



Εθνικό Μετσόβιο Πολυτεχνείο

**Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών**

Εργαστήριο Λειτουργικών Συστημάτων

3^η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Κρυπτογραφική συσκευή VirtIO για QEMU – KVM

Ημερομηνία Παράδοσης : 07/06/2018

Ομάδα : oslab38

Ονοματεπώνυμο

Αριθμός Μητρώου

Βασιλάκης Εμμανουήλ

03114167

Γιάννου Αγγελική

03114021

Εξάμηνο

Ακαδημαϊκό Έτος

8^ο

2017-2018

1. Εργαλείο chat πάνω από TCP/IP sockets

1.1 Κώδικας

```
/*
 * socket-server.c
 * Simple TCP/IP communication using sockets
 *
 * Vasilakis Emmanouil, Giannou Aggeliki
 */

#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>

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

#include "socket-common.h"

int main(void)
{
    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd;
    socklen_t len;
    struct sockaddr_in sa;

    /* Make sure a broken connection doesn't kill us */
    signal(SIGPIPE, SIG_IGN);

    /* 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 to a well-known port */
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(TCP_PORT);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
        perror("bind");
        exit(1);
    }
    fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);

    /* Listen for incoming connections */
    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);
    }
    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));

    /* We break out of the loop when the remote peer goes away */
    chat(newsd);

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

/* This will never happen */
return 1;
}

```

```

/*
 * socket-client.c
 * Simple TCP/IP communication using sockets
 *
 * Vasilakis Emmanouil, Giannou Aggeliki
 */

#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>

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

#include "socket-common.h"

int main(int argc, char *argv[])
{
    int sd, port;
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
        exit(1);
    }
    hostname = argv[1];
    port = atoi(argv[2]); /* Needs better error checking */

    /* 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");

    /* Look up remote hostname on DNS */
    if ( !(hp = gethostbyname(hostname)) ) {
        printf("DNS lookup failed for host %s\n", hostname);
        exit(1);
    }

    /* 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");
}

```

```

        chat(sd);

        if (close(sd) < 0) {
            perror("close");
            exit(1);
        }
        fprintf(stdout, "Server is down, client is exiting\n");

        return 0;
    }
}

```

```

/*
 * socket-common.h
 *
 * Simple TCP/IP communication using sockets
 *
 * Vasilakis Emmanouil, Giannou Aggeliki
 */

#ifndef _SOCKET_COMMON_H
#define _SOCKET_COMMON_H

/* Compile-time options */
#define TCP_PORT 35001
#define TCP_BACKLOG 5

#endif /* _SOCKET_COMMON_H */

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = write(fd, buf, cnt);
        if (ret < 0)
            return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

```

```

/* Chat implementation, shared code between client and server */
void chat(int socket_fd)
{
    char buf[100];
    fd_set readfds;
    ssize_t n;

    for(;;) {
        FD_ZERO(&readfds);
        FD_SET(0, &readfds);
        FD_SET(socket_fd, &readfds);
        if (select(socket_fd+1, &readfds, NULL, NULL, NULL) == -1) {
            perror("select()\n");
            exit(1);
        }
        if (FD_ISSET(socket_fd, &readfds)) {
            /* Read from peer and write it to standard output */
            n = read(socket_fd, buf, sizeof(buf));

            if (n < 0) {
                perror("read from socket failed\n");
                exit(1);
            }
            else if (n == 0) { /* Peer is down */
                fprintf(stderr, "Peer went away\n");
                return;
            }

            if (insist_write(1, buf, n) != n) {
                perror("write to stdout failed\n");
                exit(1);
            }
        }
        if (FD_ISSET(0, &readfds)) {
            /* Read from stdin and write it to socket */
            n = read(0, buf, sizeof(buf)-1);

            if (n < 0) {
                perror("read from stdin failed\n");
                exit(1);
            }
            else if (n == 0) {
                perror("ERROR: EOF stdin\n");
                exit(1);
            }

            buf[n] = '\0'; /* last character \0 */
            if (insist_write(socket_fd, buf, n+1) != n+1) {
                perror("write to remote peer failed\n");
                exit(1);
            }
        }
    }
}

```

1.2 Σχόλια

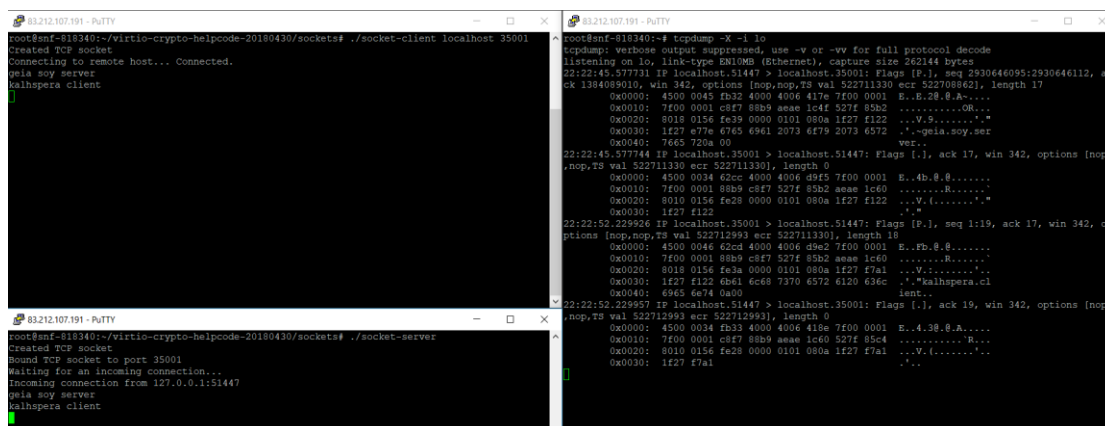
Σκοπός του ερωτήματος αυτού, είναι η επικοινωνία δύο διεργασιών (client και server) με χρήση TCP/IP sockets.

Η μόνη διαφοροποίηση στην συμπεριφορά του client και του server είναι στο πώς διαχειρίζονται τα sockets αφού τα δημιουργήσουν. Συγκεκριμένα, από τη μεριά του server, βλέπουμε ότι πραγματοποιείται το bind και μετά βρίσκεται σε κατάσταση listen, περιμένοντας συνεχώς νέες συνδέσεις από τον client τις οποίες κάνει accept. Για κάθε accept που κάνει προκύπτει ένα νέο socket descriptor, μέσω του οποίου επικοινωνεί με τον client. Ο client το μόνο που κάνει είναι αίτημα για connect στο socket του server.

Το chat υλοποιείται μέσω της συνάρτησης chat, που βρίσκεται στο αρχείο socket-common.h και καλείται και από την μεριά του client και από την μεριά του server. Χρησιμοποιούμε την συνάρτηση select για να αποφασίσουμε κάθε φορά από ποιον file descriptor θα διαβάσουμε, δηλαδή αν θα διαβάσουμε από το stdin ή από τον socket descriptor. Σε περίπτωση που διαβάζουμε από stdin, ό,τι διαβάζουμε το προωθούμε στο socket descriptor, ώστε να περάσουμε το μήνυμα στον συνομιλητή μας. Σε περίπτωση που διαβάζουμε από το socket descriptor, τυπώνουμε ό,τι διαβάζουμε από τον συνομιλητή μας στο stdout, ώστε να μπορεί να το δει ο χρήστης.

1.3 Παράδειγμα εκτέλεσης

Παρακάτω παρουσιάζουμε ένα στιγμιότυπο της λειτουργίας του chat, όπου φαίνεται η καταγεγραμμένη κίνηση στο δίκτυο που προκύπτει από τα μηνύματα που ανταλλάσσονται. Μπορούμε εύκολα να διακρίνουμε το περιεχόμενο των μηνυμάτων.



```
root@snf-818340:~# tcpdump -X -i lo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
22:22:45.577731 IP localhost.51447 > localhost.35001: Flags [P.], seq 2930646095:2930646112, win 0, len 17
E..4b.8.8.....
0x0000: 4500 0045 f032 0000 417e 7f00 0001  E..4b.8.8.....
0x0010: 7f00 0001 c9f7 88b9 aaaa 1c4f 527f 85b2  ....OR...
0x0020: 8018 0156 fe39 0000 0101 080a 1f27 f122  ..V.S.....
0x0030: 1f27 f77e 6745 6961 2073 6f79 2073 6572  ..-geia.soy.ser
0x0040: 7465 720a 00                                val..
22:22:45.577744 IP localhost.35001 > localhost.51447: Flags [P.], ack 17, win 342, options [nop,TS val 522711330, seq 522711330], length 0
E..4b.8.8.....
0x0000: 4500 0045 f032 0000 417e 7f00 0001  E..4b.8.8.....
0x0010: 7f00 0001 88b9 c9f7 527f 85b2 aaaa 1c4f  ....R.....
0x0020: 8018 0156 fe39 0000 0101 080a 1f27 f122  ..V.S.....
0x0030: 1f27 f77e 6745 6961 2073 6f79 2073 6572  ..-geia.soy.ser
0x0040: 7465 720a 00                                val..
22:22:52.229926 IP localhost.35001 > localhost.51447: Flags [P.], seq 1:19, ack 17, win 342, options [nop,TS val 522712993, seq 522712993], length 18
E..4b.8.8.....
0x0000: 4500 0046 62cd 4000 4006 d9e2 7f00 0001  E..4b.8.8.....
0x0010: 7f00 0001 88b9 c9f7 527f 85b2 aaaa 1c4f  ....R.....
0x0020: 8018 0156 fe3a 0000 0101 080a 1f27 f7a1  ..V.S.....
0x0030: 1f27 f122 6b61 6c68 7370 6572 6120 636c  .."kalphera.cl
0x0040: 6945 6e74 0a00                                lent..
22:22:52.229957 IP localhost.51447 > localhost.35001: Flags [P.], ack 19, win 342, options [nop,TS val 522712993, seq 522712993], length 0
E..4.38.8.A....
0x0000: 4500 0034 fb33 4000 4006 418e 7f00 0001  E..4.38.8.A....
0x0010: 7f00 0001 c9f7 88b9 aaaa 1c4f 527f 85b2  ....R.....
0x0020: 8018 0156 fe38 0000 0101 080a 1f27 f7a1  ..V.S.....
0x0030: 1f27 f7a1                                ..V.S.....
```

2. Κρυπτογραφημένο chat πάνω από TCP/IP

2.1 Κώδικας crypto-common.h

```
/*
 * crypto-common.h
 *
 * Simple TCP/IP communication using sockets and cryptodev
 *
 * Vasilakis Emmanouil, Giannou Aggeliki
 */

#ifndef _SOCKET_COMMON_H
#define _SOCKET_COMMON_H

/* Compile-time options */
#define TCP_PORT 35001
#define TCP_BACKLOG 5

#endif /* _SOCKET_COMMON_H */

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

#define DATA_SIZE 256
#define BLOCK_SIZE 16
#define KEY_SIZE 16 /* AES128 */

const char *MY_IV = "liastesntomates"; /* length 15 + '\0' which is automatically added */
const char *MY_KEY = "gyrosmetzatziki";

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = write(fd, buf, cnt);
        if (ret < 0) return ret;
        buf += ret;
        cnt -= ret;
    }
    return orig_cnt;
}

/* Insist until all of the data has been read */
ssize_t insist_read(int fd, void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = read(fd, buf, cnt);
        if (ret <= 0) return ret;
        buf += ret;
        cnt -= ret;
    }
}
```



```

        return orig_cnt;
    }

/*
 * Chat implementation, shared code between client and server
 * Different session for different clients
 */
void chat(int socket_fd)
{
    fd_set readfds;
    int useful_bytes;
    ssize_t n;
    int cryptofd;
    struct session_op sess;
    struct crypt_op cryp;
    struct {
        unsigned char    plaintext[DATA_SIZE],
                        ciphertext[DATA_SIZE],
                        iv[BLOCK_SIZE],
                        key[KEY_SIZE];
    } data;

    cryptofd = open("/dev/crypto", O_RDWR);
    if (cryptofd < 0) {
        perror("open(/dev/crypto)");
        exit(1);
    }

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

    memcpy(data.key, MY_KEY, KEY_SIZE);
    memcpy(data.iv, MY_IV, BLOCK_SIZE);

    /*
     * Get crypto session for AES128
     */
    sess.cipher = CRYPTO_AES_CBC;
    sess.keylen = KEY_SIZE;
    sess.key = data.key;

    if (ioctl(cryptofd, CIOGSESSION, &sess)) {
        perror("ioctl(CIOGSESSION)");
        exit(1);
    }

    for(;;) {
        FD_ZERO(&readfds);
        FD_SET(0, &readfds);
        FD_SET(socket_fd, &readfds);
        if (select(socket_fd+1, &readfds, NULL, NULL, NULL) == -1) {
            perror("select()\n");
            exit(1);
        }
        if (FD_ISSET(socket_fd, &readfds)) {
            /* Read from peer and write it to standard output */
            n = insist_read(socket_fd, data.ciphertext, sizeof(data.ciphertext));

            if (n < 0) {
                perror("read from socket failed\n");
                exit(1);
            }
            else if (n == 0) {
                /* Peer is down */
            }
        }
    }
}

```

```

        fprintf(stderr, "Peer went away\n");

        /* Finish crypto session */
        if (ioctl(cryptofd, CIOCFSESSION, &sess.ses)) {
            perror("ioctl(CIOCFSESSION)");
            exit(1);
        }
        if (close(cryptofd) < 0) {
            perror("close(cryptofd)");
            exit(1);
        }
        return;
    }

    /*
     * Decrypt data.ciphertext to data.plaintext
     */
    cryp.ses = sess.ses;
    cryp.src = data.ciphertext;
    cryp.dst = data.plaintext;
    cryp.len = DATA_SIZE;
    cryp.iv = data.iv;
    cryp.op = COP_DECRYPT;
    if (ioctl(cryptofd, CIOCCRYPT, &cryp)) {
        perror("ioctl(CIOCCRYPT)");
        exit(1);
    }

    /*
     * Find number of useful bytes to print,
     * '\0' + bytes read from stdin inside data.plaintext[0]
     */
    useful_bytes = (int) data.plaintext[0] + 1;
    if (insist_write(1, data.plaintext + 1, useful_bytes) != useful_bytes) {
        perror("write to stdout failed\n");
        exit(1);
    }
}

if (FD_ISSET(0, &readfds)) {
    /* Read from stdin and write it to socket */
    n = read(0, data.plaintext + 1, sizeof(data.plaintext)-2);

    if (n < 0) {
        perror("read from stdin failed\n");
        exit(1);
    }
    else if (n == 0) {
        perror("ERROR:   EOF stdin\n");
        exit(1);
    }

    /* last character \0 */
    data.plaintext[n+1] = '\0';

    /* number of bytes read from stdin */
    data.plaintext[0] = (char) n;

    /*
     * Encrypt data.plaintext to data.ciphertext
     */
    cryp.ses = sess.ses;
    cryp.len = DATA_SIZE;
    cryp.src = data.plaintext;

```

```

    crypt.dst = data.ciphertext;
    crypt.iv = data.iv;
    crypt.op = COP_ENCRYPT;

    if (ioctl(cryptofd, CIOCCRYPT, &crypt)) {
        perror("ioctl(CIOCCRYPT)");
        exit(1);
    }

    if (insist_write(socket_fd, data.ciphertext, sizeof(data.ciphertext)) !=
        sizeof(data.ciphertext)) {
        perror("write to remote peer failed\n");
        exit(1);
    }
}
}
}

```

2.2 Σχόλια

Σε σχέση με το προηγούμενο ζήτημα, μοναδική αλλαγή παρατηρείται στον κώδικα του socket-common.h (crypto-common.h πλέον) και στην υλοποίηση της συνάρτησης chat, όπου έχουν προστεθεί οι κατάλληλες κλήσεις συστήματος, ώστε να γίνεται επιτυχώς η κρυπτογράφηση και αποκρυπτογράφηση των μηνυμάτων με χρήση του /dev/crypto. Γίνεται η θεώρηση ότι ανεξαρτήτως του μεγέθους του μηνύματος, θα στέλνονται 256 bytes δεδομένων, τα οποία κρυπτογραφούνται προτού σταλούν, και το πρώτο εκ των οποίων μας δείχνει τον αριθμό των bytes που έστειλε πραγματικά ο χρήστης και είναι ωφέλιμο. Μέσω αυτής της τιμής, γνωρίζουμε πόσα bytes δεδομένων πρέπει να τυπώσουμε στο stdout, καθώς τα υπόλοιπα αποτελούν σκουπίδια (padding).

2.3 Παράδειγμα εκτέλεσης

Παρακάτω παρουσιάζουμε ένα στιγμιότυπο της λειτουργίας του κρυπτογραφημένου chat, όπου φαίνεται η καταγεγραμμένη κίνηση στο δίκτυο και η κρυπτογράφηση των μηνυμάτων που ανταλλάσσονται.

```

8121210791 - PUTTY
root@kali:~/crypto-helpcode-20180430/cryptodev# ./crypto-client localhost 35001
Created TCP socket
Connecting to remote host... Connected.
gata soy server
kathipera client

8121210791 - PUTTY
root@kali:~/crypto-helpcode-20180430/cryptodev# ./crypto-server
Created TCP socket
Bound TCP socket to port 35001
Waiting for an incoming connection...
Incoming connection from 127.0.0.1:51446
gata soy server
kathipera client

22:12:02.909706 IP localhost.35001 > localhost.51446: Flags [P], ack 256, win 350, options [n
op, nop, TS val 522550463, ecr 522550463], length 0
0x0000: 4500 0034 2e07 4000 4006 0abb 7f00 0001 E..4..8.8.....
0x0010: 7f00 0001 8b89 c8f6 fe94 400f 3353 b86e .....8.38.n
0x0020: 8010 015e fe28 0000 0101 080a 1f25 7d87 .....f.....N.
0x0030: 1f25 7d87
22:12:10.425995 IP localhost.35001 > localhost.51446: Flags [P], seq 1:257, ack 256, win 350
, options [nop, nop, TS val 522552542, ecr 522550463], length 256
0x0000: 4500 0134 2e08 4000 4006 0dba 7f00 0001 E..4..8.8.....
0x0010: 7f00 0001 8b89 c8f6 fe94 400f 3353 b86e .....8.38.n
0x0020: 8010 015e fe28 0000 0101 080a 1f25 7d87 .....f.....N.
0x0030: 1f25 7d87 e30e fe7e 7610 678a 4d94 35fe .N.....v.g.M.5.
0x0040: c839 a3d2 e251 e220 1161 fe63 a224 759d .9.B.Q.....a.d.
0x0050: c3bc 3b97 8017 4010 3b0c f02b c5f1 6c0c .....0..+..
0x0060: a32a a539 7332 d6aa c6d0 9419 7c79 ab4a *.9a2.....[.
0x0070: c6d2 7f6a e15e 8e62 9967 d562 3763 1469 .w..j...g.O...
0x0080: 5715 9a0e 615e 8092 5bca 830a 5679 ac2b W..s..f...V..
0x0090: 2010 5be2 2932 f9aa 3a81 af55 b994 286e .{.2..>..U..f.
0x00a0: 5e10 ac39 d10e 576f 61e3 39ef 546d d973 .9.....a.f.T..s
0x00b0: 4c6f 2285 513a d427 359f 5446 5e41 994d Lw".Q."5.TP*A.M
0x00c0: d126 9331 bbf7 f324 9f67 e436 a199 d036 .k.l...g..6...
0x00d0: a44a 12ef af9a 2533 e8ac 61c0 90ac b968 .f..O..B...
0x00e0: 9a43 947f a61f fb44 839e a2e2 0a49 f65e M.....D...I..
0x00f0: ecc6 0c03 50b8 7240 e350 0e86 a0c6 52a0 ...P..8.P...R.
0x0100: 8a72 f312 a1b7 03e7 4453 d937 62ac fe32 .....M..7k..2
0x0110: e63a a23a 4376 9845 60a6 d5c8 ecc8 01cb .:TCV.B....h..
0x0120: aed4 e03d e5f8 2621 2128 a8b4 a9ae 2c1e ...Xa!f.....
0x0130: e707 69d0
22:12:10.426024 IP localhost.51446 > localhost.35001: Flags [P], ack 257, win 350, options [n
op, nop, TS val 522552542, ecr 522552542], length 0
0x0000: 4500 0034 e94e 4000 4006 6359 7f00 0001 E..4.h8.8.cY....
0x0010: 7f00 0001 c8f6 8b89 3353 b86e fe94 410f .....8.n..A.
0x0020: 8010 015e fe28 0000 0101 080a 1f25 7d87 .....f.....N.
0x0030: 1f25 7d87

```

3. Υλοποίηση συσκευής cryptodev με VirtIO

3.1 Κώδικας

FRONTEND

```
/*
 * crypto-chrdev.c
 *
 * Implementation of character devices
 * for virtio-crypto device
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 */
#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"

/*
 * Global data
 */
struct cdev crypto_chrdev_cdev;

/**
 * Given the minor number of the inode return the crypto device
 * that owns that number.
 */
static struct crypto_device *get_crypto_dev_by_minor(unsigned int minor)
{
    struct crypto_device *crdev;
    unsigned long flags;

    debug("Entering");

    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);

    debug("Leaving");
    return crdev;
}
```

```

/*****
 * Implementation of file operations
 * for the Crypto character device
 *****/

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

    debug("Entering open");

    num_in = 0;
    num_out = 0;
    syscall_type = kzalloc(sizeof(unsigned int), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;
    host_fd = kzalloc(sizeof(int), 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;
    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(&syscall_type_sg, syscall_type, sizeof(unsigned int));
    sgs[num_out++] = &syscall_type_sg;
    sg_init_one(&host_fd_sg, host_fd, sizeof(int));
    sgs[num_out + num_in++] = &host_fd_sg;

    /**
     * Send data to the host.
     */

    /* Lock ?? */
    if (down_interruptible(&crdev->lock)) return -ERESTARTSYS;
    ret = virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
    if (ret) {
        debug("virtqueue_add_sgs failed in open");
    }
}

```

```

        up(&crdev->lock);
        goto fail;
    }
    virtqueue_kick(crdev->vq);

    /**
     * Wait for the host to process our data.
     */
    while (virtqueue_get_buf(crdev->vq, &len) == NULL);    /* do nothing */

    /* Unlock */
    up(&crdev->lock);

    /* If host failed to open() return -ENODEV. */
    if (*host_fd == -1) {
        debug("open(/dev/crypto)");
        ret = -ENODEV;
        goto fail;
    }
    crof->host_fd = *host_fd;

fail:
    kfree(syscall_type);
    kfree(host_fd);
    debug("Leaving open");
    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;
    unsigned int *syscall_type;
    unsigned int num_out, num_in, len;
    int *host_fd;
    struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
    struct virtqueue *vq = crdev->vq;

    debug("Entering release");

    num_out = 0;
    num_in = 0;

    syscall_type = kzalloc(sizeof(unsigned int), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;
    host_fd = kzalloc(sizeof(int), GFP_KERNEL);
    *host_fd = crof->host_fd;

    sg_init_one(&syscall_type_sg, syscall_type, sizeof(unsigned int));
    sgs[num_out++] = &syscall_type_sg;
    sg_init_one(&host_fd_sg, host_fd, sizeof(int));
    sgs[num_out + num_in++] = &host_fd_sg;
    /* host_fd with W flag, so we can return errno */

    /**
     * Send data to the host.
     */

    /* Lock ?? */
    if (down_interruptible(&crdev->lock)) return -ERESTARTSYS;
}

```

```

ret = virtqueue_add_sgsvq, sgsv, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
if (ret) {
    debug("virtqueue_add_sgsv failed in close");
    up(&crdev->lock);
    goto out1;
}
virtqueue_kick(vq);

/**
 * Wait for the host to process our data.
 */
while (virtqueue_get_buf(vq, &len) == NULL); /* do nothing */

/* Unlock */
up(&crdev->lock);

if (*host_fd) { /* *host_fd = 0->SUCCESS | errno->failure */
    debug("release failed");
    ret = *host_fd;
}

out1:
kfree(syscall_type);
kfree(host_fd);
kfree(crof);

debug("Leaving release");
return ret;
}

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    long ret = 0;
    int err;
    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, host_return_val_sg,
                                                session_key_sg, session_op_sg, ses_id_sg,
                                                crypt_op_sg, src_sg, iv_sg, dst_sg, *sgs[8];

    unsigned int num_out, num_in, len;
    unsigned int *syscall_type, *ioctl_cmd;
    int *host_fd, *host_return_val;
    struct session_op *session_op;
    struct crypt_op *crypt_op;
    unsigned char *src, *iv, *dst, *session_key, *temp;
    __u32 *ses_id;

    debug("Entering ioctl");

    /**
     * Allocate all data that will be sent to the host.
     */
    syscall_type = kzalloc(sizeof(unsigned int), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPT_SYSCALL_IOCTL;
    host_fd = kzalloc(sizeof(int), GFP_KERNEL);
    *host_fd = crof->host_fd;
    ioctl_cmd = kzalloc(sizeof(unsigned int), GFP_KERNEL);
    *ioctl_cmd = cmd;

    host_return_val = kzalloc(sizeof(int), GFP_KERNEL);
    *host_return_val = -1;

```

```

num_out = 0;
num_in = 0;

/**
 * These are common to all ioctl commands.
 */
sg_init_one(&syscall_type_sg, syscall_type, sizeof(unsigned int));
sgs[num_out++] = &syscall_type_sg;
sg_init_one(&host_fd_sg, host_fd, sizeof(int));
sgs[num_out++] = &host_fd_sg;
sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(unsigned int));
sgs[num_out++] = &ioctl_cmd_sg;

/**
 * Add all the cmd specific sg lists.
 */
switch (cmd) {
    case CIOCGSESSION:
        debug("CIOCGSESSION");
        session_op = kzalloc(sizeof(struct session_op), GFP_KERNEL);
        if (copy_from_user(session_op, (struct session_op *) arg, sizeof(struct session_op)))
        {
            kfree(session_op);
            return -EFAULT;
        }

        session_key = kzalloc(sizeof(char) * session_op->keylen, GFP_KERNEL);
        if (copy_from_user(session_key, session_op->key, sizeof(char) * session_op->keylen)) {
            kfree(session_op);
            kfree(session_key);
            return -EFAULT;
        }

        temp = session_op->key;
        session_op->key = session_key;

        sg_init_one(&session_key_sg, session_key, sizeof(char) * session_op->keylen);
        sgs[num_out++] = &session_key_sg;
        sg_init_one(&session_op_sg, session_op, sizeof(struct session_op));
        sgs[num_out + num_in++] = &session_op_sg;

        sg_init_one(&host_return_val_sg, host_return_val, sizeof(int));
        sgs[num_out + num_in++] = &host_return_val_sg;

        /* Lock ?? */
        if (down_interruptible(&crdev->lock)) return -ERESTARTSYS;

        err = virtqueue_add_sgs(vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
        if (err) {
            debug("virtqueue_add_sgs failed in CIOCGSESSION");
            up(&crdev->lock);
            ret = err;
            goto out2;
        }
        virtqueue_kick(vq);

        /**
         * Wait for the host to process our data.
         */
        while (virtqueue_get_buf(vq, &len) == NULL); /* do nothing */

        /* Unlock */

```



```

up(&crdev->lock);

session_op->key = temp;
if (copy_to_user((struct session_op *) arg, session_op, sizeof(struct session_op))){
    kfree(session_key);
    kfree(session_op);
    return -EFAULT;
}

ret = *host_return_val;

kfree(session_key);
kfree(session_op);
break;

case CIOCFSESSION:
    debug("CIOCFSESSION");
    ses_id = kzalloc(sizeof(__u32), GFP_KERNEL);
    if (copy_from_user(ses_id, (__u32 *) arg, sizeof(__u32)))
    {
        kfree(ses_id);
        return -EFAULT;
    }
    sg_init_one(&ses_id_sg, ses_id, sizeof(__u32));
    sgs[num_out++] = &ses_id_sg;

    sg_init_one(&host_return_val_sg, host_return_val, sizeof(int));
    sgs[num_out + num_in++] = &host_return_val_sg;

    /* Lock ?? */
    if (down_interruptible(&crdev->lock)) return -ERESTARTSYS;

    err = virtqueue_add_sgs(vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
    if (err) {
        debug("virtqueue_add_sgs failed in CIOCFSESSION");
        up(&crdev->lock);
        ret = err;
        goto out2;
    }
    virtqueue_kick(vq);

    /**
     * Wait for the host to process our data.
     */
    while (virtqueue_get_buf(vq, &len) == NULL); /* do nothing */

    /* Unlock */
    up(&crdev->lock);

    ret = *host_return_val;

    kfree(ses_id);
    break;

case CIOCCRYPT:
    debug("CIOCCRYPT");
    crypt_op = kzalloc(sizeof(struct crypt_op), GFP_KERNEL);
    if (copy_from_user(crypt_op, (struct crypt_op *) arg, sizeof(struct crypt_op)))
    {
        kfree(crypt_op);
        return -EFAULT;
    }

```

```

src = kzalloc(sizeof(char) * crypt_op->len, GFP_KERNEL);
if (copy_from_user(src, crypt_op->src, sizeof(char) * crypt_op->len)) {
    kfree(crypt_op);
    kfree(src);
    return -EFAULT;
}

iv = kzalloc(sizeof(char) * VIRTIO_CRYPTO_BLOCK_SIZE, GFP_KERNEL);
if (copy_from_user(iv, crypt_op->iv, sizeof(char) * VIRTIO_CRYPTO_BLOCK_SIZE)) {
    kfree(crypt_op);
    kfree(src);
    kfree(iv);
    return -EFAULT;
}

dst = kzalloc(sizeof(char) * crypt_op->len, GFP_KERNEL);

temp = crypt_op->dst;
crypt_op->src = src;
crypt_op->iv = iv;
crypt_op->dst = 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, sizeof(char) * crypt_op->len);
sgs[num_out++] = &src_sg;
sg_init_one(&iv_sg, crypt_op->iv, sizeof(char) * VIRTIO_CRYPTO_BLOCK_SIZE);
sgs[num_out++] = &iv_sg;
sg_init_one(&dst_sg, crypt_op->dst, sizeof(char) * crypt_op->len);
sgs[num_out + num_in++] = &dst_sg;

sg_init_one(&host_return_val_sg, host_return_val, sizeof(int));
sgs[num_out + num_in++] = &host_return_val_sg;

/* Lock ?? */
if (down_interruptible(&crdev->lock)) return -ERESTARTSYS;

err = virtqueue_add_sgs(vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
if (err) {
    debug("virtqueue_add_sgs failed in CIOCCRYPT");
    up(&crdev->lock);
    ret = err;
    goto out2;
}
virtqueue_kick(vq);

/**
 * Wait for the host to process our data.
 */
while (virtqueue_get_buf(vq, &len) == NULL); /* do nothing */

/* Unlock */
up(&crdev->lock);

if (copy_to_user(temp, dst, sizeof(char) * crypt_op->len))
{
    kfree(crypt_op);
    kfree(src);
    kfree(iv);
    kfree(dst);
    return -EFAULT;
}

```

```

        ret = *host_return_val;
        kfree(crypt_op);
        kfree(src);
        kfree(iv);
        kfree(dst);
        break;

    default:
        debug("Unsupported ioctl command");
        break;
}

out2:
    kfree(host_return_val);
    kfree(host_fd);
    kfree(ioctl_cmd);
    kfree(syscall_type);

    debug("Leaving ioctl");

    return ret;
}

static ssize_t crypto_chrdev_read(struct file *filp, char __user *usrbuf,
    size_t cnt, loff_t *f_pos)
{
    debug("Entering");
    debug("Leaving");
    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(void)
{
    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);
        goto out;
    }
    ret = cdev_add(&crypto_chrdev_cdev, dev_no, crypto_minor_cnt);
    if (ret < 0) {
        debug("failed to add character device");
        goto out_with_chrdev_region;
    }

    debug("Completed successfully");
}

```

```

        return 0;

out_with_chrdev_region:
    unregister_chrdev_region(dev_no, crypto_minor_cnt);
out:
    return ret;
}

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

    debug("entering");
    dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
    cdev_del(&crypto_chrdev_cdev);
    unregister_chrdev_region(dev_no, crypto_minor_cnt);
    debug("leaving");
}

```

BACKEND

```

/*
 * Virtio Crypto Device
 *
 * Implementation of virtio-crypto qemu backend device.
 *
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 */

#include <qemu/iov.h>
#include "hw/virtio/virtio-serial.h"
#include "hw/virtio/virtio-crypto.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <crypto/cryptodev.h>

#include <unistd.h>
#include <errno.h>

static uint32_t get_features(VirtIODevice *vdev, uint32_t features)
{
    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;

    int *host_fd, *ret;
    unsigned int *cmd;
    struct session_op *session_op;
    struct crypt_op crypt_op;
    __u32 *ses_id;

    DEBUG_IN();

    if (!virtqueue_pop(vq, &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_CRYPTO_SYSCALL_TYPE_OPEN:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN");
        host_fd = elem.in_sg[0].iov_base;
        *host_fd = open("/dev/crypto", O_RDWR);
        break;

    case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE");
        host_fd = elem.in_sg[0].iov_base;
        *host_fd = close(*host_fd);
        if (*host_fd) *host_fd = errno;
        break;

    case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL");
        host_fd = elem.out_sg[1].iov_base;
        cmd = elem.out_sg[2].iov_base;
        switch (*cmd) {
            case CIOGSESSION:
                DEBUG("CIOGSESSION IOCTL");
                ret = elem.in_sg[1].iov_base;
                session_op = elem.in_sg[0].iov_base;
                session_op->key = elem.out_sg[3].iov_base;
                *ret = ioctl(*host_fd, CIOGSESSION, session_op);
                break;

            case CIOCFSESSION:
                DEBUG("CIOCFSESSION IOCTL");
                ret = elem.in_sg[0].iov_base;
                ses_id = elem.out_sg[3].iov_base;
                *ret = ioctl(*host_fd, CIOCFSESSION, ses_id);
                break;
        }
    }
}

```

```

        case CIOCCRYPT:
            DEBUG("CIOCCRYPT IOCTL");
            ret = elem.in_sg[1].iov_base;
            memcpy(&crypt_op, elem.out_sg[3].iov_base, sizeof(struct crypt_op));
            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;
            *ret = ioctl(*host_fd, CIOCCRYPT, &crypt_op);
            break;

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

default:
    DEBUG("Unknown syscall_type");
}

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

static void virtio_crypto_realize(DeviceState *dev, Error **errp)
{
    VirtIODevice *vdev = VIRTIO_DEVICE(dev);

    DEBUG_IN();

    virtio_init(vdev, "virtio-crypto", 13, 0);
    virtio_add_queue(vdev, 128, vq_handle_output);
}

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

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

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

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

    k->realize = virtio_crypto_realize;
    k->unrealize = virtio_crypto_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_crypto_info = {
    .name = TYPE_VIRTIO_CRYPTO,
    .parent = TYPE_VIRTIO_DEVICE,

```

```

        .instance_size    = sizeof(VirtCrypto),
        .class_init       = virtio_crypto_class_init,
};

static void virtio_crypto_register_types(void)
{
    type_register_static(&virtio_crypto_info);
}

type_init(virtio_crypto_register_types)

```

3.2 Σχόλια

Σκοπός αυτού του μέρους της εργαστηριακής άσκησης είναι η επικοινωνία μεταξύ μίας εικονικής μηχανής (guest) και του πραγματικού μηχανήματος (host), ώστε ο πρώτος να μπορεί να αξιοποιήσει το υλικό του δεύτερου (τεχνική paravirtualization), για την υλοποίηση της κρυπτογράφησης. Εμείς κληθήκαμε να επεξεργαστούμε τον βοηθητικό κώδικα που μας δόθηκε, για την επίτευξη της επικοινωνίας αυτής.

Ουσιαστικά, όταν το guest userspace καλεί μία συνάρτηση που αφορά το /dev/crypto (π.χ. ioctl), τότε ο πυρήνας αυτού μέσω των virtqueues – πρότυπο virtio – “στέλνει” στον host τα απαραίτητα δεδομένα, ώστε ο δεύτερος να υλοποιήσει τελικά την κλήση αυτή. Ο host ειδοποιείται ότι υπάρχουν νέα δεδομένα μέσω της virtqueue_kick, ενώ ο guest “περιμένει” να επιστρέψει ο host τα επεξεργασμένα δεδομένα με χρήση της virtqueue_get_buf. Αξίζει να σημειωθεί ότι, χρησιμοποιείται mutex lock στο crdev καθώς πρέπει να εξασφαλίσουμε ότι ο host θα μας επιστρέψει τα δικά μας δεδομένα. Πιο συγκεκριμένα, αν για παράδειγμα, δύο διεργασίες πρόσθεταν κάποια δεδομένα στην virtqueue, τότε δεν είναι απαραίτητο ότι ο host θα τα επεξεργαζόταν με τη σειρά που του δόθηκαν. Κατά συνέπεια, θα ήταν απαραίτητο κάποιο tagging, προκειμένου η κάθε διεργασία να μπορεί να αναγνωρίσει τα δεδομένα που της αντιστοιχούν. Όσον αφορά τον host, αρχικά μέσω της συνάρτησης virtqueue_pop “παίρνει” τα στοιχεία που του έστειλε ο guest, τα επεξεργάζεται κατάλληλα και ύστερα μέσω της virtqueue_push τα στέλνει στον guest. Μόλις ο host εκτελέσει την virtqueue_push, η συνάρτηση virtqueue_get_buf σταματάει να επιστρέφει NULL. Σημειώνεται ότι, η συνάρτηση virtio_notify ενημερώνει τον guest μέσω interrupt ότι έχει γίνει ενημέρωση των δεδομένων.

Η αρχικοποίηση του mutex lock γίνεται στο αρχείο crypto-module.c.

3.3 Παράδειγμα εκτέλεσης

Τα αποτελέσματα είναι ίδια με τα παραπάνω, για αυτό και δεν παρατίθενται παραδείγματα εκτέλεσης.