Εργαστήριο Λειτουργικών Συστημάτων

3η Εργαστηριακή Άσκηση

Ευθμιος Τζίνης Α.Μ: 03112007 Κωσταντίνος Καλλάς Α.Μ: 03112057

19 Ιουνίου

Σημαντική σημείωση: Τα ζητήματα 1 και 2 έχουν υλοποιηθεί μαζί, στο ίδιο κομάτι κώδικα, εξαιτίας της σχετικότητας τους ως προς τις λύσεις. Όλοι οι κώδικες βρίσκονται στο τέλος της αναφοράς μας.

Ζήτημα 1

Σε αυτό το μέρος της άσκησης υλοποιήσαμε την λειτουργικότητα ενός $chat\ peer-to-peer$ με την χρήση των *sockets* και της εντολής *select*. Η όλη μας εφαρμογή βασίζεται πάνω στο πρωτόκολλο $TCP\ IP$ όπου και ο *host* αλλά και ο *client* ανταλλάσουν μυνήματα μέσω του *chat*. Στην υλοποίησή μας η σύνδεση γίνεται ανάμεσα σε έναν *host* και σε μόνο έναν *client*, θα μπορούσαμε εύκολα να γενικεύσουμε το πρόγραμμά μας προκειμένου ο *host* να μπορεί να δέχεται περισσότερες συνδέσεις με βάση τον κώδικα που μας έχει δοθεί μαζί με την εκφώνηση της άσκησης αλλά το κύριο ζητούμενο της άσκησης δεν είναι αυτό.

Οι κώδικες που υλοποιήσαμε αποτελούνται από 3 αρχεία: $socket-server.c, socket-client.c, socket-common.h$. Πρέπει να επισημάνουμε ότι υπάρχει αρκετό κομμάτι του κώδικα που είνα ικοινό τόσο για τον πελάτη όσο και για τον *host* που αυτή είναι η μέθοδος $chat-service$ που πραγματοποιεί την επικοινωνία μεταξύ αυτών των 2 στέλνοντας και λαμβάνοντας μυνήματα αφού έχει πραγματοποιηθεί η σύνδεση. Πάνω σε αυτά θα εξηγήσουμε διάφορες πολιτικές που ακολουθήσαμε για την ολοκλήρωση του ζητήματος αλλά και διάφορα σημεία προκλήσεων που αντιμετωπίσαμε:

Στο $socket-server.c$ βλέπουμε ότι ο *server* αρχικά περιμένει να ακούσει σε μία καινούρια σύνδεση και μπλοκάρει εκεί μέχρι να έρθει μία καινούρια σύνδεση στην πόρτα που αυτός ακούει. Όταν αυτή έρθει και μπορεί να μετατρέψει την διεύθυνση σε μορφή $IPNET$ τότε καλεί την μέθοδο $chat-service$ για να ξεκινήσει από την μεριά του την ανταλλαγή μυνημάτων.

Παρόμοια στο $socket-client.c$ βλέπουμε ότι ο *client* αρχικά προσπαθεί να συνδεθεί στην θύρα του *host* που έχει βρεί μέσω της κλήσης $gethostbyname()$. Όταν αυτό γίνει καλεί την μέθοδο $chat-service$ για να ξεκινήσει από την μεριά του την ανταλλαγή μυνημάτων.

Στο $socket-common.h$ έχουμε υλοποιήσει αρκετές $wrapper-functions$ προκειμένου να κάνουμε τον κώδικα πιο ευανάγνωστο και πιό λειτουργικό. Προφανώς και οι 2 μπορούν να έχουν τις λειτουργικότητες με τις οποίες εφοδιάσαμε την εφαρμογή μας που είναι αυτές που περιγράφονται στην εκφώνηση της άσκησης ($Quit, Help$ κτλπ). Συγκεκριμένα στην μέθοδο $chat-service$ που είναι και το ζουμί της υλοποίησής μας μπορούμε με την χρήση της FD_SET να εισάγουμε ένα κανάλι ανάμεσα στο σύνολο από το οποίο η $Select$ θα μπορεί να επιλέξει για να δει αν θα διαβάσει ή θα γράψει δηλαδή αν θα επιλέξει ανάμεσα στο *socket* που αντιστοιχεί σε αυτά που μας έχει γράψει ο απέναντι ή στο *stdin* δηλάδή σε αυτά που γράφουμε εμείς για να στείλουμε στον απέναντι, αντίστοιχα. Επιπλέον για αυτό τον λόγο χρησιμοποιούμε την FD_ISSET κατά την οποία η $Select()$ δεν μπλοκάρει και γι αυτό τον λόγο το πρόγραμμά μας συνεχώς μπορεί να λαμβάνει και να αποστέλνει μυνήματα χωρίς να κολλάει σε κάποια απότις δύο διαδικασίες.

Λεπτομέρειες Υλοποίησης: Παρατηρούμε ότι το μύνημα δεν το αποστέλουμε δηλαδή ουσιαστικά η κλάση της $Insist\_read\_write()$ δεν τερματίζει εκτός αν το τελευταίο γράμμα από αυτό που έχουμε γράψει είναι το $newline$ ή έχουμε φτάσει στο τέλος του $buffer$. Αυτό μας εξασφαλίζει ότι τα μυνήματά μας θα φτάνουνε συγχρονισμένα στον παραλήπτη που θέλουμε όλα μαζι και μάλιστα αυτά φτάνουν στον παραλήπτη σε πακέτα που είναι ίσου μεγέθους με το μέγεθος του $DATA\_SIZE = 256$. Φυσικά εκεί δημιουργείται ένα πρόβλημα καθώς το $stdout$ είναι τόσο το κανάλι το οποίο εμφανίζει τα μυνήματα του απέναντι που συνομιλούμε όσο και το κανάλι που εμφανίζεται το μύνημα μας (αυτό που γραφουμε). Φυσικά αυτό δεν ήταν ο σκοπός της άσκησης και γι αυτό τον λόγο δεν δώσαμε κάποια ιδιαίτερη λύση παρά μόνο όταν μας έρχεται κάποιο μύνημα να διαγράφεται η υπάρχουσα γραμμή του $stdout$ δηλαδή αυτή που γράφουμε χωρίς να επηρεάζεται προφανώς αυτό που θα στείλουμε τελικά στον απέναντι σαν τελικό μήνυμα.

Ζήτημα 2

Σε αυτό το μέρος της άσκησης εισηγάγαμε δυναμικά το $module$ του $cryptodev - linux$ στον πυρήνα μας προκειμένου να τον χρησιμοποιήσουμε για την κωδικοποίηση των μυνημάτων μας. Αυτό μπορούμε να το επιτύχουμε με απευθείας κλήσεις εισαγωγής εξαγωγής δεδομένων στην κρυπτογραφική μας συσκευή $/dev/crypto$. Το μοντέλο κρυπτογραφίας βασίζεται σε προσυμφωνημένο κλειδί (που φυσικά ανταλλάχτηκε μέσω του "$red - telephone$").

Προφανώς η υλοποίησή μας βασίζεται στο ζήτημα 1 μόνο που αυτή την φορά κρυπτογραφούμε και τα μηνύματα στην περίπτωση που ο χρήστης το ζητάει (με ειδικό $flag$). Όλες οι συναρτήσεις είναι οι ίδιες απλώς πριν σταλούν τα μηνύματα και μόλις παραληφθούν περνούν από το στάδιο της κρυπτογράφησης και της.

Λεπτομέρειες Υλοποίησης: Προκειμένου να είμαστε απολύτως σωστοί στην κρυπτογράφησή μας και για να μην επιτρέπουμε σε έναν κακόβουλο $l33t - h4ck3r$ ο οποίος θέλει να κλέψει το key της συνεδρίας που έχουμε εγκαταστήσει κάθε φορά πριν στείλουμε το μύνημα γεμίζουμε με τυχαία δεδομένα όλο τον $buffer$ προκειμένου να μην υπάρχει τρόπος να αποκαλύψουμε μέρος του κλειδιού σε κάποιον επιτιθέμενο στο $cipher - text$ που στέλνουμε. Γεγονός που είναι πολύ επικίνδυνο ιδίως όταν χρησιμοποιούμε $zero - padding$ τεχνικές.

Ζήτημα 3

Ο σκοπός αυτής της άσκησης είναι να υλοποιήσουμε τον οδηγό μιας εικονικής συσκευής $cryptodev$, με βοήθεια του πλαισίου $VirtIO$, ο οποίος ουσιαστικά προσφέρει στον χρήστη πρόσβαση στον οδηγό του $Host$, επιταχύνοντας έτσι την διαδικάσια.

Η υλοποίηση του οδηγού ουσιαστικά χωρίζεται σε δύο τμήματα, στον κώδικα του $host$ και στον κώδικα του $guest$. Στην πλευρά του $guest$ έπρεπε να υλοποιηθεί το $interface$ με όλες τις κλήσεις συστήματος που μπορεί να κάνει ο χρήστης στην συσκευή. Ουσιαστικά το κομμάτι στον $guest$ είναι υπεύθυνο για το πέρασμα των δεδομένων στον $host$ και για την διατήρηση κάποιου $state$.

Η ευθύνη της πλευράς του $host$ είναι να παραλάβει σωστά τα δεδομένα από τον

guest και να τα διαμορφώσει κατάλληλα ώστε να καλέσει τον οδηγό *crypto* και να επιστρέψει τα αποτελέσματα του σωστά μορφοποιημένα πίσω στον *guest*.

Τα δύο κομμάτια είναι συνδεδεμένα μεταξύ τους με τη βοήθεια της δομής *virtqueue* η οποία ουσιαστικά αποτελέι δίαυλο επικοινωνίας μεταξύ $guest - host$.

Λεπτομέρειες υλοποίησης

Στην πλευρά του *guest* χρειάστηκε να αλλάξουμε μόνο το αρχείο $crypto - chrdev.c$ και να προσθέσουμε ένα *spinlock* στην δομη $crypto_device$ στο *crypto.h*. Η δομή του $crypto - chrdev.c$ είναι τυπική δομή οδηγού αφου περιέχει όλες τις συναρτήσεις διεπαφής $open, close, ioctl$. Ουσιαστικά η διαφορά έγκειται στο ότι σε αυτές τις συναρτήσεις δημιουργήσαμε *scatterlists* τα οπόια και γεμίσαμε με τα δεδομένα του χρήστη για να μπορέσουμε να τα στείλουμε στον *host*. Ένα σημαντικό σημείο είναι ότι για την μεταφορά δεδομένων στον *host* χρειάστηκε να κάνουμε $deep - copy$ τις δομές , αφού οι δείκτες δεν περιέχουν την σωστή τιμή όταν τους περάσεις σε άλλο $address - space$ δηλαδή σε διαφορετική διευθυνσιοδότηση οι δείκτες διευθύνσεων που είχαμε από το προηγούμενο επίπεδο δεν παίζουν κανέναν ρόλο στο τωρινό. Επίσης χρειάστηκε να βάλουμε *spinlocks* γύρω από το σημείο που στέλνουμε δεδομένα και περιμένουμε απάντηση έτσι ώστε να αποφευχθούν πιθανά $race - conditions$ μεταξύ διαφορέτικων ανοιχτών διεργασιών που έχουν προκύψει από κοινή διακλάδωση $fork$ και επηρεάζουν το ίδιο αρχείο συσκευής.

Για το τμήμα του *host* κάναμε παρόμοια δουλειά αφού μετά την λήψη δεδομένων από τον *quest* η πλευρά του *host* ¨επανασυναρμολογεί' τις δομές δεδομένων και καλεί το $crypto - device$. Όταν παίρνει απάντηση την μορφοποιεί κατάλληλα και την στέλνει ξανά στον *guest*.

Γενικά δεν συναντήσαμε κάποια ιδιαίτερη δυσκολία για την δημιουργία του οδηγού εκτός από το τμήμα που χρειάστηκε να κάνουμε $deep - copy$ τα δεδομένα όπου χρειάστηκε να κάνουμε αρκετή ώρα *debugging* για να λύσουμε κάποια απρόσεκτα λάθη που δεν επέτρεπαν το σωστό πέρασμα των δεδομένων.

Παρατήρηση: Τα 3 αρχεία κώδικα που χρειάστηκε να επεξεργαστούμε βρίσκονται στο παράρτημα της εργασίας

C Codes

socket-server.c

```c
#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 <sys/select.h>

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

#include "socket-common.h"

void chat_close(int newsd){
  /* Make sure we don't leak open files */
  if (close(newsd) < 0)
    err_sys("close");
}


/* Convert a buffer to upercase */
void toupper_buf(char *buf, size_t n)
{
  size_t i;

  for (i = 0; i < n; i++)
    buf[i] = toupper(buf[i]);
```

```
34 }
35
36
37 int main(int argc, char *argv[])
38 {
39   char buf[100];
40   char addrstr[INET_ADDRSTRLEN];
41   int sd, newsd;
42   int crypted;
43   ssize_t n;
44   socklen_t len;
45   struct sockaddr_in sa;
46
47
48   if (argc > 2) {
49                 fprintf(stderr, "Usage: %s [-c]\n", argv
                        [0]);
50                 exit(1);
51         }
52
53         crypted = 0;
54         if(argc == 2){
55                 crypted = 1;
56         }
57
58
59
60   /* TODO CHeck if it should be here */
61   /* Make sure a broken connection doesn't kill us */
62   signal(SIGPIPE, SIG_IGN);
63
64   /* Create TCP/IP socket, used as main chat channel */
65   if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
66     perror("socket");
67     exit(1);
68   }
69   fprintf(stderr, "Created TCP socket\n");
70
71   /* Bind to a well-known port */
72   memset(&sa, 0, sizeof(sa));
73   sa.sin_family = AF_INET;
74   sa.sin_port = htons(TCP_PORT);
75   sa.sin_addr.s_addr = htonl(INADDR_ANY);
76   if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
77     perror("bind");
78     exit(1);
```

```
79    }
80    fprintf(stderr, "Bound TCP socket to port %d\n",
          TCP_PORT);
81
82    /* Listen for incoming connections */
83    if (listen(sd, TCP_BACKLOG) < 0) {
84      perror("listen");
85      exit(1);
86    }
87
88
89
90    fprintf(stderr, "Waiting for an incoming connection...\
          n");
91
92    /* Accept an incoming connection */
93    len = sizeof(struct sockaddr_in);
94    if ((newsd = accept(sd, (struct sockaddr *)&sa, &len))
          < 0) {
95      perror("accept");
96      exit(1);
97    }
98    if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(
          addrstr))) {
99      perror("could not format IP address");
100     exit(1);
101   }
102   fprintf(stderr, "Incoming connection from %s:%d\n",
103     addrstr, ntohs(sa.sin_port));
104
105   chat_service(newsd, crypted);
106
107   /* TODO
108    * Maybe shutdown the open sockets
109    */
110
111   // Elpizw na mhn ftasei edw
112
113   /* Loop forever, accept()ing connections */
114   for (;;) {
115     fprintf(stderr, "Waiting for an incoming connection
          ...\n");
116
117     /* Accept an incoming connection */
118     len = sizeof(struct sockaddr_in);
```

```
119     if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)
            ) < 0) {
120       perror("accept");
121       exit(1);
122     }
123     if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof
            (addrstr))) {
124       perror("could not format IP address");
125       exit(1);
126     }
127     fprintf(stderr, "Incoming connection from %s:%d\n",
128       addrstr, ntohs(sa.sin_port));
129
130     /* We break out of the loop when the remote peer goes
            away */
131     for (;;) {
132       n = read(newsd, buf, sizeof(buf));
133       if (n <= 0) {
134         if (n < 0)
135           perror("read from remote peer failed");
136         else
137           fprintf(stderr, "Peer went away\n");
138         break;
139       }
140       toupper_buf(buf, n);
141       if (insist_write(newsd, buf, n) != n) {
142         perror("write to remote peer failed");
143         break;
144       }
145     }
146     /* Make sure we don't leak open files */
147     if (close(newsd) < 0)
148       perror("close");
149   }
150
151   /* This will never happen */
152   return 1;
153 }
```

socket-client.c

```
1
2 /*
3  * socket-client.c
```

```
 * Simple TCP/IP communication using sockets
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 */

#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 <sys/select.h>

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

#include "socket-common.h"

void chat_close(int sd){

  /*
   * TODO Call it only when the user asks it
   * TODO Wrapper
   * Let the remote know we're not going to write anything else.
   * Try removing the shutdown() call and see what happens.
   */
  if (shutdown(sd, SHUT_WR) < 0)
    err_sys("shutdown");
}


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

```
48
49
50    if (argc < 3 || argc > 4) {
51      fprintf(stderr, "Usage: %s hostname port [-c]\n",
            argv[0]);
52      exit(1);
53    }
54
55    hostname = argv[1];
56    port = atoi(argv[2]); /* Needs better error checking */
57    crypted = 0;
58    if(argc == 4){
59      crypted = 1;
60    }
61
62
63    /* Create TCP/IP socket, used as main chat channel */
64    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
65      perror("socket");
66      exit(1);
67    }
68    fprintf(stderr, "Created TCP socket\n");
69
70    /* Look up remote hostname on DNS */
71    if ( !(hp = gethostbyname(hostname))) {
72      printf("DNS lookup failed for host %s\n", hostname);
73      exit(1);
74    }
75
76    /* Connect to remote TCP port */
77    sa.sin_family = AF_INET;
78    sa.sin_port = htons(port);
79    memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(struct
          in_addr));
80    fprintf(stderr, "Connecting to remote host... ");
          fflush(stderr);
81    if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) <
          0) {
82      perror("connect");
83      exit(1);
84    }
85    fprintf(stderr, "Connected.\n");
86
87
88
89
```

```
90
91      /*
92       * TODO
93       * Implement some testing module that shows what is
           being typed on the cable
94       * See the exercise guide and the commands netstat,
           tcpdump, nc, telnet
95       */
96
97      /*
98       * Read
99       * TODO
100      * 1) Check/Ask if it is possible for the Read to
           return less than
101      *    what the user has written
102      * 2) There is one TODO inside -common.h
103      *
104      */
105
106     chat_service(sd, crypted);
107
108
109     // Hope we never reach it
110     return 0;
111 }
```

socket-common.c

```
1
2  /*
3   * socket-common.h
4   *
5   * Simple TCP/IP communication using sockets
6   *
7   * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
8   */
9
10 #ifndef _SOCKET_COMMON_H
11 #define _SOCKET_COMMON_H
12
13 /* Compile-time options */
14 #define TCP_PORT     35001
15 #define TCP_BACKLOG 5
16 #define STR_BUF_LENGTH 255
```

```
17
18 #define HELLO_THERE "Hello there!"
19
20 // Crappy MIN, MAX implementation
21 #define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
22 #define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
23
24 #endif /* _SOCKET_COMMON_H */
25
26
27
28 #include <fcntl.h>
29 #include <sys/ioctl.h>
30 #include <sys/stat.h>
31 #include <crypto/cryptodev.h>
32
33 #define DATA_SIZE       256
34 #define BLOCK_SIZE      16
35 #define KEY_SIZE   16   /* AES128 */
36
37 struct Daten{
38   unsigned char    in[DATA_SIZE],
39       encrypted[DATA_SIZE],
40       decrypted[DATA_SIZE],
41       iv[BLOCK_SIZE],
42       key[KEY_SIZE];
43 };
44
45
46
47 /* Insist until all of the data has been read */
48 ssize_t insist_read(int fd, void *buf, size_t cnt)
49 {
50         ssize_t ret;
51         size_t orig_cnt = cnt;
52
53         while (cnt > 0) {
54     ret = read(fd, buf, cnt);
55     if (ret <= 0)
56                         return ret;
57     buf += ret;
58                 cnt -= ret;
59         }
60
61         return orig_cnt;
62 }
```

```
63
64
65  static int decrypt(int cfd, struct session_op *sess,
        struct Daten *data){
66          struct crypt_op cryp;
67          memset(&cryp, 0, sizeof(cryp));
68          /*
69           * Decrypt data.encrypted to data.decrypted
70           */
71          cryp.ses = sess->ses;
72          cryp.len = DATA_SIZE;
73          //cryp.len = sizeof(data->in);
74          cryp.iv = data->iv;
75          cryp.src = data->encrypted;
76          cryp.dst = data->decrypted;
77          cryp.op = COP_DECRYPT;
78          if (ioctl(cfd, CIOCCRYPT, &cryp)) {
79                  perror("ioctl(CIOCCRYPT)");
80                  return 1;
81          }
82          return 0;
83  }
84  static int encrypt(int cfd, struct session_op *sess,
        struct Daten *data){
85
86          struct crypt_op cryp;
87          memset(&cryp, 0, sizeof(cryp));
88          /*
89           * Encrypt data.in to data.encrypted
90           */
91          cryp.ses = sess->ses;
92    cryp.len = DATA_SIZE;
93          //cryp.len = sizeof(data->in);
94          cryp.iv = data->iv;
95          cryp.len = sizeof(data->in);
96          cryp.src = data->in;
97          cryp.dst = data->encrypted;
98          cryp.iv = data->iv;
99          cryp.op = COP_ENCRYPT;
100
101         if (ioctl(cfd, CIOCCRYPT, &cryp)) {
102                 perror("ioctl(CIOCCRYPT)");
103                 return 1;
104         }
105         return 0;
106 }
```

```
107
108 static int crypto_initialize(int cfd, struct session_op *
    sess, unsigned char *key){
109         /*
110          * Get crypto session for AES128
111          */
112         sess->cipher = CRYPTO_AES_CBC;
113         sess->keylen = KEY_SIZE;
114         sess->key = key;
115
116         if (ioctl(cfd, CIOCGSESSION, sess)) {
117                 perror("ioctl(CIOCGSESSION)");
118                 return 1;
119         }
120         return 0;
121 }
122
123 static int crypto_close(int cfd, struct session_op sess){
124
125         /* Finish crypto session */
126         if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
127                 perror("ioctl(CIOCFSESSION)");
128                 return 1;
129         }
130         return 0;
131
132 }
133
134
135 static int fill_urandom_buf(unsigned char *buf, size_t
    cnt)
136 {
137         int crypto_fd;
138         int ret = -1;
139
140         crypto_fd = open("/dev/urandom", O_RDONLY);
141         if (crypto_fd < 0)
142                 return crypto_fd;
143
144         ret = insist_read(crypto_fd, buf, cnt);
145         close(crypto_fd);
146
147         return ret;
148 }
149
150
```

```
151 static int our_test_crypto(int cfd)
152 {
153         int i = -1;
154         struct session_op sess;
155         struct crypt_op cryp;
156         unsigned char *message;
157         unsigned char *key;
158         struct Daten data;
159         memset(&sess, 0, sizeof(sess));
160         memset(&cryp, 0, sizeof(cryp));
161
162
163         message = "KALLAS\n";
164         for (i=0; i<7; i++){
165                 data.in[i] = message[i];
166         }
167         printf("The inital Message is %s", data.in );
168
169         message = "KEYS";
170         key = "LOCK";
171         for(i=0; i<4; i++){
172                 data.key[i]=key[i];
173                 data.iv[i]=key[i];
174         }
175
176
177         printf("\nOriginal data:\n");
178         for (i = 0; i < DATA_SIZE; i++)
179                 printf("%x", data.in[i]);
180         printf("\n");
181
182         /*
183          * Get crypto session for AES128
184          */
185         crypto_initialize(cfd, &sess, data.key);
186
187         /*
188          * Encrypt data.in to data.encrypted
189          */
190         encrypt(cfd, &sess, &data);
191
192         printf("\nEncrypted data:\n");
193         for (i = 0; i < DATA_SIZE; i++) {
194                 printf("%x", data.encrypted[i]);
195         }
196         printf("\n");
```

```
197
198             /*
199              * Decrypt data.encrypted to data.decrypted
200              */
201             decrypt(cfd, &sess, &data);
202
203             printf("\nDecrypted data:\n");
204             for (i = 0; i < DATA_SIZE; i++) {
205                     printf("%x", data.decrypted[i]);
206             }
207             printf("\n");
208
209             /* Verify the result */
210             if (memcmp(data.in, data.decrypted, sizeof(data.
                    in)) != 0) {
211                     fprintf(stderr, "\nFAIL: Decrypted and
                           original data differ.\n");
212                     return 1;
213             } else
214                     fprintf(stderr, "\nTest passed.\n");
215
216             /* Finish crypto session */
217             crypto_close(cfd, sess);
218             return 0;
219     }
220
221
222
223
224
225     /*
226      * Wrapper functions
227      */
228     void err_sys(char * msg)
229     {
230       printf("Error message: %s\n", msg);
231       exit(1);
232     }
233
234
235     int Read(int fd, void *buf, size_t count)
236     {
237       int n;
238
239       if( (n = read(fd, buf, count)) < 0 )
240         err_sys("read error");
```

```
241   return n;
242 }
243
244 int Select(int ndfs, fd_set *rdfs, fd_set *wrfs, fd_set *
      exfs, struct timeval *timeout)
245 {
246   int n;
247
248   n = select(ndfs, rdfs, wrfs, exfs, timeout);
249
250   if(n == -1){
251     err_sys("select_error");
252   }else if(n){
253     return n;
254   }else{
255     return n;
256   }
257
258   // Na mhn ftasei edw
259   return -1;
260
261 }
262
263
264 /* Insist until all of the data has been written */
265 ssize_t insist_write(int fd, const void *buf, size_t cnt)
266 {
267   ssize_t ret;
268   size_t orig_cnt = cnt;
269
270   while (cnt > 0) {
271           ret = write(fd, buf, cnt);
272           if (ret < 0)
273                   return ret;
274           buf += ret;
275           cnt -= ret;
276   }
277
278   return orig_cnt;
279 }
280
281 ssize_t Insist_write(int fd, const void *buf, size_t cnt)
282 {
283   ssize_t n;
284   if (  (n = insist_write(fd, buf, cnt)) != cnt) {
285     err_sys("insist_write");
```

```
286   }
287   return n;
288
289 }
290
291
292 /*
293  * Insist on reading the whole sd message
294  */
295 ssize_t Insist_crypted_read_write(int fd, char *buf, int
        out_fd, int cfd ,struct session_op *sess, struct Daten
        *data)
296 {
297
298
299   int n = 0;
300   int temp;
301   int i;
302
303   temp = insist_read(fd,buf,DATA_SIZE);
304
305   if (temp == 0) {
306     printf("The other chatter exited GORBATSOF\n");
307     exit(0);
308   }
309
310   /*
311          * Decrypt data.encrypted to data.decrypted
312          */
313   for (i=0; i<DATA_SIZE; i++){
314     data->encrypted[i] = buf[i];
315   }
316
317         decrypt(cfd, sess, data);
318
319
320   for (i=0; i<DATA_SIZE; i++){
321      buf[i] = data->decrypted[i];
322   }
323
324   // printf("The decrypted Message is %s", data->
        decrypted );
325   // printf("The decrypted Message is %s", buf );
326
327   n = Insist_write(out_fd, buf, strlen(buf));
328
```

```
329   return n;
330 }
331
332
333 /*
334  * Insist on reading the whole sd message
335  */
336 ssize_t Insist_read_write(int fd, char *buf, int out_fd)
337 {
338
339
340   int n = 0;
341   int temp;
342
343   /* Read answer and write it to standard output */
344   for (;;) {
345     temp = read(fd, buf, STR_BUF_LENGTH);
346
347     if (temp < 0)
348       err_sys("read");
349
350     if (temp == 0) {
351       printf("The other chatter exited GORBATSOF\n");
352       exit(0);
353     }
354
355     n += Insist_write(out_fd, buf, temp);
356
357     // An to teleutaio gramma einai \n h o buffer einai
          gematos teleiwsame
358     if(n == STR_BUF_LENGTH || buf[temp-1] == '\n')
359       break;
360   }
361   return n;
362 }
363
364 void chat_close(int f);
365
366 void chat_commands(char *buf, int fd){
367   if(!strcmp(buf,"-h\n") || !strcmp(buf,"-H\n")){
368     printf("Hello to my little friends, I came here to
          connect your souls\n");
369     printf("                              GORBATSOF
                                          \n");
370     printf("List of commands:\n");
371     printf("\thelp (-h -H)\n");
```

```
372       printf("\tquit (-q -Q)\n");
373     }
374     else if(!strcmp(buf,"-q\n") || !strcmp(buf,"-Q\n")){
375       printf("Exiting GORBATSOF...\n");
376       chat_close(fd);
377     }
378     else printf("for help about GORBATSOF commands please
          type -h or -H\n");
379 }
380
381
382
383
384
385 /*
386  * Chat service
387  */
388
389 int chat_service(int sd, int crypted)
390 {
391
392   fd_set rdfs;
393   int ret;
394   char buf[STR_BUF_LENGTH + 1];
395   int cfd, i;
396
397         struct session_op sess;
398         unsigned char *key;
399         struct Daten data;
400         memset(&sess, 0, sizeof(sess));
401
402         cfd = open("/dev/crypto", O_RDWR);
403         if (cfd < 0) {
404                 perror("open(/dev/crypto)");
405                 return 1;
406         }
407
408
409   FD_ZERO(&rdfs);
410   FD_SET(0, &rdfs);
411   FD_SET(sd, &rdfs);
412
413
414   /*
415    * Crypto Session initialize
416    */
```

```
417   key = "LOCK\0";
418
419         for(i=0; i<DATA_SIZE; i++){
420                 data.key[i]=0;
421                 data.iv[i]=0;
422         }
423
424   // Deadly Warning -- Not tested
425
426         for(i=0; i<strlen(key); i++){
427                 data.key[i]=key[i];
428                 data.iv[i]=key[i];
429         }
430         crypto_initialize(cfd, &sess, data.key);
431
432
433   for( ; ; )
434   {
435     /*
436      * Select between stdin and socket
437      */
438     printf("Me: ");
439     fflush(0);
440     FD_SET(0, &rdfs);
441     FD_SET(sd, &rdfs);
442     ret = Select(sd+1, &rdfs, NULL, NULL, NULL);
443     if(ret)
444     {
445       /*
446        * Stdin
447        */
448       if(FD_ISSET(0,&rdfs))
449       {
450         ret = Read(0,buf,STR_BUF_LENGTH);
451         buf[ret] = '\0';
452         if(buf[0]=='-'){
453           chat_commands(buf, sd);
454         }
455         else
456         {
457           if(crypted)
458           {
459
460             /*
461              * STEVi
462              */
```

```
463             if (fill_urandom_buf(data.in, DATA_SIZE) < 0)
                    {
464               perror("getting data from /dev/urandom\n");
465               return 1;
                      }
466
467
468             for (i=0; i<strlen(buf); i++){
469               data.in[i] = buf[i];
470             }
471             data.in[strlen(buf)] = '\0';
472
473             /*
474              * Encrypt data.in to data.encrypted
475              */
476             encrypt(cfd, &sess, &data);
477
478             for (i=0; i<DATA_SIZE; i++){
479               buf[i] = data.encrypted[i];
480             }
481             Insist_write(sd, buf, DATA_SIZE);
482           }
483           else{
484             Insist_write(sd, buf, strlen(buf));
485           }
486         }
487         //FD_CLR(0,&rdfs);
488       }
489
490       /*
491        * Socket
492        */
493       else if(FD_ISSET(sd,&rdfs))
494       {
495         printf("\33[2K\r");
496         printf("Other: ");
497         /*
498          * TODO
499          * 1) If you overflow the buffer you should not
                  print other
500          * 2) reprint Me: blahblah sometimes
501          */
502         fflush(0);
503         if(crypted){
504           Insist_crypted_read_write(sd, buf, 0 , cfd, &
                  sess, &data);
505         }
```

```
506             else{
507               Insist_read_write(sd, buf, 0 );
508             }
509
510             //FD_CLR(sd, &rdfs);
511           }
512           else
513             err_sys("Kanena fd den einai set :'(");
514         }
515       }
516
517       return 0;
518     }
```

virtio-crypto.c

```
1   /*
2    * Virtio Crypto Device
3    *
4    * Implementation of virtio-crypto qemu backend device.
5    *
6    * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
7    * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
8    *
9    */
10
11  #include <qemu/iov.h>
12  #include "hw/virtio/virtio-serial.h"
13  #include "hw/virtio/virtio-crypto.h"
14  #include <sys/types.h>
15  #include <sys/stat.h>
16  #include <fcntl.h>
17  #include <sys/ioctl.h>
18  #include <crypto/cryptodev.h>
19
20  #define IV_LENGTH 16
21
22  static uint32_t get_features(VirtIODevice *vdev, uint32_t
        features)
23  {
24          DEBUG_IN();
25          return features;
26  }
27
```

```
28 static void get_config(VirtIODevice *vdev, uint8_t *
       config_data)
29 {
30         DEBUG_IN();
31 }
32
33 static void set_config(VirtIODevice *vdev, const uint8_t
       *config_data)
34 {
35         DEBUG_IN();
36 }
37
38 static void set_status(VirtIODevice *vdev, uint8_t status
       )
39 {
40         DEBUG_IN();
41 }
42
43 static void vser_reset(VirtIODevice *vdev)
44 {
45         DEBUG_IN();
46 }
47
48 static void vq_handle_output(VirtIODevice *vdev,
       VirtQueue *vq)
49 {
50         int i;
51         VirtQueueElement elem;
52         unsigned int *syscall_type;
53         int dev_fd = -1;
54         unsigned int cmd;
55         struct session_op sess;
56         struct crypt_op cryp;
57         int host_error = 0;
58         int *close_fd;
59
60         DEBUG_IN();
61
62         if (!virtqueue_pop(vq, &elem)) {
63                 DEBUG("No item to pop from VQ :(");
64                 return;
65         }
66
67         DEBUG("I have got an item from VQ :)");
68
69         syscall_type = elem.out_sg[0].iov_base;
```

```
70          switch (*syscall_type) {
71          case VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN:
72                  DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN");
73                  /* ?? */
74
75                  /**
76                   * TODO
77                   * We suppose that the backend will do
                         everything as the guest does
78                   * For example: The guest want to open
                         crypto, then we open crypto
79                   * So because of that:
80                   *
81                   * Here goes an open to the crypto device
                         and
82                   * somehow we have to keep the file
                         descriptor and associate it with
83                   * the specific guest vq ????
84                   **/
85
86                  /**
87                   * Open the crypto device :)
88                   **/
89                  dev_fd = open("/dev/crypto", O_RDWR, 0);
90                  printf("Host: I opened '%d'\n", dev_fd);
91                  memcpy(elem.in_sg[0].iov_base, &dev_fd,
                        sizeof(dev_fd));
92                  break;
93
94          case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
95                  DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE")
                        ;
96
97                  close_fd = elem.out_sg[1].iov_base;
98
99                  if(close(*close_fd)){
100                         printf("I fucking failed, **
                               exits violently, in deep rage
                               **\n");
101                 }
102                 else{
103                         printf("Host: I closed '%d'\n", *
                               close_fd);
104                 }
105                 break;
106
```

```
107             case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
108                     DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL")
                            ;
109
110                     DEBUG("Potential Hazard!!");
111                     dev_fd = *((int *) elem.out_sg[1].
                            iov_base);
112                     DEBUG("Ok with Hazard!!");
113                     cmd = *((int *) elem.out_sg[2].iov_base);
114
115                     switch (cmd) {
116                     case CIOCGSESSION:
117                             memcpy(&sess, elem.out_sg[3].
                                    iov_base, sizeof(sess));
118
119                             /**
120                              * TODO
121                              * Handle those fucking errors
                                    !!!!!!!
122                              **/
123                             printf("Problemino\n");
124                             printf("Key: %lu\n", sess.key);
125
126
127                             if(sess.keylen) sess.key = elem.
                                    out_sg[4].iov_base;
128                             if(sess.mackeylen) sess.mackey =
                                    elem.out_sg[5].iov_base;
129                             printf("Key: %lu\n", sess.key);
130
131                             printf("Out num: %u\n", elem.
                                    out_num);
132                             printf("Iov base: %lu\n",  elem.
                                    out_sg[4].iov_base);
133                             printf("Sess key: %x\n", sess.key
                                    );
134
135
136                             for(i=0;i<24;i++){
137                                     printf("%x", sess.key[i])
                                            ;
138                             }
139
140
141                             DEBUG("Host: Calling CIOCGSESSION
                                    !!");
```

```
142                     if (ioctl(dev_fd, CIOCGSESSION, &
                            sess)) {
143                             perror("ioctl(
                                CIOCGSESSION)");
144                             host_error = 1;
145                             memcpy(elem.in_sg[1].
                                iov_base, &host_error,
                                 sizeof(host_error));
146                             goto out;
147                     }
148                     DEBUG("Host: Successful
                        CIOCGSESSION!!");
149
150                     memcpy(elem.in_sg[0].iov_base, &
                        sess, sizeof(sess));
151                     break;
152             case CIOCFSESSION:
153                     memcpy(&sess, elem.out_sg[3].
                        iov_base, sizeof(sess));
154
155                     if(sess.keylen) sess.key = elem.
                        out_sg[4].iov_base;
156                     if(sess.mackeylen) sess.mackey =
                        elem.out_sg[5].iov_base;
157
158                     DEBUG("Host: Calling CIOCFSESSION
                        !!");
159                     if (ioctl(dev_fd, CIOCFSESSION, &
                        sess)) {
160                             perror("ioctl(
                                CIOCFSESSION)");
161                             host_error = 1;
162                             memcpy(elem.in_sg[1].
                                iov_base, &host_error,
                                 sizeof(host_error));
163                             goto out;
164                     }
165                     DEBUG("Host: Successful
                        CIOCFSESSION!!");
166
167                     memcpy(elem.in_sg[0].iov_base, &
                        sess, sizeof(sess));
168                     break;
169             case CIOCCRYPT:
170                     memcpy(&cryp, elem.out_sg[3].
                        iov_base, sizeof(cryp));
```

```
171
172                         cryp.iv = elem.out_sg[4].iov_base
                                ;
173                         if(cryp.len) cryp.src = elem.
                                out_sg[5].iov_base;
174
175                         DEBUG("Host: Calling CIOCCRYPT!!"
                                );
176                         if (ioctl(dev_fd, CIOCCRYPT, &
                                cryp)) {
177                                 perror("ioctl(CIOCCRYPT)"
                                        );
178                                 host_error = 1;
179                                 memcpy(elem.in_sg[3].
                                        iov_base, &host_error,
                                         sizeof(host_error));
180                                 goto out;
181                         }
182                         DEBUG("Host: Successful CIOCCRYPT
                                !!");
183
184                         if(cryp.len){
185                                 memcpy(elem.in_sg[0].
                                        iov_base, cryp.dst,
                                        cryp.len);
186                                 memcpy(elem.in_sg[1].
                                        iov_base, cryp.mac,
                                        cryp.len);
187                         }
188
189                         memcpy(elem.in_sg[2].iov_base, &
                                cryp, sizeof(cryp));
190                         break;
191                 default:
192                         DEBUG("Unsupported ioctl command"
                                );
193                         break;
194                 }
195
196                 break;
197
198         default:
199                 DEBUG("Unknown syscall_type");
200         }
201
202
```

```
203 out:
204         virtqueue_push(vq, &elem, 0);
205         virtio_notify(vdev, vq);
206 }
207
208 static void virtio_crypto_realize(DeviceState *dev, Error
        **errp)
209 {
210     VirtIODevice *vdev = VIRTIO_DEVICE(dev);
211
212         DEBUG_IN();
213
214     virtio_init(vdev, "virtio-crypto", 13, 0);
215         virtio_add_queue(vdev, 128, vq_handle_output);
216 }
217
218 static void virtio_crypto_unrealize(DeviceState *dev,
        Error **errp)
219 {
220         DEBUG_IN();
221 }
222
223 static Property virtio_crypto_properties[] = {
224     DEFINE_PROP_END_OF_LIST(),
225 };
226
227 static void virtio_crypto_class_init(ObjectClass *klass,
        void *data)
228 {
229     DeviceClass *dc = DEVICE_CLASS(klass);
230     VirtioDeviceClass *k = VIRTIO_DEVICE_CLASS(klass);
231
232         DEBUG_IN();
233     dc->props = virtio_crypto_properties;
234     set_bit(DEVICE_CATEGORY_INPUT, dc->categories);
235
236     k->realize = virtio_crypto_realize;
237     k->unrealize = virtio_crypto_unrealize;
238     k->get_features = get_features;
239     k->get_config = get_config;
240     k->set_config = set_config;
241     k->set_status = set_status;
242     k->reset = vser_reset;
243 }
244
245 static const TypeInfo virtio_crypto_info = {
```

```
246     .name          = TYPE_VIRTIO_CRYPTO,
247     .parent        = TYPE_VIRTIO_DEVICE,
248     .instance_size = sizeof(VirtCrypto),
249     .class_init    = virtio_crypto_class_init,
250 };
251
252 static void virtio_crypto_register_types(void)
253 {
254     type_register_static(&virtio_crypto_info);
255 }
256
257 type_init(virtio_crypto_register_types)
```

crypto-chrdev.c

```
1  /*
2   * crypto-chrdev.c
3   *
4   * Implementation of character devices
5   * for virtio-crypto device
6   *
7   * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
8   * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
9   * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
10  *
11  */
12 #include <linux/cdev.h>
13 #include <linux/poll.h>
14 #include <linux/sched.h>
15 #include <linux/module.h>
16 #include <linux/wait.h>
17 #include <linux/virtio.h>
18 #include <linux/virtio_config.h>
19
20 #include "crypto.h"
21 #include "crypto-chrdev.h"
22 #include "debug.h"
23
24 #include "cryptodev.h"
25
26
27 #define IV_LENGTH 16
28 /*
29  * Global data
```

```
30  */
31 struct cdev crypto_chrdev_cdev;
32
33 /**
34  * Given the minor number of the inode return the crypto
        device
35  * that owns that number.
36  **/
37 static struct crypto_device *get_crypto_dev_by_minor(
        unsigned int minor)
38 {
39         struct crypto_device *crdev;
40         unsigned long flags;
41
42         debug("Entering");
43
44         spin_lock_irqsave(&crdrvdata.lock, flags);
45         list_for_each_entry(crdev, &crdrvdata.devs, list)
               {
46                 if (crdev->minor == minor)
47                         goto out;
48         }
49         crdev = NULL;
50
51 out:
52         spin_unlock_irqrestore(&crdrvdata.lock, flags);
53
54         debug("Leaving");
55         return crdev;
56 }
57
58 /*********************************
59  * Implementation of file operations
60  * for the Crypto character device
61  *********************************/
62
63 static int crypto_chrdev_open(struct inode *inode, struct
        file *filp)
64 {
65         int ret = 0;
66         int err;
67         unsigned int len;
68         struct crypto_open_file *crof;
69         struct crypto_device *crdev;
70         unsigned int syscall_type =
             VIRTIO_CRYPTO_SYSCALL_OPEN;
```

```
71          struct scatterlist syscall_type_sg,
                file_descriptor_sg,
72                              *sgs[2];
73          int host_fd = -1;
74          int num_out, num_in;
75
76          num_out = 0;
77          num_in = 0;
78
79          debug("Entering");
80
81          ret = -ENODEV;
82          if ((ret = nonseekable_open(inode, filp)) < 0)
83                  goto fail;
84
85          /* Associate this open file with the relevant
                crypto device. */
86          crdev = get_crypto_dev_by_minor(iminor(inode));
87          if (!crdev) {
88                  debug("Could not find crypto device with
                        %u minor",
89                          iminor(inode));
90                  ret = -ENODEV;
91                  goto fail;
92          }
93
94          crof = kzalloc(sizeof(*crof), GFP_KERNEL);
95          if (!crof) {
96                  ret = -ENOMEM;
97                  goto fail;
98          }
99          crof->crdev = crdev;
100         crof->host_fd = -1;
101         filp->private_data = crof;
102
103         /**
104          * We need two sg lists, one for syscall_type and
                one to get the
105          * file descriptor from the host.
106          **/
107         sg_init_one(&syscall_type_sg, &syscall_type,
                sizeof(syscall_type));
108         sgs[num_out++] = &syscall_type_sg;
109
110         sg_init_one(&file_descriptor_sg, &host_fd, sizeof
                (host_fd));
```

```
111         sgs[num_out + num_in++] = &file_descriptor_sg;;
112
113         spin_lock(&crdev->lock);
114
115         /**
116          * Wait for the host to process our data.
117          **/
118         err = virtqueue_add_sgs(crdev->vq, sgs, num_out,
                num_in,
119                                 &syscall_type_sg,
                                    GFP_ATOMIC);
120         virtqueue_kick(crdev->vq);
121         while (virtqueue_get_buf(crdev->vq, &len) == NULL
                )
122                 /* do nothing */;
123
124
125         spin_unlock(&crdev->lock);
126
127         debug("Host opened: '%d'", host_fd);
128
129         /* If host failed to open() return -ENODEV. */
130         if(host_fd < 0){
131                 debug("Host failed to open crypto device"
                        );
132                 ret = -ENODEV;
133                 goto fail;
134         }
135
136         /* Host was succesful opening */
137         crof->host_fd = host_fd;
138
139 fail:
140         debug("Leaving");
141         return ret;
142 }
143
144 static int crypto_chrdev_release(struct inode *inode,
        struct file *filp)
145 {
146         int ret = 0;
147         unsigned int len;
148         int err;
149         struct crypto_open_file *crof = filp->
                private_data;
150         struct crypto_device *crdev = crof->crdev;
```

```
151         unsigned int syscall_type =
                VIRTIO_CRYPTO_SYSCALL_CLOSE;
152         int host_fd;
153         struct scatterlist syscall_type_sg,
                file_descriptor_sg,
154                           *sgs[2];
155         int num_out, num_in;
156
157         num_out = 0;
158         num_in = 0;
159
160         debug("Entering");
161
162         if(!crof){
163                 debug("Private data not exists :(");
164                 ret = -1;
165                 goto out;
166         }
167
168
169         host_fd = crof->host_fd;
170
171         /**
172          * We need two sg lists, one for syscall_type and
                one to give the
173          * file descriptor from the host.
174          **/
175         sg_init_one(&syscall_type_sg, &syscall_type,
                sizeof(syscall_type));
176         sgs[num_out++] = &syscall_type_sg;
177
178         sg_init_one(&file_descriptor_sg, &host_fd, sizeof
                (host_fd));
179         sgs[num_out++] = &file_descriptor_sg;;
180
181
182
183
184         spin_lock(&crdev->lock);
185         /**
186          * Wait for the host to process our data.
187          **/
188         err = virtqueue_add_sgs(crdev->vq, sgs, num_out,
                num_in,
189                                 &syscall_type_sg,
                                    GFP_ATOMIC);
```

```
190         virtqueue_kick(crdev->vq);
191         while (virtqueue_get_buf(crdev->vq, &len) == NULL
                )
192                 /* do nothing */;
193
194         spin_unlock(&crdev->lock);
195         /**
196          * TODO
197          * Maybe implement a close() check
198          * WARNING: Close can fail : I/O, Signal, or
                invalid fd
199          **/
200         debug("Host: closed file '%d'", host_fd);
201
202
203         kfree(crof);
204 out:
205         debug("Leaving");
206         return ret;
207
208 }
209
210 static long crypto_chrdev_ioctl(struct file *filp,
        unsigned int cmd,
211                                 unsigned long arg)
212 {
213         int i;
214         long ret = 0;
215         int err;
216         int host_error = 0;
217         struct crypto_open_file *crof = filp->
                private_data;
218         struct crypto_device *crdev = crof->crdev;
219         struct virtqueue *vq = crdev->vq;
220         struct scatterlist syscall_type_sg,
                file_descriptor_sg, cmd_sg, data_out_sg,
                data_in_sg,*sgs[10];
221         struct scatterlist  sess_key_sg, /* out */
222                             sess_mackey_sg, /* out */
223                             crypt_src_sg, /* out */
224                             crypt_dst_sg, /* in */
225                             crypt_mac_sg, /* in */
226                             crypt_iv_sg, /* out */ /*
                                    Considering that iv does
                                    not change from host */
227                             error_sg; /* in */
```

```
228
229         unsigned char *key = NULL, *mackey = NULL, *src =
                NULL, *dst=NULL, *mac=NULL, *iv=NULL;
230
231 #define MSG_LEN 100
232         unsigned char output_msg[MSG_LEN], input_msg[
                MSG_LEN];
233         unsigned int num_out, num_in,
234                      syscall_type =
                             VIRTIO_CRYPTO_SYSCALL_IOCTL,
235                      len;
236         int host_fd;
237         /* Essential crypto structs */
238         struct session_op sess_out, sess_in;
239         struct crypt_op cryp_out, cryp_in;
240
241         debug("Entering");
242
243
244
245         num_out = 0;
246         num_in = 0;
247
248         host_fd = crof->host_fd;
249
250         /**
251          *  These are common to all ioctl commands.
252          **/
253         sg_init_one(&syscall_type_sg, &syscall_type,
                sizeof(syscall_type));
254         sgs[num_out++] = &syscall_type_sg;
255
256         sg_init_one(&file_descriptor_sg, &host_fd, sizeof
                (host_fd));
257         sgs[num_out++] = &file_descriptor_sg;;
258
259         sg_init_one(&cmd_sg, &cmd, sizeof(cmd));
260         sgs[num_out++] = &cmd_sg;
261
262
263         /**
264          * TODO
265          * 1) Think of not busy waiting until data comes
                    at the buffer, instead
266          *    be notified by host with virtio_notify (
                    Check Documentation)
```

```
267             * 2) Find how does the guest know about
                    virtio_notify ??
268             *
269             * 3) Error handling
270             *
271             * 4) Remember to kfree
272             **/
273
274
275            /**
276             *  Add all the cmd specific sg lists.
277             **/
278            switch (cmd) {
279            case CIOCGSESSION:
280                    debug("CIOCGSESSION");
281
282                    /**
283                     * Copy the sess struct
284                     **/
285                    err = copy_from_user(&sess_out, (struct
                            session_op *)arg, sizeof(sess_out));
286
287                    debug("Succesfully copied data_out");
288
289                    sg_init_one(&data_out_sg, &sess_out,
                            sizeof(sess_out));
290                    sgs[num_out++] = &data_out_sg;
291
292                    debug("Key pointer: %lu", (unsigned long)
                            sess_out.key);
293                    debug("Mackey pointer: %lu", (unsigned
                            long)sess_out.mackey);
294
295                    /**
296                     * Initialize key sgs
297                     **/
298                    key = kzalloc(sess_out.keylen, GFP_KERNEL
                            );
299                    if (!key) {
300                            ret = -ENOMEM;
301                            goto out;
302                    }
303                    err = copy_from_user(key, sess_out.key,
                            sess_out.keylen);
304                    debug("Keylen: %d\n", sess_out.keylen);
305                    for(i=0;i<sess_out.keylen; i++){
```

```
306                     debug("%x", *(key+i));
307             }
308             debug("\n");
309
310             mackey = kzalloc(sess_out.mackeylen,
                    GFP_KERNEL);
311             if (!mackey) {
312                     ret = -ENOMEM;
313                     goto out;
314             }
315             err = copy_from_user(mackey, sess_out.
                    mackey, sess_out.mackeylen);
316
317             debug("Succesfully copied keys");
318
319             if(sess_out.keylen){
320                     sg_init_one(&sess_key_sg, key,
                            sess_out.keylen);
321                     sgs[num_out++] = &sess_key_sg;
322             }
323             else{
324                     num_out++;
325             }
326
327             /*
328             if(sess_out.mackeylen){
329                     sg_init_one(&sess_mackey_sg,
                            mackey, sess_out.mackeylen);
330                     sgs[num_out++] = &sess_mackey_sg;
331             }
332             else{
333                     num_out++;
334             }
335             */
336
337             sg_init_one(&data_in_sg, &sess_in, sizeof
                    (sess_in));
338             sgs[num_out + num_in++] = &data_in_sg;
339             break;
340
341     case CIOCFSESSION:
342             debug("CIOCFSESSION");
343
344             /**
345              * Copy the sess struct
346              **/
```

```
347                 err = copy_from_user(&sess_out, (struct
                        session_op *)arg, sizeof(sess_out));
348
349
350                 /**
351                  * Initialize key sgs
352                  **/
353                 key = kzalloc(sess_out.keylen, GFP_KERNEL
                        );
354                 if (!key) {
355                         ret = -ENOMEM;
356                         goto out;
357                 }
358                 err = copy_from_user(key, sess_out.key,
                        sess_out.keylen);
359                 debug("Keylen: %d\n", sess_out.keylen);
360                 for(i=0;i<sess_out.keylen; i++){
361                         debug("%x", *(key+i));
362                 }
363                 debug("\n");
364
365                 mackey = kzalloc(sess_out.mackeylen,
                        GFP_KERNEL);
366                 if (!mackey) {
367                         ret = -ENOMEM;
368                         goto out;
369                 }
370                 err = copy_from_user(mackey, sess_out.
                        mackey, sess_out.mackeylen);
371
372                 debug("Succesfully copied keys");
373
374                 if(sess_out.keylen){
375                         sg_init_one(&sess_key_sg, key,
                                sess_out.keylen);
376                         sgs[num_out++] = &sess_key_sg;
377                 }
378                 else{
379                         num_out++;
380                 }
381
382                 if(sess_out.mackeylen){
383                         sg_init_one(&sess_mackey_sg,
                                mackey, sess_out.mackeylen);
384                         sgs[num_out++] = &sess_mackey_sg;
385                 }
```

```
386                 else{
387                         num_out++;
388                 }
389
390                 sg_init_one(&data_in_sg, &sess_in, sizeof
                        (sess_in));
391                 sgs[num_out + num_in++] = &data_in_sg;
392                 break;
393
394         case CIOCCRYPT:
395                 debug("CIOCCRYPT");
396
397                 /**
398                  * Copy the crypt struct
399                  **/
400                 err = copy_from_user(&cryp_out, (struct
                        crypt_op *)arg, sizeof(cryp_out));
401
402                 sg_init_one(&data_out_sg, &cryp_out,
                        sizeof(cryp_out));
403                 sgs[num_out++] = &data_out_sg;
404
405                 debug("Succesfully copied data_out");
406
407
408                 /**
409                  * Initialize support sgs
410                  **/
411                 src = kzalloc(cryp_out.len, GFP_KERNEL);
412                 if (!src) {
413                         ret = -ENOMEM;
414                         goto out;
415                 }
416                 err = copy_from_user(src, cryp_out.src,
                        cryp_out.len);
417
418                 dst = kzalloc(cryp_out.len, GFP_KERNEL);
419                 if (!dst) {
420                         ret = -ENOMEM;
421                         goto out;
422                 }
423                 err = copy_from_user(dst, cryp_out.dst,
                        cryp_out.len);
424
425                 mac = kzalloc(cryp_out.len, GFP_KERNEL);
426                 if (!mac) {
```

```
427                 ret = -ENOMEM;
428                 goto out;
429         }
430         err = copy_from_user(mac, cryp_out.mac,
                cryp_out.len);
431
432         iv = kzalloc(IV_LENGTH, GFP_KERNEL);
433         if (!iv) {
434                 ret = -ENOMEM;
435                 goto out;
436         }
437         err = copy_from_user(iv, cryp_out.iv,
                IV_LENGTH);
438
439         debug("Succesfully copied crypt_data");
440
441         /* Out */
442         sg_init_one(&crypt_iv_sg, iv, IV_LENGTH);
443         sgs[num_out++] = &crypt_iv_sg;
444
445         if(cryp_out.len){
446                 sg_init_one(&crypt_src_sg, src,
                        cryp_out.len);
447                 sgs[num_out++] = &crypt_src_sg;
448
449                 /* In */
450                 sg_init_one(&crypt_dst_sg, dst,
                        cryp_out.len);
451                 sgs[num_out + num_in++] = &
                        crypt_dst_sg;
452
453                 sg_init_one(&crypt_mac_sg, mac,
                        cryp_out.len);
454                 sgs[num_out + num_in++] = &
                        crypt_mac_sg;
455         }
456         else{
457                 num_out++;
458                 num_in++;
459                 num_in++;
460         }
461
462         sg_init_one(&data_in_sg, &cryp_in, sizeof
                (cryp_in));
463         sgs[num_out + num_in++] = &data_in_sg;
464         break;
```

```
465
466             default:
467                     debug("Unsupported ioctl command");
468
469                     break;
470             }
471
472             sg_init_one(&error_sg, &host_error, sizeof(
                    host_error));
473             sgs[num_out + num_in++] = &error_sg;
474
475             debug("After error sg");
476
477
478             /**
479              * Wait for the host to process our data.
480              **/
481
482
483             /* Locked Area  */
484
485             spin_lock(&crdev->lock);
486             err = virtqueue_add_sgs(crdev->vq, sgs, num_out,
                    num_in,
487                                     &syscall_type_sg,
                                            GFP_ATOMIC);
488             debug("%d",err);
489             if (err) {return err;}
490             virtqueue_kick(crdev->vq);
491             while (virtqueue_get_buf(crdev->vq, &len) == NULL
                    )
492                     /* do nothing */;
493
494             spin_unlock(&crdev->lock);
495
496
497             /**
498              * TODO
499              * Handle the output of the virtqueue
500              * We have to reconnect the struct correctly
501              **/
502             if(host_error){
503                     ret = host_error;
504                     goto out;
505             }
506             switch (cmd) {
```

```
507             case CIOCGSESSION:
508                     err = copy_to_user(sess_in.key, key,
                            sess_in.keylen);
509                     err = copy_to_user(sess_in.mackey, mackey
                            , sess_in.mackeylen);
510                     err = copy_to_user((struct session_op *)
                            arg, &sess_in, sizeof(sess_in));
511             case CIOCFSESSION:
512                     err = copy_to_user(sess_in.key, key,
                            sess_in.keylen);
513                     err = copy_to_user(sess_in.mackey, mackey
                            , sess_in.mackeylen);
514                     err = copy_to_user((struct session_op *)
                            arg, &sess_in, sizeof(sess_in));
515             case CIOCCRYPT:
516                     err = copy_to_user(cryp_in.iv, iv,
                            IV_LENGTH);
517                     err = copy_to_user(cryp_in.src, src,
                            cryp_in.len);
518                     err = copy_to_user(cryp_in.dst, dst,
                            cryp_in.len);
519                     err = copy_to_user(cryp_in.mac, mac,
                            cryp_in.len);
520                     err = copy_to_user((struct crypt_op *)arg
                            , &cryp_in, sizeof(cryp_in));
521             default:
522                     debug("Unsupported ioctl command");
523
524                     break;
525             }
526
527     out:
528             debug("Leaving");
529
530             return ret;
531     }
532
533     static ssize_t crypto_chrdev_read(struct file *filp, char
            __user *usrbuf,
534                                       size_t cnt, loff_t *
                                          f_pos)
535     {
536             debug("Entering");
537             debug("Leaving");
538             return -EINVAL;
539     }
```

```
540
541 static struct file_operations crypto_chrdev_fops =
542 {
543         .owner          = THIS_MODULE,
544         .open           = crypto_chrdev_open,
545         .release        = crypto_chrdev_release,
546         .read           = crypto_chrdev_read,
547         .unlocked_ioctl = crypto_chrdev_ioctl,
548 };
549
550 int crypto_chrdev_init(void)
551 {
552         int ret;
553         dev_t dev_no;
554         unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES
            ;
555
556         debug("Initializing character device...");
557         cdev_init(&crypto_chrdev_cdev, &
            crypto_chrdev_fops);
558         crypto_chrdev_cdev.owner = THIS_MODULE;
559
560         dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
561         ret = register_chrdev_region(dev_no,
            crypto_minor_cnt, "crypto_devs");
562         if (ret < 0) {
563                 debug("failed to register region, ret = %
                    d", ret);
564                 goto out;
565         }
566         ret = cdev_add(&crypto_chrdev_cdev, dev_no,
            crypto_minor_cnt);
567         if (ret < 0) {
568                 debug("failed to add character device");
569                 goto out_with_chrdev_region;
570         }
571
572         debug("Completed successfully");
573         return 0;
574
575 out_with_chrdev_region:
576         unregister_chrdev_region(dev_no, crypto_minor_cnt
            );
577 out:
578         return ret;
579 }
```

```
580
581 void crypto_chrdev_destroy(void)
582 {
583         dev_t dev_no;
584         unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES
        ;
585
586         debug("entering");
587         dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
588         cdev_del(&crypto_chrdev_cdev);
589         unregister_chrdev_region(dev_no, crypto_minor_cnt
        );
590         debug("leaving");
591 }
```

crypto.h

```
1  #ifndef _CRYPTO_H
2  #define _CRYPTO_H
3
4  #define VIRTIO_CRYPTO_BLOCK_SIZE     16
5
6  #define VIRTIO_CRYPTO_SYSCALL_OPEN   0
7  #define VIRTIO_CRYPTO_SYSCALL_CLOSE  1
8  #define VIRTIO_CRYPTO_SYSCALL_IOCTL  2
9
10 /* The Virtio ID for virtio crypto ports */
11 #define VIRTIO_ID_CRYPTO             13
12
13 /**
14  * Global driver data.
15  **/
16 struct crypto_driver_data {
17         /* The list of the devices we are handling. */
18         struct list_head devs;
19
20         /* The minor number that we give to the next
            device. */
21         unsigned int next_minor;
22
23         spinlock_t lock;
24 };
25 extern struct crypto_driver_data crdrvdata;
26
```

```
27
28 /**
29  * Device info.
30  **/
31 struct crypto_device {
32         /* Next crypto device in the list, head is in the
                crdrvdata struct */
33         struct list_head list;
34
35         /* The virtio device we are associated with. */
36         struct virtio_device *vdev;
37
38         struct virtqueue *vq;
39         /* ?? Lock ?? */
40
41         spinlock_t lock;
42
43         /* The minor number of the device. */
44         unsigned int minor;
45 };
46
47
48 /**
49  *  Crypto open file.
50  **/
51 struct crypto_open_file {
52         /* The crypto device this open file is associated
                with. */
53         struct crypto_device *crdev;
54
55         /* The fd that this device has on the Host. */
56         int host_fd;
57 };
58
59 #endif
```