός είναι η περίπτωση δημιουργίας race condition αν δύο διαφορετικές διεργασίες προσθέσουν ή αφαιρέσουν δεδομένα από το ίδιο virtqueue.
Η επίλυση του race condition μέσω συγχρονισμού θα γινόταν και με semaphore αντί για spinlock. Τότε θα έπρεπε μέσα στη while αν δεν ήταν έτοιμα τα δεδομένα να κάναμε sleep την διεργασία με την εντολή wait_event_interruptible και ως συνθήκη ώστε να ξυπνήσει θα έπρεπε να είναι έτοιμα τα δεδομένα. Επίσης, υπεύθυνος για να να καλέσει την wake_up και να ξυπνήσει την διεργασία είναι ο interrupt handler vq_has_data(), ο οποίος τρέχει σε interrupt context, όταν το backend επεξεργαστεί κάποιον απομονωτή του VirtQueue και καλέσει την virtio_notify().

BACKEND

static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
Όταν προστεθούν δεδομένα στη virtqueue, ενημερώνεται το backend και εκτελούνται οι παρακάτω συναρτήσεις. Αρχικά, καλείται η **virtqueue_pop(VirtQueue *vq, VirtQueueElement *elem)** ώστε να πάρει το backend τα δεδομένα από την ούρα. Ανάλογα με τα δεδομένα αυτά, το backend θα εκτελέσει διαφορετικές λειτουργίες. Οι λίστες για ανάγνωση και εγγραφή διαβάζονται αντίστοιχα από τις κατάλληλες θέσεις των πινάκων out_sg και in_sg. Μέσω του iov_base παίρνουμε την διεύθυνση του αντικειμένου το οποίο έχει κάθε λίστα scatter gather. Συγκεκριμένα, αν το frontend στείλει στο backend κλήση συστήματος open, το backend θα εξάγει το syscall_type μέσω της εντολής *syscall_type = elem->out_sg[0].iov_base;* και στη συνέχεια θα συσχετίσει το host_fd που προκύπτει με το άνοιγμα του αρχείου με το VirtQueue μέσω της εντολής *host_fd = elem->in_sg[0].iov_base;.* Στη συνέχεια θα καλέσει τις κατάλληλες κλήσεις συστήματος - στην προκειμένη περίπτωση open - ώστε να παράξει τα δεδομένα που ζητήθηκαν από το frontend.
Στη συνέχεια, εφόσον ολοκληρωθεί η διαδικασία και τα δεδομένα είναι έτοιμα, τα προσθέτει στη vq μέσω της **virtqueue_push()** και ενημερώνει το frontend μέσω της **virtio_notify()**.

Το όρισμα **VirtIODevice** που χρησιμοποιείται στην **vq_handle_output** είναι μία virtio cryptodev συσκευή η οποία δημιουργείται μέσω της συνάρτησης **virtio_cryptodev_realize,** μαζί με την virtqueue που απαιτείται για την ορθή επικοινωνία.

Για την συνάρτηση **static int crypto_chrdev_release(struct inode *inode, struct file *filp)** εκτελείται η αντίστοιχη διαδικασία με την **crypto_chrdev_open** με τη διαφορά ότι το backend πρέπει να καλέσει την κλήση συστήματος close(). Επιπλέον, το file descriptor του αρχείου της κρυπτογραφικής συσκευής που θα κλείσει αποθηκεύεται σε readable scatter gather list, εφόσον το περνάει το frontend στο backend και όχι το ανάποδο.

Επίσης, και για την **static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg**), η διαδικασία που ακολουθείται είναι ίδια με την διαδικασία που περιγράφηκε και στην open με τη διαφορά ότι χρειάζονται μερικά ακόμη scatter gather list. Εκτός από την κλήση συστήματος στη λίστα syscall_type_sg, δημιουργούμε άλλο ένα scatter gather list ioctl_cmd_sg για τις επιμέρους κλήσεις τις ioctl() - CIOCGSESSION, CIOCFSESSION, CIOCCRYPT. Για κάθε τέτοια κλήση, χρησιμοποιούμε scatter gather lists για να περάσουμε και να πάρουμε τα κατάλληλα δεδομένα με τον τρόπο που περιγράφηκε στην **crypto_chrdev_open.**

Επιπλέον αξίζει να σημειωθεί ότι καθώς οι κλήσεις της ioctl() χρειάζονται ορίσματα που δίνει ο χρήστης σε user space, χρειάζεται η κλήση της συνάρτησης **copy_from_user** και copy_to_user αντίστοιχα για να τα χρησιμοποιήσουμε στον πυρήνα. Αυτό είναι απαραίτητο γιατί μια διεργασία μπορεί να περάσει οποιονδήποτε δείκτη επιθυμεί και ο πυρήνας έχοντας πλήρη πρόσβαση μπορεί να “πανωγράψει” το περιεχόμενο της μνήμης στο συγκεκριμένο σημείο. Επομένως, για λόγους ασφάλειας χρησιμοποιούνται οι παραπάνω συναρτήσεις, οι οποίες ελέγχουν αν η διεύθυνση που έδωσε ο χρήστης ανήκει όντως στο user space.

Ο κώδικας της εργασίας έχει σταλεί σε ξεχωριστό αρχείο.