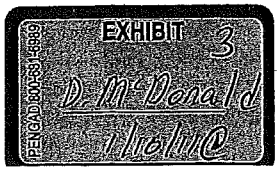
Exhibit 2

```
/*-----------------------------------------------------------------------
  key.c :         Key Management Engine for BSD

  Copyright 1995 by Bao Phan,  Randall Atkinson, & Dan McDonald,
  All Rights Reserved.  All Rights have been assigned to the US
  Naval Research Laboratory (NRL).  The NRL Copyright Notice and
  License governs distribution and use of this software.

  Patents are pending on this technology.  NRL grants a license
  to use this technology at no cost under the terms below with
  the additional requirement that software, hardware, and
  documentation relating to use of this technology must include
  the note that:
        This product includes technology developed at and
    licensed from the Information Technology Division,
    US Naval Research Laboratory.

-----------------------------------------------------------------------*/
/*-----------------------------------------------------------------------
#   @(#)COPYRIGHT   1.1a (NRL) 17 August 1995

COPYRIGHT NOTICE

All of the documentation and software included in this software
distribution from the US Naval Research Laboratory (NRL) are
copyrighted by their respective developers.

This software and documentation were developed at NRL by various
people.  Those developers have each copyrighted the portions that they
developed at NRL and have assigned All Rights for those portions to
NRL.  Outside the USA, NRL also has copyright on the software
developed at NRL. The affected files all contain specific copyright
notices and those notices must be retained in any derived work.

NRL LICENSE

NRL grants permission for redistribution and use in source and binary
forms, with or without modification, of the software and documentation
created at NRL provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software
   must display the following acknowledgement:

    This product includes software developed at the Information
    Technology Division, US Naval Research Laboratory.

4. Neither the name of the NRL nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THE SOFTWARE PROVIDED BY NRL IS PROVIDED BY NRL AND CONTRIBUTORS ``AS
IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NRL OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
 68  The views and conclusions contained in the software and documentation
 69  are those of the authors and should not be interpreted as representing
 70  official policies, either expressed or implied, of the US Naval
 71  Research Laboratory (NRL).
 72
 73  ----------------------------------------------------------------------*/
 74
 75  #include <sys/types.h>
 76  #include <sys/param.h>
 77  #include <sys/proc.h>
 78  #include <sys/mbuf.h>
 79  #include <sys/socket.h>
 80  #include <sys/socketvar.h>
 81  #include <sys/time.h>
 82  #include <sys/kernel.h>
 83  #include <net/raw cb.h>
 84  #include <net/if.h>
 85  #include <net/if types.h>
 86  #include <net/if dl.h>
 87  #include <net/route.h>
 88  #include <netinet/in.h>
 89  #include <netinet/in var.h>
 90  #include <netinet/if_ether.h>
 91
 92  #include <netinet6/in6.h>
 93  #include <netinet6/in6 var.h>
 94  #include <netinet6/ipsec.h>
 95  #include <netinet6/key.h>
 96  #include <netinet6/in6_debug.h>
 97
 98  #define MAXHASHKEYLEN (2 * sizeof(int) + 2 * sizeof(struct
     sockaddr_in6))
 99
100  /*
101   *  Not clear whether these values should be
102   *  tweakable at kernel config time.
103   */
104  #define KEYTBLSIZE 61
105  #define KEYALLOCTBLSIZE 61
106  #define SO2SPITBLSIZE 61
107
108  /*
109   *  These values should be tweakable...
110   *  perhaps by using sysctl
111   */
112
113  #define MAXLARVALTIME 240;    /* Lifetime of a larval key table entry */
114  #define MAXKEYACQUIRE 1;      /* Max number of key acquire messages sent
     */
115                                /*   per destination address
                                   */
116  #define MAXACQUIRETIME 15;    /* Lifetime of acquire message */
117
118  /*
119   *  Key engine tables and global variables
120   */
121
122  struct key tblnode keytable[KEYTBLSIZE];
123  struct key allocnode keyalloctbl[KEYALLOCTBLSIZE];
124  struct key_so2spinode so2spitbl[SO2SPITBLSIZE];
125
126  struct keyso cb keyso cb;
127  struct key tblnode nullkeynode  = { 0, 0, 0, 0, 0 };
128  struct key registry *keyregtable;
129  struct key acquirelist *key acquirelist;
130  u long maxlarvallifetime = MAXLARVALTIME;
131  int maxkeyacquire = MAXKEYACQUIRE;
```

```
132  u_long maxacquiretime = MAXACQUIRETIME;
133
134  extern void dump_secassoc();
135
136
137  /*-------------------------------------------------------------------
138   * (temporary) Dump a data buffer
139   -------------------------------------------------------------------*
     /
140
141  void
142  dump_buf(buf, len)
143       char *buf;
144       int len;
145  {
146    int i;
147
148    printf("buf=0x%x len=%d:\n", buf, len);
149    for (i = 0; i < len; i++) {
150      printf("0x%x ", (u_int8)*(buf+i));
151    }
152    printf("\n");
153  }
154
155
156  /*-------------------------------------------------------------------
157   * (temporary) Dump a key tblnode structrue
158   -------------------------------------------------------------------*
     /
159
160  void
161  dump_keytblnode(ktblnode)
162       struct key_tblnode *ktblnode;
163  {
164    if (!ktblnode) {
165      printf("NULL key table node pointer!\n");
166      return;
167    }
168    printf("solist=0x%x ", ktblnode->solist);
169    printf("secassoc=0x%x ", ktblnode->secassoc);
170    printf("next=0x%x\n", ktblnode->next);
171  }
172
173
174  /*-------------------------------------------------------------------
175   * key secassoc2msghdr():
176   *      Copy info from a security association into a key message buffer.
177   *      Assume message buffer is sufficiently large to hold all security
178   *      association information including src, dst, from, key and iv.
179   -------------------------------------------------------------------*
     /
180  int
181  key_secassoc2msghdr(secassoc, km, keyinfo)
182       struct ipsec assoc *secassoc;
183       struct key msghdr *km;
184       struct key_msgdata *keyinfo;
185  {
186    char *cp;
187    DPRINTF(IDL_GROSS_EVENT, ("Entering key_secassoc2msghdr\n"));
188
189    if ((km == 0) || (keyinfo == 0) || (secassoc == 0))
190      return(-1);
191
192    km->type = secassoc->type;
193    km->state = secassoc->state;
194    km->label = secassoc->label;
195    km->spi = secassoc->spi;
```

```
196     km->keylen = secassoc->keylen;
197     km->ivlen = secassoc->ivlen;
198     km->algorithm = secassoc->algorithm;
199     km->lifetype = secassoc->lifetype;
200     km->lifetime1 = secassoc->lifetime1;
201     km->lifetime2 = secassoc->lifetime2;
202
203     /*
204      *  Stuff src/dst/from/key/iv in buffer after
205      *  the message header.
206      */
207     cp = (char *)(km + 1);
208
209   #define ROUNDUP(a) \
210     ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
211   #define ADVANCE(x, n) \
212       { x += ROUNDUP(n); }
213
214     DPRINTF(IDL_FINISHED, ("sa2msghdr: 1\n"));
215     keyinfo->src = (struct sockaddr *)cp;
216     if (secassoc->src.sin6_len) {
217       bcopy((char *)&(secassoc->src), cp, secassoc->src.sin6_len);
218       ADVANCE(cp, secassoc->src.sin6_len);
219     } else {
220       bzero(cp, sizeof(struct sockaddr_in6));
221       ADVANCE(cp, sizeof(struct sockaddr_in6));
222     }
223     DPRINTF(IDL_FINISHED, ("sa2msghdr: 2\n"));
224
225     keyinfo->dst = (struct sockaddr *)&(secassoc->dst);
226     if (secassoc->dst.sin6_len) {
227       bcopy((char *)&(secassoc->dst), cp, secassoc->dst.sin6_len);
228       ADVANCE(cp, secassoc->dst.sin6_len);
229     } else {
230       bzero(cp, sizeof(struct sockaddr_in6));
231       ADVANCE(cp, sizeof(struct sockaddr_in6));
232     }
233     DPRINTF(IDL_FINISHED, ("sa2msghdr: 3\n"));
234
235     keyinfo->from = (struct sockaddr *)cp;
236     if (secassoc->from.sin6_len) {
237       bcopy((char *)&(secassoc->from), cp, secassoc->from.sin6_len);
238       ADVANCE(cp, secassoc->from.sin6_len);
239     } else {
240       bzero(cp, sizeof(struct sockaddr_in6));
241       ADVANCE(cp, sizeof(struct sockaddr_in6));
242     }
243     DPRINTF(IDL_FINISHED, ("sa2msghdr: 4\n"));
244
245     keyinfo->key = cp;
246     keyinfo->keylen = secassoc->keylen;
247     if (secassoc->keylen) {
248       bcopy((char *)(secassoc->key), cp, secassoc->keylen);
249       ADVANCE(cp, secassoc->keylen);
250     }
251
252     DPRINTF(IDL_FINISHED, ("sa2msghdr: 5\n"));
253     keyinfo->iv = cp;
254     keyinfo->ivlen = secassoc->ivlen;
255     if (secassoc->ivlen) {
256       bcopy((char *)(secassoc->iv), cp, secassoc->ivlen);
257       ADVANCE(cp, secassoc->ivlen);
258     }
259
260     DDO(IDL_FINISHED,printf("msgbuf(len=%d):\n",(char *)cp - (char *)km));
261     DDO(IDL_FINISHED,dump_buf((char *)km, (char *)cp - (char *)km));
262     DPRINTF(IDL_FINISHED, ("sa2msghdr: 6\n"));
```

```
263     return(0);
264   }
265
266
267   /*-----------------------------------------------------------------------
268    * key msghdr2secassoc():
269    *      Copy info from a key message buffer into an ipsec_assoc
270    *      structure
271    -----------------------------------------------------------------------*
       /
272   int
273   key_msghdr2secassoc(secassoc, km, keyinfo)
274        struct ipsec assoc *secassoc;
275        struct key msghdr *km;
276        struct key_msgdata *keyinfo;
277   {
278     DPRINTF(IDL_GROSS_EVENT, ("Entering key_msghdr2secassoc\n"));
279
280     if ((km == 0) || (keyinfo == 0) || (secassoc == 0))
281       return(-1);
282
283     secassoc->len = sizeof(*secassoc);
284     secassoc->type = km->type;
285     secassoc->state = km->state;
286     secassoc->label = km->label;
287     secassoc->spi = km->spi;
288     secassoc->keylen = km->keylen;
289     secassoc->ivlen = km->ivlen;
290     secassoc->algorithm = km->algorithm;
291     secassoc->lifetype = km->lifetype;
292     secassoc->lifetime1 = km->lifetime1;
293     secassoc->lifetime2 = km->lifetime2;
294
295     if (keyinfo->src)
296       bcopy((char *)(keyinfo->src), (char *)&(secassoc->src),
297         keyinfo->src->sa_len);
298
299     if (keyinfo->dst)
300       bcopy((char *)(keyinfo->dst), (char *)&(secassoc->dst),
301         keyinfo->dst->sa_len);
302
303     if (keyinfo->from)
304       bcopy((char *)(keyinfo->from), (char *)&(secassoc->from),
305         keyinfo->from->sa_len);
306
307     /*
308      *  Make copies of key and iv
309      */
310     if (secassoc->ivlen) {
311       K_Malloc(secassoc->iv, caddr_t, secassoc->ivlen);
312       if (secassoc->iv == 0) {
313         DPRINTF(IDL_CRITICAL,("msghdr2secassoc: can't allocate mem for
          iv\n"));
314         return(-1);
315       }
316       bcopy((char *)keyinfo->iv, (char *)secassoc->iv, secassoc->ivlen);
317     } else
318       secassoc->iv = NULL;
319
320     if (secassoc->keylen) {
321       K_Malloc(secassoc->key, caddr_t, secassoc->keylen);
322       if (secassoc->key == 0) {
323         DPRINTF(IDL_CRITICAL,("msghdr2secassoc: can't allocate mem for
          key\n"));
324         if (secassoc->iv)
325       KFree(secassoc->iv);
326         return(-1);
```

```
327       }
328       bcopy((char *)keyinfo->key, (char *)secassoc->key,
          secassoc->keylen);
329     } else
330       secassoc->key = NULL;
331     return(0);
332   }
333
334
335   /*---------------------------------------------------------------------
336    * addrpart equal():
337    *      Determine if the address portion of two sockaddrs are equal.
338    *      Currently handles only AF INET and AF INET6 address families.
339    ---------------------------------------------------------------------*
       /
340   int
341   addrpart equal(sa1, sa2)
342        struct sockaddr *sa1;
343        struct sockaddr *sa2;
344   {
345
346     if ((sa1->sa family == sa2->sa_family))
347       switch(sa1->sa_family) {
348       case AF INET:
349         if (((struct sockaddr in *)sa1)->sin addr.s addr ==
350         ((struct sockaddr_in *)sa2)->sin_addr.s_addr)
351       return(1);
352         break;
353       case AF INET6:
354         if (IN6 ADDR EQUAL(((struct sockaddr in6 *)sa1)->sin6_addr,
355                 ((struct sockaddr_in6 *)sa2)->sin6_addr))
356       return(1);
357         break;
358       }
359     return(0);
360   }
361
362
363   /*---------------------------------------------------------------------
364    * my addr():
365    *      Determine if an address belongs to one of my configured
       interfaces.
366    *      Currently handles only AF INET and AF INET6 addresses.
367    ---------------------------------------------------------------------*
       /
368   int
369   my_addr(sa)
370        struct sockaddr *sa;
371   {
372     extern struct in6 ifaddr *in6 ifaddr;
373     extern struct in ifaddr *in_ifaddr;
374     struct in6 ifaddr *i6a = 0;
375     struct in_ifaddr *ia = 0;
376
377     switch(sa->sa family) {
378     case AF INET6:
379       for (i6a = in6 ifaddr; i6a; i6a = i6a->i6a_next) {
380         if (IN6 ADDR EQUAL(((struct sockaddr_in6 *)sa)->sin6_addr,
381                 i6a->i6a_addr.sin6_addr))
382       return(1);
383       }
384       break;
385     case AF INET:
386       for (ia = in ifaddr; ia; ia = ia->ia next) {
387         if (((struct sockaddr in *)sa)->sin_addr.s_addr ==
388          ia->ia addr.sin_addr.s_addr)
389       return(1);
```

```
390       }
391       break;
392     }
393     return(0);
394   }
395
396
397   /*---------------------------------------------------------------------
398    * key inittables():
399    *       Allocate space and initialize key engine tables
400    ---------------------------------------------------------------------*
       /
401   void
402   key_inittables()
403   {
404     struct key_tblnode *keynode;
405     int i;
406
407     K_Malloc(keyregtable, struct key_registry *, sizeof(struct
       key registry));
408     if (keyregtable == 0)
409       panic("key inittables");
410     bzero((char *)keyregtable, sizeof(struct key registry));
411     K_Malloc(key acquirelist, struct key acquirelist *,
412          sizeof(struct key_acquirelist));
413     if (key acquirelist == 0)
414       panic("key inittables");
415     bzero((char *)key acquirelist, sizeof(struct key_acquirelist));
416     for (i = 0; i < KEYTBLSIZE; i++)
417       bzero((char *)&keytable[i], sizeof(struct key_tblnode));
418     for (i = 0; i < KEYALLOCTBLSIZE; i++)
419       bzero((char *)&keyalloctbl[i], sizeof(struct key_allocnode));
420     for (i = 0; i < SO2SPITBLSIZE; i++)
421       bzero((char *)&so2spitbl[i], sizeof(struct key_so2spinode));
422   }
423
424
425   /*---------------------------------------------------------------------
426    * key gethashval():
427    *       Determine keytable hash value.
428    ---------------------------------------------------------------------*
       /
429   int
430   key_gethashval(buf, len, tblsize)
431        char *buf;
432        int len;
433        int tblsize;
434   {
435     int i, j = 0;
436
437     /*
438      * Todo: Use word size xor and check for alignment
439      *       and zero pad if necessary.  Need to also pick
440      *       a good hash function and table size.
441      */
442     if (len <= 0) {
443       DPRINTF(IDL_CRITICAL,("key_gethashval got bogus len!\n"));
444       return(-1);
445     }
446     for(i = 0; i < len; i++) {
447       j ^=  (u_int8)(*(buf + i));
448     }
449     return (j % tblsize);
450   }
451
452
453   /*---------------------------------------------------------------------
```

```
454   * key_createkey():
455   *      Create hash key for hash function
456   *      key is: type+src+dst if keytype = 1
457   *              type+src+dst+spi if keytype = 0
458   *      Uses only the address portion of the src and dst sockaddrs to
459   *      form key.  Currently handles only AF INET and AF INET6 sockaddrs
460   ------------------------------------------------------------------------*
      /
461   int
462   key_createkey(buf, type, src, dst, spi, keytype)
463        char *buf;
464        u_int type;
465        struct sockaddr *src;
466        struct sockaddr *dst;
467        u_int32 spi;
468        u_int keytype;
469   {
470     char *cp, *p;
471
472     DPRINTF(IDL_FINISHED,("Entering key_createkey\n"));
473
474     if (!buf || !src || !dst)
475       return(-1);
476
477     cp = buf;
478     bcopy((char *)&type, cp, sizeof(type));
479     cp += sizeof(type);
480
481     /*
482      * Assume only IPv4 and IPv6 addresses.
483      */
484   #define ADDRPART(a) \
485        ((a)->sa_family == AF_INET6) ? \
486        (char *)&(((struct sockaddr_in6 *)(a))->sin6_addr) : \
487        (char *)&(((struct sockaddr_in *)(a))->sin_addr)
488
489   #define ADDRSIZE(a) \
490        ((a)->sa_family == AF_INET6) ? sizeof(struct in_addr6) : \
491        sizeof(struct in_addr)
492
493     DPRINTF(IDL_GROSS_EVENT,("src addr:\n"));
494     DDO(IDL_GROSS_EVENT,dump_smart_sockaddr(src));
495     DPRINTF(IDL_GROSS_EVENT,("dst addr:\n"));
496     DDO(IDL_GROSS_EVENT,dump_smart_sockaddr(dst));
497
498     p = ADDRPART(src);
499     bcopy(p, cp, ADDRSIZE(src));
500     cp += ADDRSIZE(src);
501
502     p = ADDRPART(dst);
503     bcopy(p, cp, ADDRSIZE(dst));
504     cp += ADDRSIZE(dst);
505
506   #undef ADDRPART
507   #undef ADDRSIZE
508
509     if (keytype == 0) {
510       bcopy((char *)&spi, cp, sizeof(spi));
511       cp += sizeof(spi);
512     }
513
514     DPRINTF(IDL_FINISHED,("hash key:\n"));
515     DDO(IDL_FINISHED, dump_buf(buf, cp - buf));
516     return(cp - buf);
517   }
518
519
```

```
520  /*-----------------------------------------------------------------------
521   * key sosearch():
522   *      Search the so2spi table for the security association allocated
      to
523   *      the socket.  Returns pointer to a struct key_so2spinode which
      can
524   *      be used to locate the security association entry in the
      keytable.
525   -----------------------------------------------------------------------*
      /
526  struct key so2spinode *
527  key_sosearch(type, src, dst, so)
528       u int type;
529       struct sockaddr *src;
530       struct sockaddr *dst;
531       struct socket *so;
532  {
533    struct key_so2spinode *np = 0;
534
535    if (!(src && dst)) {
536      DPRINTF(IDL CRITICAL,("key_sosearch: got null src or dst
         pointer!\n"));
537      return(NULL);
538    }
539
540    for (np = so2spitbl[((u_int32)so) % SO2SPITBLSIZE].next; np; np = np->
      next) {
541      if ((so == np->socket) && (type == np->keynode->secassoc->type)
542      && addrpart equal(src,
543                (struct sockaddr *)&(np->keynode->secassoc->src))
544      && addrpart equal(dst,
545                (struct sockaddr *)&(np->keynode->secassoc->dst)))
546        return(np);
547    }
548    return(NULL);
549  }
550
551
552  /*-----------------------------------------------------------------------
553   * key sodelete():
554   *       Delete entries from the so2spi table.
555   *         flag = 1  purge all entries
556   *         flag = 0  delete entries with socket pointer matching socket
557   -----------------------------------------------------------------------*
      /
558  void
559  key_sodelete(socket, flag)
560       struct socket *socket;
561       int flag;
562  {
563    struct key so2spinode *prevnp, *np;
564    int s = splnet();
565
566    DPRINTF(IDL MAJOR_EVENT,("Entering keysodelete w/so=0x%x flag=%d\n",
      socket,flag));
567    if (flag) {
568      int i;
569
570      for (i = 0; i < SO2SPITBLSIZE; i++)
571        for(np = so2spitbl[i].next; np; np = np->next) {
572      KFree(np);
573        }
574      splx(s);
575      return;
576    }
577
578    prevnp = &so2spitbl[((u_int32)socket) % SO2SPITBLSIZE];
```

```
579     for(np = prevnp->next; np; np = np->next) {
580       if (np->socket == socket) {
581         struct socketlist *socklp, *prevsocklp;
582
583         (np->keynode->alloc_count)--;
584
585         /*
586          * If this socket maps to a unique secassoc,
587          * we go ahead and delete the secassoc, since it
588          * can no longer be allocated or used by any other
589          * socket.
590          */
591         if (np->keynode->secassoc->state & K_UNIQUE) {
592       if (key_delete(np->keynode->secassoc) != 0)
593         panic("key_sodelete");
594       np = prevnp;
595       continue;
596         }
597
598         /*
599          * We traverse the socketlist and remove the entry
600          * for this socket
601          */
602         DPRINTF(IDL_FINISHED,("keysodelete: deleting from socklist..."));
603         prevsocklp = np->keynode->solist;
604         for (socklp = prevsocklp->next; socklp; socklp = socklp->next) {
605       if (socklp->socket == socket) {
606         prevsocklp->next = socklp->next;
607         KFree(socklp);
608         break;
609       }
610       prevsocklp = socklp;
611         }
612         DPRINTF(IDL_FINISHED,("done\n"));
613         prevnp->next = np->next;
614         KFree(np);
615         np = prevnp;
616       }
617       prevnp = np;
618     }
619     splx(s);
620   }
621
622
623   /*---------------------------------------------------------------------
624    * key_deleteacquire():
625    *      Delete an entry from the key acquirelist
626    ---------------------------------------------------------------------*
       /
627   void
628   key_deleteacquire(type, target)
629        u_int type;
630        struct sockaddr *target;
631   {
632     struct key_acquirelist *ap, *prev;
633
634     prev = key_acquirelist;
635     for(ap = key_acquirelist->next; ap; ap = ap->next) {
636       if (addrpart_equal(target, (struct sockaddr *)&(ap->target)) &&
637       (type == ap->type)) {
638         DPRINTF(IDL_MAJOR_EVENT,("Deleting entry from acquire list!\n"));
639         prev->next = ap->next;
640         KFree(ap);
641         ap = prev;
642       }
643       prev = ap;
644     }
```

```
645  }
646
647
648  /*-----------------------------------------------------------------------
649   * key search():
650   *      Search the key table for an entry with same type, src addr, dest
651   *      addr, and spi.  Returns a pointer to struct key_tblnode if found
652   *      else returns null.
653   -----------------------------------------------------------------------*
      /
654  struct key tblnode *
655  key_search(type, src, dst, spi, indx, prevkeynode)
656       u int type;
657       struct sockaddr *src;
658       struct sockaddr *dst;
659       u int32 spi;
660       int indx;
661       struct key_tblnode **prevkeynode;
662  {
663    struct key_tblnode *keynode, *prevnode;
664
665    if (indx > KEYTBLSIZE || indx < 0)
666      return (NULL);
667    if (!(&keytable[indx]))
668      return (NULL);
669
670  #define sec type keynode->secassoc->type
671  #define sec spi keynode->secassoc->spi
672  #define sec src keynode->secassoc->src
673  #define sec_dst keynode->secassoc->dst
674
675    prevnode = &keytable[indx];
676    for (keynode = keytable[indx].next; keynode; keynode = keynode->next)
       {
677      if ((type == sec type) && (spi == sec spi) &&
678      addrpart equal(src, (struct sockaddr *)&(sec src))
679      && addrpart_equal(dst, (struct sockaddr *)&(sec_dst)))
680        break;
681      prevnode = keynode;
682    }
683    *prevkeynode = prevnode;
684    return(keynode);
685  }
686
687
688  /*-----------------------------------------------------------------------
689   * key addnode():
690   *      Insert a key_tblnode entry into the key table.  Returns a
       pointer
691   *      to the newly created key tblnode.
692   -----------------------------------------------------------------------*
      /
693  struct key tblnode *
694  key_addnode(indx, secassoc)
695       int indx;
696       struct ipsec_assoc *secassoc;
697  {
698    struct key_tblnode *keynode;
699
700    DPRINTF(IDL GROSS EVENT,("Entering key addnode w/indx=%d
       secassoc=0x%x\n",indx, (u_int32)secassoc));
701
702    if (!(&keytable[indx]))
703      return(NULL);
704    if (!secassoc) {
705      panic("key_addnode: Someone passed in a null secassoc!\n");
706    }
```

```
707
708     K_Malloc(keynode, struct key_tblnode *, sizeof(struct key_tblnode));
709     if (keynode == 0)
710       return(NULL);
711     bzero((char *)keynode, sizeof(struct key_tblnode));
712
713     K_Malloc(keynode->solist, struct socketlist *, sizeof(struct
        socketlist));
714     if (keynode->solist == 0) {
715       KFree(keynode);
716       return(NULL);
717     }
718     bzero((char *)(keynode->solist), sizeof(struct socketlist));
719
720     keynode->secassoc = secassoc;
721     keynode->solist->next = NULL;
722     keynode->next = keytable[indx].next;
723     keytable[indx].next = keynode;
724     return(keynode);
725   }
726
727
728   /*---------------------------------------------------------------------
729    * key_add():
730    *      Add a new security association to the key table.  Caller is
731    *      responsible for allocating memory for the struct ipsec_assoc as

732    *      well as the buffer space for the key and iv.  Assumes the
       security
733    *      association passed in is well-formed.
734    ---------------------------------------------------------------------*
       /
735   int
736   key_add(secassoc)
737        struct ipsec_assoc *secassoc;
738   {
739     char buf[MAXHASHKEYLEN];
740     int len, indx;
741     int inbound = 0;
742     int outbound = 0;
743     struct key_tblnode *keynode, *prevkeynode;
744     struct key_allocnode *np;
745     int s;
746
747     DPRINTF(IDL_GROSS_EVENT, ("Entering key_add w/secassoc=0x%x\n",
        secassoc));
748
749     if (!secassoc) {
750       panic("key_add: who the hell is passing me a null pointer");
751     }
752
753     /*
754      *  For storage purposes, the two esp modes are
755      *  treated the same.
756      */
757     if (secassoc->type == SS_ENCRYPTION_NETWORK)
758       secassoc->type = SS_ENCRYPTION_TRANSPORT;
759
760     /*
761      * Should we allow a null key to be inserted into the table ?
762      * or can we use null key to indicate some policy action...
763      */
764
765     /*
766      *  For esp using des-cbc or tripple-des we call
767      * des_set_odd_parity.
768      */
```

```
769     if (secassoc->key && (secassoc->type == SS ENCRYPTION TRANSPORT) &&
770         ((secassoc->algorithm == IPSEC ALGTYPE ESP DES CBC) ||
771          (secassoc->algorithm == IPSEC ALGTYPE_ESP_3DES)))
772       des_set_odd_parity(secassoc->key);
773
774     /*
775      *  Check if secassoc with same spi exists before adding
776      */
777     bzero((char *)&buf, sizeof(buf));
778     len = key_createkey((char *)&buf, secassoc->type,
779                 (struct sockaddr *)&(secassoc->src),
780                 (struct sockaddr *)&(secassoc->dst),
781                 secassoc->spi, 0);
782     indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
783     DPRINTF(IDL GROSS EVENT,("keyadd: keytbl hash position=%d\n", indx));
784     keynode = key_search(secassoc->type, (struct sockaddr *)&(secassoc->
        src),
785                  (struct sockaddr *)&(secassoc->dst),
786                  secassoc->spi, indx, &prevkeynode);
787     if (keynode) {
788       DPRINTF(IDL_MAJOR_EVENT,("keyadd: secassoc already exists!\n"));
789       return(-2);
790     }
791
792     inbound = my addr((struct sockaddr *)&(secassoc->dst));
793     outbound = my addr((struct sockaddr *)&(secassoc->src));
794     DPRINTF(IDL_FINISHED,("inbound=%d outbound=%d\n", inbound, outbound));
795
796     /*
797      * We allocate mem for an allocation entry if needed.
798      * This is done here instead of in the allocaton code
799      * segment so that we can easily recover/cleanup from a
800      * memory allocation error.
801      */
802     if (outbound || (!inbound && !outbound)) {
803       K Malloc(np, struct key_allocnode *, sizeof(struct key_allocnode));
804       if (np == 0) {
805         DPRINTF(IDL_CRITICAL,("keyadd: can't allocate allocnode!\n"));
806         return(-1);
807       }
808     }
809
810     s = splnet();
811
812     if ((keynode = key addnode(indx, secassoc)) == NULL) {
813       DPRINTF(IDL_CRITICAL,("keyadd: key_addnode failed!\n"));
814       if (np)
815         KFree(np);
816       splx(s);
817       return(-1);
818     }
819     DPRINTF(IDL EVENT,("Added new keynode:\n"));
820     DDO(IDL GROSS EVENT, dump keytblnode(keynode));
821     DDO(IDL_GROSS_EVENT, dump_secassoc(keynode->secassoc));
822
823     /*
824      *  We add an entry to the allocation table for
825      *  this secassoc if the interfaces are up and
826      *  the secassoc is outbound.  In the case
827      *  where the interfaces are not up, we go ahead
828      *  and do it anyways.  This wastes an allocation
829      *  entry if the secassoc later turned out to be
830      *  inbound when the interfaces are ifconfig up.
831      */
832     if (outbound || (!inbound && !outbound)) {
833       len = key createkey((char *)&buf, secassoc->type,
834                 (struct sockaddr *)&(secassoc->src),
```

```
835                 (struct sockaddr *)&(secassoc->dst),
836                 0, 1);
837       indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
838       DPRINTF(IDL_GROSS_EVENT,("keyadd: keyalloc hash position=%d\n",
          indx));
839       np->keynode = keynode;
840       np->next = keyalloctbl[indx].next;
841       keyalloctbl[indx].next = np;
842     }
843     if (inbound)
844       secassoc->state |= K_INBOUND;
845     if (outbound)
846       secassoc->state |= K_OUTBOUND;
847
848     key_deleteacquire(secassoc->type, (struct sockaddr
      *)&(secassoc->dst));
849
850     splx(s);
851     return 0;
852   }
853
854
855   /*--------------------------------------------------------------------
856    * key_get():
857    *      Get a security association from the key table.
858    --------------------------------------------------------------------*
       /
859   int
860   key_get(type, src, dst, spi, secassoc)
861        u_int type;
862        struct sockaddr *src;
863        struct sockaddr *dst;
864        u_int32 spi;
865        struct ipsec_assoc **secassoc;
866   {
867     char buf[MAXHASHKEYLEN];
868     struct key_tblnode *keynode, *prevkeynode;
869     int len, indx;
870
871     /*
872      *  For storage purposes, the two esp modes are
873      *  treated the same.
874      */
875     if (type == SS_ENCRYPTION_NETWORK)
876       type = SS_ENCRYPTION_TRANSPORT;
877
878     bzero(&buf, sizeof(buf));
879     *secassoc = NULL;
880     len = key_createkey((char *)&buf, type, src, dst, spi, 0);
881     indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
882     DPRINTF(IDL_GROSS_EVENT,("keyget: indx=%d\n",indx));
883     keynode = key_search(type, src, dst, spi, indx, &prevkeynode);
884     if (keynode) {
885       DPRINTF(IDL_EVENT,("keyget: found it! keynode=0x%x",keynode));
886       *secassoc = keynode->secassoc;
887       return(0);
888     } else
889       return(-1);  /* Not found */
890   }
891
892
893   /*--------------------------------------------------------------------
894    * key_dump():
895    *      Dump all valid entries in the keytable to a pf_key socket.  Each
896    *      security associaiton is sent one at a time in a pf_key message.
       A
897    *      message with seqno = 0 signifies the end of the dump
```

```
 897   transaction.
 898   ------------------------------------------------------------------------*
       /
 899   int
 900   key_dump(so)
 901        struct socket *so;
 902   {
 903     int len, i;
 904     int seq = 1;
 905     struct mbuf *m;
 906     struct key msgdata keyinfo;
 907     struct key msghdr *km;
 908     struct key tblnode *keynode;
 909     extern struct sockaddr key src;
 910     extern struct sockaddr key_dst;
 911
 912     /*
 913      * Routine to dump the key table to a routing socket
 914      * Use for debugging only!
 915      */
 916
 917     DPRINTF(IDL_GROSS_EVENT,("Entering key_dump()"));
 918     /*
 919      * We need to speed this up later.  Fortunately, key_dump
 920      * messages are not sent often.
 921      */
 922     for (i = 0; i < KEYTBLSIZE; i++) {
 923       for (keynode = keytable[i].next; keynode; keynode = keynode->next) {
 924
 925         /*
 926          * We exclude dead/larval/zombie security associations for now
 927          * but it may be useful to also send these up for debugging
               purposes
 928          */
 929         if (keynode->secassoc->state & (K_DEAD | K_LARVAL | K_ZOMBIE))
 930       continue;
 931
 932         len = (sizeof(struct key msghdr) +
 933            ROUNDUP(keynode->secassoc->src.sin6 len) +
 934            ROUNDUP(keynode->secassoc->dst.sin6 len) +
 935            ROUNDUP(keynode->secassoc->from.sin6_len) +
 936            ROUNDUP(keynode->secassoc->keylen) +
 937            ROUNDUP(keynode->secassoc->ivlen));
 938         K Malloc(km, struct key_msghdr *, len);
 939         if (km == 0)
 940       return(ENOBUFS);
 941         if (key secassoc2msghdr(keynode->secassoc, km, &keyinfo) != 0)
 942       panic("key dump");
 943         km->key msglen = len;
 944         km->key msgvers = KEY VERSION;
 945         km->key msgtype = KEY DUMP;
 946         km->key pid = curproc->p_pid;
 947         km->key seq = seq++;
 948         km->key errno = 0;
 949         MGETHDR(m, M WAIT, MT DATA);
 950         m->m len = m->m_pkthdr.len = 0;
 951         m->m next = 0;
 952         m->m nextpkt = 0;
 953         m->m pkthdr.rcvif = 0;
 954         m copyback(m, 0, len, (caddr_t)km);
 955         KFree(km);
 956         if (sbappendaddr(&so->so_rcv, &key_src, m, (struct mbuf *)0) == 0)
 957       m_free(m);
 958         else
 959       sorwakeup(so);
 960       }
 961   }
```

```
 962    K_Malloc(km, struct key_msghdr *, sizeof(struct key_msghdr));
 963    if (km == 0)
 964      return(ENOBUFS);
 965    bzero((char *)km, sizeof(struct key msghdr));
 966    km->key msglen = sizeof(struct key_msghdr);
 967    km->key msgvers = KEY VERSION;
 968    km->key msgtype = KEY DUMP;
 969    km->key pid = curproc->p_pid;
 970    km->key seq = 0;
 971    km->key errno = 0;
 972    MGETHDR(m, M WAIT, MT DATA);
 973    m->m len = m->m_pkthdr.len = 0;
 974    m->m next = 0;
 975    m->m nextpkt = 0;
 976    m->m pkthdr.rcvif = 0;
 977    m copyback(m, 0, km->key_msglen, (caddr_t)km);
 978    KFree(km);
 979    if (sbappendaddr(&so->so_rcv, &key_src, m, (struct mbuf *)0) == 0)
 980      m free(m);
 981    else
 982      sorwakeup(so);
 983    DPRINTF(IDL_GROSS_EVENT,("Leaving key_dump()\n"));
 984    return(0);
 985  }
 986
 987  /*-------------------------------------------------------------------------
 988   * key delete():
 989   *      Delete a security association from the key table.
 990   -------------------------------------------------------------------------*
       /
 991  int
 992  key_delete(secassoc)
 993       struct ipsec_assoc *secassoc;
 994  {
 995    char buf[MAXHASHKEYLEN];
 996    int len, indx;
 997    struct key tblnode *keynode = 0;
 998    struct key tblnode *prevkeynode = 0;
 999    struct socketlist *socklp, *deadsocklp;
1000    struct key so2spinode *np, *prevnp;
1001    struct key_allocnode *ap, *prevap;
1002    int s;
1003
1004    DPRINTF(IDL_GROSS_EVENT,("Entering key_delete w/secassoc=0x%x\n",
      secassoc));
1005
1006    if (secassoc->type == SS ENCRYPTION NETWORK)
1007      secassoc->type = SS_ENCRYPTION_TRANSPORT;
1008
1009    bzero((char *)&buf, sizeof(buf));
1010    len = key_createkey((char *)&buf, secassoc->type,
1011                (struct sockaddr *)&(secassoc->src),
1012                (struct sockaddr *)&(secassoc->dst),
1013                secassoc->spi, 0);
1014    indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
1015    DPRINTF(IDL_GROSS_EVENT,("keydelete: keytbl hash position=%d\n",
      indx));
1016    keynode = key_search(secassoc->type, (struct sockaddr *)&(secassoc->
      src),
1017                 (struct sockaddr *)&(secassoc->dst),
1018                 secassoc->spi, indx, &prevkeynode);
1019
1020    if (keynode) {
1021      s = splnet();
1022      DPRINTF(IDL EVENT,("keydelete: found keynode to delete\n"));
1023      keynode->secassoc->state |= K_DEAD;
1024
```

```
1025      if (keynode->ref count > 0) {
1026        DPRINTF(IDL MAJOR EVENT,("keydelete: secassoc still held, marking
          for deletion only!\n"));
1027        splx(s);
1028        return(0);
1029      }
1030
1031      prevkeynode->next = keynode->next;
1032
1033      /*
1034       *  Walk the socketlist and delete the
1035       *  entries mapping sockets to this secassoc
1036       *  from the so2spi table.
1037       */
1038      DPRINTF(IDL GROSS EVENT,("keydelete: deleting socklist..."));
1039      for(socklp = keynode->solist->next; socklp; ) {
1040        prevnp = &so2spitbl[((u int32)(socklp->socket)) % SO2SPITBLSIZE];
1041        for(np = prevnp->next; np; np = np->next) {
1042      if ((np->socket == socklp->socket) && (np->keynode == keynode)) {
1043        prevnp->next = np->next;
1044        KFree(np);
1045        break;
1046      }
1047      prevnp = np;
1048        }
1049        deadsocklp = socklp;
1050        socklp = socklp->next;
1051        KFree(deadsocklp);
1052      }
1053      DPRINTF(IDL_GROSS_EVENT,("done\n"));
1054      /*
1055       * If an allocation entry exist for this
1056       * secassoc, delete it.
1057       */
1058      bzero((char *)&buf, sizeof(buf));
1059      len = key createkey((char *)&buf, secassoc->type,
1060              (struct sockaddr *)&(secassoc->src),
1061              (struct sockaddr *)&(secassoc->dst),
1062              0, 1);
1063      indx = key gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
1064      DPRINTF(IDL_GROSS_EVENT,("keydelete: alloctbl hash position=%d\n",
          indx));
1065      prevap = &keyalloctbl[indx];
1066      for (ap = prevap->next; ap; ap = ap->next) {
1067        if (ap->keynode == keynode) {
1068      prevap->next = ap->next;
1069      KFree(ap);
1070      break;
1071        }
1072        prevap = ap;
1073      }
1074
1075      if (keynode->secassoc->iv)
1076        KFree(keynode->secassoc->iv);
1077      if (keynode->secassoc->key)
1078        KFree(keynode->secassoc->key);
1079      KFree(keynode->secassoc);
1080      if (keynode->solist)
1081        KFree(keynode->solist);
1082      KFree(keynode);
1083      splx(s);
1084      return(0);
1085    }
1086    return(-1);
1087  }
1088
1089
```

```
1090  /*-------------------------------------------------------------------------
1091   * key flush():
1092   *      Delete all entries from the key table.
1093   *-----------------------------------------------------------------------*
       /
1094  void
1095  key_flush(void)
1096  {
1097    struct key_tblnode *keynode;
1098    int i;
1099
1100    /*
1101     * This is slow, but simple.
1102     */
1103    DPRINTF(IDL_FINISHED,("Flushing key table..."));
1104    for (i = 0; i < KEYTBLSIZE; i++) {
1105      while (keynode = keytable[i].next)
1106        if (key delete(keynode->secassoc) != 0)
1107      panic("key_flush");
1108    }
1109    DPRINTF(IDL_FINISHED,("done\n"));
1110  }
1111
1112
1113  /*-------------------------------------------------------------------------
1114   * key getspi():
1115   *      Get a unique spi value for a key management daemon/program.  The

1116   *      spi value, once assigned, cannot be assigned again.
1117   *-----------------------------------------------------------------------*
       /
1118  int
1119  key_getspi(type, src, dst, spi)
1120       u int type;
1121       struct sockaddr *src;
1122       struct sockaddr *dst;
1123       u_int32 *spi;
1124  {
1125    struct ipsec assoc *secassoc;
1126    struct key_tblnode *keynode, *prevkeynode;
1127    int count, done, len, indx;
1128    int maxcount = 1000;
1129    u int32 val;
1130    char buf[MAXHASHKEYLEN];
1131    int s;
1132
1133
1134    DPRINTF(IDL MAJOR EVENT,("Entering getspi w/type=%d\n",type));
1135    if (!(src && dst))
1136      return(-1);
1137
1138    /*
1139     *  For storage purposes, the two esp modes are
1140     *  treated the same.
1141     */
1142    if (type == SS ENCRYPTION NETWORK)
1143      type = SS_ENCRYPTION_TRANSPORT;
1144
1145    done = count = 0;
1146    do {
1147      count++;
1148      /*
1149       *  Currently, valid spi values are 32 bits wide except for
1150       *  the value of zero.  This need to change to take into
1151       *  account more restrictive spi ranges.
1152       *
1153       *  TODO:  Kebe says to allow key mgnt daemon to specify range
```

```
1154        *         of valid spi to get.
1155        */
1156       val = random();
1157       DPRINTF(IDL_FINISHED,("%u ",val));
1158       if (val) {
1159         DPRINTF(IDL_FINISHED,("\n"));
1160         bzero(&buf, sizeof(buf));
1161         len = key_createkey((char *)&buf, type, src, dst, val, 0);
1162         indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
1163         if(!key_search(type, src, dst, val, indx, &prevkeynode)) {
1164       s = splnet();
1165       K_Malloc(secassoc, struct ipsec_assoc *, sizeof(struct
          ipsec_assoc));
1166       if (secassoc == 0) {
1167         DPRINTF(IDL_CRITICAL,("key_getspi: can't allocate memory\n"));
1168         splx(s);
1169         return(-1);
1170       }
1171       bzero((char *)secassoc, sizeof(struct ipsec_assoc));
1172
1173       DPRINTF(IDL_FINISHED,("getspi: indx=%d\n",indx));
1174       secassoc->len = sizeof(struct ipsec_assoc);
1175       secassoc->type = type;
1176       secassoc->spi = val;
1177       secassoc->state |= K_LARVAL;
1178       if (my_addr((struct sockaddr *)&(secassoc->dst)))
1179         secassoc->state |= K_INBOUND;
1180       if (my_addr((struct sockaddr *)&(secassoc->src)))
1181         secassoc->state |= K_OUTBOUND;
1182
1183       bcopy((char *)src, (char *)&(secassoc->src), src->sa_len);
1184       bcopy((char *)dst, (char *)&(secassoc->dst), dst->sa_len);
1185       secassoc->from.sin6_family = AF_INET6;
1186       secassoc->from.sin6_len = sizeof(struct sockaddr_in6);
1187
1188       /*
1189        * We need to add code to age these larval key table
1190        * entries so they don't linger forever waiting for
1191        * a KEY_UPDATE message that may not come for various
1192        * reasons.  This is another task that key_reaper can
1193        * do once we have it coded.
1194        */
1195       secassoc->lifetime1 = time.tv_sec + maxlarvallifetime;
1196
1197       if (!(keynode = key_addnode(indx, secassoc))) {
1198         DPRINTF(IDL_CRITICAL,("key_getspi: can't add node\n"));
1199         splx(s);
1200         return(-1);
1201       }
1202       DPRINTF(IDL_FINISHED,("key_getspi: added node 0x%x\n",keynode));
1203       done++;
1204       splx(s);
1205         }
1206       }
1207     } while ((count < maxcount) && !done);
1208     DPRINTF(IDL_FINISHED,("getspi returns
          w/spi=%u,count=%d\n",val,count));
1209     if (done) {
1210       *spi = val;
1211       return(0);
1212     } else {
1213       *spi = 0;
1214       return(-1);
1215     }
1216   }
1217
1218
```

```
1219  /*-------------------------------------------------------------------------
1220   * key update():
1221   *      Update a keytable entry that has an spi value assigned but is
1222   *      incomplete (e.g. no key/iv).
1223   -------------------------------------------------------------------------*
       /
1224  int
1225  key_update(secassoc)
1226       struct ipsec_assoc *secassoc;
1227  {
1228    struct key tblnode *keynode, *prevkeynode;
1229    struct key allocnode *np = 0;
1230    u_int8 newstate;
1231    int len, indx, inbound, outbound;
1232    char buf[MAXHASHKEYLEN];
1233    int s;
1234
1235    /*
1236     *  For storage purposes, the two esp modes are
1237     *  treated the same.
1238     */
1239    if (secassoc->type == SS_ENCRYPTION_NETWORK)
1240      secassoc->type = SS_ENCRYPTION_TRANSPORT;
1241
1242    bzero(&buf, sizeof(buf));
1243    len = key_createkey((char *)&buf, secassoc->type,
1244                (struct sockaddr *)&(secassoc->src),
1245                (struct sockaddr *)&(secassoc->dst),
1246                secassoc->spi, 0);
1247    indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
1248    if(!(keynode = key_search(secassoc->type,
1249                  (struct sockaddr *)&(secassoc->src),
1250                  (struct sockaddr *)&(secassoc->dst),
1251                  secassoc->spi, indx, &prevkeynode))) {
1252      return(ESRCH);
1253    }
1254    if (keynode->secassoc->state & K_DEAD)
1255      return(ESRCH);
1256
1257    /* Should we also restrict updating of only LARVAL entries ? */
1258
1259    s = splnet();
1260
1261    inbound = my_addr((struct sockaddr *)&(secassoc->dst));
1262    outbound = my_addr((struct sockaddr *)&(secassoc->src));
1263
1264    newstate = keynode->secassoc->state;
1265    newstate &= ~K_LARVAL;
1266    if (inbound)
1267      newstate |= K_INBOUND;
1268    if (outbound)
1269      newstate |= K_OUTBOUND;
1270
1271    if (outbound || (!inbound && !outbound)) {
1272      K_Malloc(np, struct key_allocnode *, sizeof(struct key_allocnode));
1273      if (np == 0) {
1274        DPRINTF(IDL_CRITICAL,("keyupdate: can't allocate allocnode!\n"));
1275        splx(s);
1276        return(ENOBUFS);
1277      }
1278    }
1279
1280    /*
1281     *  We now copy the secassoc over. We don't need to copy
1282     *  the key and iv into new buffers since the calling routine
1283     *  does that already.
1284     */
```

```
1285
1286    *(keynode->secassoc) = *secassoc;
1287    keynode->secassoc->state = newstate;
1288
1289    /*
1290     * Should we allow a null key to be inserted into the table ?
1291     * or can we use null key to indicate some policy action...
1292     */
1293
1294    if (keynode->secassoc->key &&
1295         (keynode->secassoc->type == SS_ENCRYPTION TRANSPORT) &&
1296         ((keynode->secassoc->algorithm == IPSEC ALGTYPE ESP DES_CBC) ||
1297      (keynode->secassoc->algorithm == IPSEC ALGTYPE_ESP_3DES)))
1298        des_set_odd_parity(keynode->secassoc->key);
1299
1300    /*
1301     *  We now add an entry to the allocation table for this
1302     *  updated key table entry.
1303     */
1304    if (outbound || (!inbound && !outbound)) {
1305      len = key createkey((char *)&buf, secassoc->type,
1306              (struct sockaddr *)&(secassoc->src),
1307              (struct sockaddr *)&(secassoc->dst),
1308              0, 1);
1309      indx = key gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
1310      DPRINTF(IDL_FINISHED,("keyupdate: keyalloc hash position=%d\n",
         indx));
1311      np->keynode = keynode;
1312      np->next = keyalloctbl[indx].next;
1313      keyalloctbl[indx].next = np;
1314    }
1315
1316    key deleteacquire(secassoc->type, (struct sockaddr
      *)&(secassoc->dst));
1317
1318    splx(s);
1319    return(0);
1320  }
1321
1322  /*--------------------------------------------------------------------
1323   * key register():
1324   *      Register a socket as one capable of acquiring security
       associations
1325   *      for the kernel.
1326   ------------------------------------------------------------------------*
       /
1327  int
1328  key_register(socket, type)
1329       struct socket *socket;
1330       u_int type;
1331  {
1332    struct key registry *p, *new;
1333    int s = splnet();
1334
1335    DPRINTF(IDL MAJOR_EVENT,("Entering key_register w/so=0x%x,type=%d\n",
      socket,type));
1336
1337    if (!(keyregtable && socket))
1338      panic("key_register");
1339
1340    /*
1341     * Make sure entry is not already in table
1342     */
1343    for(p = keyregtable->next; p; p = p->next) {
1344      if ((p->type == type) && (p->socket == socket)) {
1345        splx(s);
1346        return(EEXIST);
```

```
1347      }
1348    }
1349
1350    K_Malloc(new, struct key_registry *, sizeof(struct key_registry));
1351    if (new == 0) {
1352      splx(s);
1353      return(ENOBUFS);
1354    }
1355    new->type = type;
1356    new->socket = socket;
1357    new->next = keyregtable->next;
1358    keyregtable->next = new;
1359    splx(s);
1360    return(0);
1361  }
1362
1363  /*--------------------------------------------------------------------
1364   * key_unregister():
1365   *      Delete entries from the registry list.
1366   *         allflag = 1 : delete all entries with matching socket
1367   *         allflag = 0 : delete only the entry matching socket and type
1368   --------------------------------------------------------------------*
      /
1369  void
1370  key_unregister(socket, type, allflag)
1371       struct socket *socket;
1372       u_int type;
1373       int allflag;
1374  {
1375    struct key_registry *p, *prev;
1376    int s = splnet();
1377
1378    DPRINTF(IDL_MAJOR_EVENT,("Entering key_unregister
      w/so=0x%x,type=%d,flag=%d\n",socket, type, allflag));
1379
1380    if (!(keyregtable && socket))
1381      panic("key_register");
1382    prev = keyregtable;
1383    for(p = keyregtable->next; p; p = p->next) {
1384      if ((allflag && (p->socket == socket)) ||
1385      ((p->type == type) && (p->socket == socket))) {
1386        prev->next = p->next;
1387        KFree(p);
1388        p = prev;
1389      }
1390      prev = p;
1391    }
1392    splx(s);
1393  }
1394
1395
1396  /*--------------------------------------------------------------------
1397   * key_acquire():
1398   *      Send a key acquire message to all registered key mgnt daemons
1399   *      capable of acquire security association of type type.
1400   *
1401   *      Return: 0 if succesfully called key mgnt. daemon(s)
1402   *              -1 if not successfull.
1403   --------------------------------------------------------------------*
      /
1404  int
1405  key_acquire(type, src, dst)
1406       u_int type;
1407       struct sockaddr *src;
1408       struct sockaddr *dst;
1409  {
1410    struct key_registry *p;
```

```
1411    struct key acquirelist *ap, *prevap;
1412    int success = 0, created = 0;
1413    struct socket *last = 0;
1414    struct mbuf *m = 0;
1415    u int etype;
1416    extern struct sockaddr key_src;
1417
1418    DPRINTF(IDL_MAJOR_EVENT,("Entering key_acquire()\n"));
1419
1420    if (!keyregtable || !src || !dst)
1421      return (-1);
1422
1423    /*
1424     * We first check the acquirelist to see if a key_acquire
1425     * message has been sent for this destination.
1426     */
1427    etype = type;
1428    if (etype == SS ENCRYPTION NETWORK)
1429      etype = SS ENCRYPTION TRANSPORT;
1430    prevap = key acquirelist;
1431    for(ap = key acquirelist->next; ap; ap = ap->next) {
1432      if (addrpart equal(dst, (struct sockaddr *)&(ap->target)) &&
1433      (etype == ap->type)) {
1434        DPRINTF(IDL MAJOR EVENT,("acquire message previously sent!\n"));
1435        if (ap->expiretime < time.tv sec) {
1436      DPRINTF(IDL_MAJOR_EVENT,("acquire message has expired!\n"));
1437      ap->count = 0;
1438      break;
1439        }
1440        if (ap->count < maxkeyacquire) {
1441      DPRINTF(IDL MAJOR_EVENT,("max acquire messages not yet
         exceeded!\n"));
1442      break;
1443        }
1444        return(0);
1445      } else if (ap->expiretime < time.tv_sec) {
1446        /*
1447         *  Since we're already looking at the list, we may as
1448         *  well delete expired entries as we scan through the list.
1449         *  This should really be done by a function like key reaper()
1450         *  but until we code key_reaper(), this is a quick and dirty
1451         *  hack.
1452         */
1453        DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting
         it!\n"));
1454        prevap->next = ap->next;
1455        KFree(ap);
1456        ap = prevap;
1457      }
1458      prevap = ap;
1459    }
1460
1461    /*
1462     * Scan registry and send KEY ACQUIRE message to
1463     * appropriate key management daemons.
1464     */
1465    for(p = keyregtable->next; p; p = p->next) {
1466      if (p->type != type)
1467        continue;
1468
1469      if (!created) {
1470        struct key_msghdr *km;
1471        int len;
1472
1473        len = sizeof(struct key_msghdr) + ROUNDUP(src->sa_len) +
1474      ROUNDUP(dst->sa len);
1475        K_Malloc(km, struct key_msghdr *, len);
```

```
1476        if (km == 0) {
1477      DPRINTF(IDL_CRITICAL,("key_acquire: no memory\n"));
1478      return(-1);
1479        }
1480        DPRINTF(IDL_FINISHED,("key_acquire/created: 1\n"));
1481        bzero((char *)km, len);
1482        km->key msglen = len;
1483        km->key msgvers = KEY VERSION;
1484        km->key msgtype = KEY_ACQUIRE;
1485        km->type = type;
1486        DPRINTF(IDL_FINISHED,("key_acquire/created: 2\n"));
1487        /*
1488         * This is inefficient and slow.
1489         */
1490
1491        /*
1492         * We zero out sin zero here for AF INET addresses because
1493         * ip_output() currently does not do it for performance reasons.
1494         */
1495        if (src->sa family == AF INET)
1496      bzero((char *)(&((struct sockaddr_in *)src)->sin zero),
1497            sizeof(((struct sockaddr in *)src)->sin_zero));
1498        if (dst->sa family == AF INET)
1499      bzero((char *)(&((struct sockaddr_in *)dst)->sin zero),
1500            sizeof(((struct sockaddr_in *)dst)->sin_zero));
1501
1502        bcopy((char *)src, (char *)(km + 1), src->sa len);
1503        bcopy((char *)dst, (char *)((int)(km + 1) + ROUNDUP(src->sa_len)),
1504          dst->sa len);
1505        DPRINTF(IDL FINISHED,("key acquire/created: 3\n"));
1506        MGETHDR(m, M WAIT, MT DATA);
1507        m->m len = m->m_pkthdr.len = 0;
1508        m->m next = 0;
1509        m->m nextpkt = 0;
1510        m->m pkthdr.rcvif = 0;
1511        m copyback(m, 0, len, (caddr_t)km);
1512        KFree(km);
1513        DPRINTF(IDL FINISHED,("key acquire/created: 4\n"));
1514        DDO(IDL FINISHED,dump_mchain(m));
1515        created++;
1516      }
1517      if (last) {
1518        struct mbuf *n;
1519        if (n = m copy(m, 0, (int)M COPYALL)) {
1520      if (sbappendaddr(&last->so_rcv, &key_src, n, (struct mbuf *)0) == 0)
1521        m freem(n);
1522      else {
1523        sorwakeup(last);
1524        success++;
1525      }
1526        }
1527        DPRINTF(IDL_FINISHED,("key_acquire/last: 1\n"));
1528      }
1529      last = p->socket;
1530    }
1531    if (last) {
1532      if (sbappendaddr(&last->so_rcv, &key_src, m, (struct mbuf *)0) == 0)
1533        m freem(m);
1534      else {
1535        sorwakeup(last);
1536        success++;
1537      }
1538      DPRINTF(IDL_FINISHED,("key_acquire/last: 2\n"));
1539    } else
1540      m_freem(m);
1541
1542    /*
```

```
1543     *  Update the acquirelist
1544     */
1545    if (success) {
1546      if (!ap) {
1547        DPRINTF(IDL MAJOR EVENT,("Adding new entry in acquirelist\n"));
1548        K Malloc(ap, struct key_acquirelist *, sizeof(struct
          key acquirelist));
1549        if (ap == 0)
1550      return(success ? 0 : -1);
1551        bzero((char *)ap, sizeof(struct key acquirelist));
1552        bcopy((char *)dst, (char *)&(ap->target), dst->sa_len);
1553        ap->type = etype;
1554        ap->next = key acquirelist->next;
1555        key_acquirelist->next = ap;
1556      }
1557      DPRINTF(IDL_EVENT,("Updating acquire counter and expiration
          time\n"));
1558      ap->count++;
1559      ap->expiretime = time.tv_sec + maxacquiretime;
1560    }
1561    DPRINTF(IDL MAJOR EVENT,("key_acquire: done! success=%d\n",success));
1562    return(success ? 0 : -1);
1563  }
1564
1565  /*-------------------------------------------------------------------------
1566   * key alloc():
1567   *      Allocate a security association to a socket.  A socket
       requesting
1568   *      unique keying (per-socket keying) is assigned a security
       assocation
1569   *      exclusively for its use.  Sockets not requiring unique keying
       are
1570   *      assigned the first security association which may or may not be
1571   *      used by another socket.
1572   -------------------------------------------------------------------------*
       /
1573  int
1574  key_alloc(type, src, dst, socket, unique_key, keynodep)
1575       u int type;
1576       struct sockaddr *src;
1577       struct sockaddr *dst;
1578       struct socket *socket;
1579       u int  unique key;
1580       struct key_tblnode **keynodep;
1581  {
1582    struct key tblnode *keynode;
1583    char buf[MAXHASHKEYLEN];
1584    struct key allocnode *np, *prevnp;
1585    struct key_so2spinode *newnp;
1586    int len;
1587    int indx;
1588
1589    DPRINTF(IDL GROSS EVENT,("Entering key_alloc w/type=%u!\n",type));
1590    if (!(src && dst)) {
1591      DPRINTF(IDL_CRITICAL,("key_alloc: received null src or dst!\n"));
1592      return(-1);
1593    }
1594
1595    /*
1596     *  We treat esp-transport mode and esp-tunnel mode
1597     *  as a single type in the keytable.
1598     */
1599    if (type == SS ENCRYPTION NETWORK)
1600      type = SS_ENCRYPTION_TRANSPORT;
1601
1602    /*
1603     * Search key allocation table
```

```
1604      */
1605     bzero((char *)&buf, sizeof(buf));
1606     len = key_createkey((char *)&buf, type, src, dst, 0, 1);
1607     indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
1608
1609   #define np_type np->keynode->secassoc->type
1610   #define np_state np->keynode->secassoc->state
1611   #define np_src (struct sockaddr *)&(np->keynode->secassoc->src)
1612   #define np_dst (struct sockaddr *)&(np->keynode->secassoc->dst)
1613
1614     prevnp = &keyalloctbl[indx];
1615     for (np = keyalloctbl[indx].next; np; np = np->next) {
1616       if ((type == np_type) && addrpart_equal(src, np_src) &&
1617       addrpart_equal(dst, np_dst) &&
1618       !(np_state & (K_LARVAL | K_DEAD | K_UNIQUE))) {
1619         if (!(unique_key))
1620       break;
1621         if (!(np_state & K_USED))
1622       break;
1623       }
1624       prevnp = np;
1625     }
1626
1627     if (np) {
1628       struct key_so2spinode *newp;
1629       struct socketlist *newsp;
1630       int s = splnet();
1631
1632       DPRINTF(IDL_MAJOR_EVENT,("key_alloc: found node to allocate\n"));
1633       keynode = np->keynode;
1634
1635       K_Malloc(newnp, struct key_so2spinode *, sizeof(struct
          key_so2spinode));
1636       if (newnp == 0) {
1637         DPRINTF(IDL_CRITICAL,("key_alloc: Can't alloc mem for so2spi
          node!\n"));
1638         splx(s);
1639         return(ENOBUFS);
1640       }
1641       K_Malloc(newsp, struct socketlist *, sizeof(struct socketlist));
1642       if (newsp == 0) {
1643         DPRINTF(IDL_CRITICAL,("key_alloc: Can't alloc mem for
          socketlist!\n"));
1644         if (newnp)
1645       KFree(newnp);
1646         splx(s);
1647         return(ENOBUFS);
1648       }
1649
1650       /*
1651        * Add a hash entry into the so2spi table to
1652        * map socket to allocated secassoc.
1653        */
1654       DPRINTF(IDL_GROSS_EVENT,("key_alloc: adding entry to so2spi
          table..."));
1655       newnp->keynode = keynode;
1656       newnp->socket = socket;
1657       newnp->next = so2spitbl[((u_int32)socket) % SO2SPITBLSIZE].next;
1658       so2spitbl[((u_int32)socket) % SO2SPITBLSIZE].next = newnp;
1659       DPRINTF(IDL_GROSS_EVENT,("done\n"));
1660
1661       if (unique_key) {
1662         /*
1663          * Need to remove the allocation entry
1664          * since the secassoc is now unique and
1665          * can't be allocated to any other socket
1666          */
```

```
1667        DPRINTF(IDL_MAJOR_EVENT,("key alloc: making keynode unique..."));
1668        keynode->secassoc->state |= K_UNIQUE;
1669        prevnp->next = np->next;
1670        KFree(np);
1671        DPRINTF(IDL_MAJOR_EVENT,("done\n"));
1672      }
1673      keynode->secassoc->state |= K_USED;
1674      keynode->secassoc->state |= K_OUTBOUND;
1675      keynode->alloc_count++;
1676
1677      /*
1678       * Add socket to list of socket using secassoc.
1679       */
1680      DPRINTF(IDL_GROSS_EVENT,("key_alloc: adding so to solist..."));
1681      newsp->socket = socket;
1682      newsp->next = keynode->solist->next;
1683      keynode->solist->next = newsp;
1684      DPRINTF(IDL_GROSS_EVENT,("done\n"));
1685      *keynodep = keynode;
1686      splx(s);
1687      return(0);
1688    }
1689    *keynodep = NULL;
1690    return(0);
1691  }
1692
1693
1694  /*-----------------------------------------------------------------------
1695   * key_free():
1696   *      Decrement the refcount for a key table entry.  If the entry is
1697   *      marked dead, and the refcount is zero, we go ahead and delete
       it.
1698   -----------------------------------------------------------------------*
       /
1699  void
1700  key_free(keynode)
1701       struct key_tblnode *keynode;
1702  {
1703    DPRINTF(IDL_MAJOR_EVENT,("Entering key_free
       w/keynode=0x%x\n",keynode));
1704    if (!keynode) {
1705      DPRINTF(IDL_CRITICAL,("Warning: key_free got null pointer\n"));
1706      return;
1707    }
1708    (keynode->ref_count)--;
1709    if (keynode->ref_count < 0) {
1710      DPRINTF(IDL_CRITICAL,("Warning: key_free decremented refcount to
       %d\n",keynode->ref_count));
1711    }
1712    if ((keynode->secassoc->state & K_DEAD) && (keynode->ref_count <= 0))
       {
1713      DPRINTF(IDL_MAJOR_EVENT,("key free: calling key_delete\n"));
1714      key_delete(keynode->secassoc);
1715    }
1716  }
1717
1718  /*-----------------------------------------------------------------------
1719   * getassocbyspi():
1720   *      Get a security association for a given type, src, dst, and spi.
1721   *
1722   *      Returns: 0 if sucessfull
1723   *               -1 if error/not found
1724   *
1725   *      Caller must convert spi to host order.  Function assumes spi is

1726   *      in host order!
1727   -----------------------------------------------------------------------*
```

```
1727   /
1728  int
1729  getassocbyspi(type, src, dst, spi, keyentry)
1730       u int type;
1731       struct sockaddr *src;
1732       struct sockaddr *dst;
1733       u int32 spi;
1734       struct key_tblnode **keyentry;
1735  {
1736    char buf[MAXHASHKEYLEN];
1737    int len, indx;
1738    struct key_tblnode *keynode, *prevkeynode = 0;
1739
1740    DPRINTF(IDL_GROSS_EVENT,("Entering getassocbyspi w/type=%u spi=%u\n",
      type,spi));
1741
1742    /*
1743     *  We treat esp-transport mode and esp-tunnel mode
1744     *  as a single type in the keytable.
1745     */
1746    if (type == SS ENCRYPTION NETWORK)
1747      type = SS_ENCRYPTION_TRANSPORT;
1748
1749    *keyentry = NULL;
1750    bzero(&buf, sizeof(buf));
1751    len = key createkey((char *)&buf, type, src, dst, spi, 0);
1752    indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
1753    DPRINTF(IDL FINISHED,("getassocbyspi: indx=%d\n",indx));
1754    DDO(IDL_FINISHED,dump sockaddr(src);dump sockaddr(dst));
1755    keynode = key search(type, src, dst, spi, indx, &prevkeynode);
1756    DPRINTF(IDL GROSS EVENT,("getassocbyspi: keysearch
      ret=0x%x\n",keynode));
1757    if (keynode && !(keynode->secassoc->state & (K DEAD | K LARVAL))) {
1758      DPRINTF(IDL EVENT,("getassocbyspi: found secassoc!\n"));
1759      (keynode->ref count)++;
1760      *keyentry = keynode;
1761    } else {
1762      DPRINTF(IDL MAJOR_EVENT,("getassocbyspi: secassoc not found!\n"));
1763      return (-1);
1764    }
1765    return(0);
1766  }
1767
1768
1769  /*-----------------------------------------------------------------------
1770   * getassocbysocket():
1771   *      Get a security association for a given type, src, dst, and
      socket.
1772   *      If not found, try to allocate one.
1773   *      Returns: 0 if successfull
1774   *               -1 if error condition/secassoc not found (*keyentry =
      NULL)
1775   *                1 if secassoc temporarily unavailable (*keynetry =
      NULL)
1776   *                  (e.g., key mgnt. daemon(s) called)
1777   -----------------------------------------------------------------------*
      /
1778  int
1779  getassocbysocket(type, src, dst, socket, unique_key, keyentry)
1780       u int type;
1781       struct sockaddr *src;
1782       struct sockaddr *dst;
1783       struct socket *socket;
1784       u int unique key;
1785       struct key_tblnode **keyentry;
1786  {
1787    struct key_tblnode *keynode = 0;
```

```
1788    struct key so2spinode *np;
1789    int len, indx;
1790    u int32 spi;
1791    u_int realtype;
1792
1793    DPRINTF(IDL GROSS EVENT,("Entering getassocbysocket w/type=%u
        so=0x%x\n",type,socket));
1794
1795    /*
1796     *  We treat esp-transport mode and esp-tunnel mode
1797     *  as a single type in the keytable.  This has a side
1798     *  effect that socket using both esp-transport and
1799     *  esp-tunnel will use the same security association
1800     *  for both modes.  Is this a problem?
1801     */
1802    realtype = type;
1803    if (type == SS ENCRYPTION NETWORK)
1804      type = SS_ENCRYPTION_TRANSPORT;
1805
1806    if (np = key sosearch(type, src, dst, socket)) {
1807      if (np->keynode && np->keynode->secassoc &&
1808      !(np->keynode->secassoc->state & (K DEAD | K LARVAL))) {
1809        DPRINTF(IDL FINISHED,("getassocbysocket: found secassoc!\n"));
1810        (np->keynode->ref count)++;
1811        *keyentry = np->keynode;
1812        return(0);
1813      }
1814    }
1815
1816    /*
1817     * No secassoc has been allocated to socket,
1818     * so allocate one, if available
1819     */
1820    DPRINTF(IDL EVENT,("getassocbyso: can't find it, trying to
        allocate!\n"));
1821    if (key_alloc(realtype, src, dst, socket, unique_key, &keynode) == 0)
        {
1822      if (keynode) {
1823        DPRINTF(IDL EVENT,("getassocbyso: key_alloc found secassoc!\n"));
1824        keynode->ref count++;
1825        *keyentry = keynode;
1826        return(0);
1827      } else {
1828        /*
1829         * Kick key mgnt. daemon(s)
1830         * (this should be done in ipsec output policy() instead or
1831         * selectively called based on a flag value)
1832         */
1833        DPRINTF(IDL FINISHED,("getassocbyso: calling key mgnt
            daemons!\n"));
1834        *keyentry = NULL;
1835        if (key acquire(realtype, src, dst) == 0)
1836      return (1);
1837        else
1838      return(-1);
1839      }
1840    }
1841    *keyentry = NULL;
1842    return(-1);
1843  }
1844
1845
```