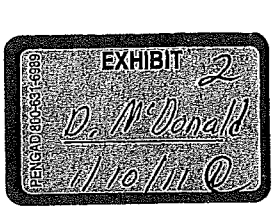
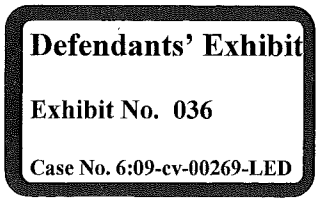
Exhibit 14



DOCUMENT PRODUCED IN NATIVE FORMAT



```
/*---------------------------------------------------------------------
 * key.h :     Declarations and Definitions for Key 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 Agreement 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
```

```
54      without specific prior written permission.
55
56  THE SOFTWARE PROVIDED BY NRL IS PROVIDED BY NRL AND CONTRIBUTORS ``AS
57  IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
58  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
59  PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NRL OR
60  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
61  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
62  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
63  PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
64  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
65  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
66  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
67
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  #ifdef linux
76  #include <netkey/osdep_linux.h>
77  #else /* linux */
78  #include <netkey/osdep_44bsd.h>
79  #endif /* linux */
80
81  /*
82   * PF_KEY messages
83   */
84
85  #define KEY_ADD          1
86  #define KEY_DELETE       2
87  #define KEY_UPDATE       3
88  #define KEY_GET          4
89  #define KEY_ACQUIRE      5
90  #define KEY_GETSPI       6
91  #define KEY_REGISTER     7
92  #define KEY_EXPIRE       8
93  #define KEY_DUMP         9
94  #define KEY_FLUSH        10
95
96  #define KEY_VERSION      1
97  #define POLICY_VERSION   1
98
99  #define SECURITY_TYPE_NONE    0
100
101 #define KEY_TYPE_AH           1
102 #define KEY_TYPE_ESP          2
103 #define KEY_TYPE_RSVP         3
104 #define KEY_TYPE_OSPF         4
105 #define KEY_TYPE_RIPV2        5
106 #define KEY_TYPE_MIPV4        6
```

```
107 #define KEY_TYPE_MIPV6        7
108 #define KEY_TYPE_MAX          7
109
110 /*
111  * Security association state
112  */
113
114 #define K_USED           0x1    /* Key used/not used */
115 #define K_UNIQUE         0x2    /* Key unique/reusable */
116 #define K_LARVAL         0x4    /* SPI assigned, but sa incomplete */
117 #define K_ZOMBIE         0x8    /* sa expired but still useable */
118 #define K_DEAD           0x10   /* sa marked for deletion, ready for reaping */
119 #define K_INBOUND        0x20   /* sa for inbound packets, ie. dst=myhost */
120 #define K_OUTBOUND       0x40   /* sa for outbound packets, ie. src=myhost */
121
122
123 #ifndef MAX_SOCKADDR_SZ
124 #ifdef INET6
125 #define MAX_SOCKADDR_SZ (sizeof(struct sockaddr_in6))
126 #else /* INET6 */
127 #define MAX_SOCKADDR_SZ (sizeof(struct sockaddr_in))
128 #endif /* INET6 */
129 #endif /* MAX_SOCKADDR_SZ */
130
131 #ifndef MAX_KEY_SZ
132 #define MAX_KEY_SZ 16
133 #endif /* MAX_KEY_SZ */
134
135 #ifndef MAX_IV_SZ
136 #define MAX_IV_SZ 16
137 #endif /* MAX_IV_SZ */
138
139 /* Security association data for IP Security */
140 struct key_secassoc {
141   u_int8  len;                  /* Length of the data (for radix) */
142   u_int8  type;                 /* Type of association */
143   u_int8  state;                /* State of the association */
144   u_int8  label;                /* Sensitivity label (unused) */
145   u_int32 spi;                  /* SPI */
146   u_int8  keylen;   /* Key length */
147   u_int8  ivlen;    /* Initialization vector length */
148   u_int8  algorithm;    /* Algorithm switch index */
149   u_int8  lifetype;             /* Type of lifetime */
150   caddr_t iv;     /* Initialization vector */
151   caddr_t key;      /* Key */
152   u_int32 lifetime1;            /* Lifetime value 1 */
153   u_int32 lifetime2;            /* Lifetime value 2 */
154   SOCKADDR *src;                /* Source host address */
155   SOCKADDR *dst;                /* Destination host address */
156   SOCKADDR *from;               /* Originator of association */
157 };
158
159 /*
```

```
160  * Structure for key message header.
161  * PF_KEY message consists of key_msghdr followed by
162  * src sockaddr, dest sockaddr, from sockaddr, key, and iv.
163  * Assumes size of key message header less than MHLEN.
164  */
165
166 struct key_msghdr {
167   u_short key_msglen;   /* length of message including src/dst/from/key/iv */
168   u_char  key_msgvers;  /* key version number */
169   u_char  key_msgtype;  /* key message type, eg. KEY_ADD */
170   pid_t   key_pid;      /* process id of message sender */
171   int     key_seq;      /* message sequence number */
172   int     key_errno;    /* error code */
173   u_int8  type;         /* type of security association */
174   u_int8  state;        /* state of security association */
175   u_int8  label;        /* sensitivity level */
176   u_int8  pad;          /* padding for allignment */
177   u_int32 spi;          /* spi value */
178   u_int8  keylen;       /* key length */
179   u_int8  ivlen;        /* iv length */
180   u_int8  algorithm;    /* algorithm identifier */
181   u_int8  lifetype;     /* type of lifetime */
182   u_int32 lifetime1;    /* lifetime value 1 */
183   u_int32 lifetime2;    /* lifetime value 2 */
184 };
185
186 struct key_msgdata {
187   SOCKADDR *src;    /* source host address */
188   SOCKADDR *dst;    /* destination host address */
189   SOCKADDR *from;   /* originator of security association */
190   caddr_t iv;             /* initialization vector */
191   caddr_t key;            /* key */
192   int ivlen;              /* key length */
193   int keylen;             /* iv length */
194 };
195
196 struct policy_msghdr {
197   u_short policy_msglen;   /* message length */
198   u_char  policy_msgvers;  /* message version */
199   u_char  policy_msgtype;  /* message type */
200   int     policy_seq;      /* message sequence number */
201   int     policy_errno;    /* error code */
202 };
203
204 #ifdef KERNEL
205
206 /*
207  * Key engine table structures
208  */
209
210 struct socketlist {
211   struct socket *socket;     /* pointer to socket */
212   struct socketlist *next;   /* next */
```

```
213 };
214
215 struct key_tblnode {
216   int alloc_count;              /* number of sockets allocated to secassoc */
217   int ref_count;                /* number of sockets referencing secassoc */
218   struct socketlist *solist;    /* list of sockets allocated to secassoc */
219   struct key_secassoc *secassoc; /* security association */
220   struct key_tblnode *next;     /* next node */
221 };
222
223 struct key_allocnode {
224   struct key_tblnode *keynode;
225   struct key_allocnode *next;
226 };
227
228 struct key_so2spinode {
229   struct socket *socket;        /* socket pointer */
230   struct key_tblnode *keynode;  /* pointer to tblnode containing secassoc */
231         /*  info for socket  */
232   struct key_so2spinode *next;
233 };
234
235 struct key_registry {
236   u_int8 type;            /* secassoc type that key mgnt. daemon can acquire */
237   struct socket *socket;  /* key management daemon socket pointer */
238   struct key_registry *next;
239 };
240
241 struct key_acquirelist {
242   u_int8 type;                  /* secassoc type to acquire */
243   SOCKADDR *target;             /* destination address of secassoc */
244   u_int32 count;                /* number of acquire messages sent */
245   u_long expiretime;            /* expiration time for acquire message */
246   struct key_acquirelist *next;
247 };
248
249 struct keyso_cb {
250   int ip4_count;          /* IPv4 */
251 #ifdef INET6
252   int ip6_count;          /* IPv6 */
253 #endif /* INET6 */
254   int any_count;          /* Sum of above counters */
255 };
256
257 #endif
258
259 /*
260  * Useful macros
261  */
262
263 #if 0
264 #define KMALLOC(p, t, n) (p = (t) malloc((unsigned int)(n)))
265 #define KFREE(p) free((char *)p);
```

```
266 #endif /* 0 */
267
268 #ifdef KERNEL
269 #ifdef __P
270 void    key_init __P((void));
271 void    key_cbinit __P((void));
272 int     key_inittables __P((void));
273 int     key_secassoc2msghdr __P((struct key_secassoc *, struct key_msghdr *,
274          struct key_msgdata *));
275 int     key_msghdr2secassoc __P((struct key_secassoc *, struct key_msghdr *,
276          struct key_msgdata *));
277 int     key_add __P((struct key_secassoc *));
278 int     key_delete __P((struct key_secassoc *));
279 int     key_get __P((u_int, SOCKADDR *, SOCKADDR *, u_int32,
280          struct key_secassoc **));
281 void    key_flush __P((void));
282 int     key_dump __P((struct socket *));
283 int     key_getspi __P((u_int, SOCKADDR *, SOCKADDR *,
284              u_int32, u_int32, u_int32 *));
285 int     key_update __P((struct key_secassoc *));
286 int     key_register __P((struct socket *, u_int));
287 void    key_unregister __P((struct socket *, u_int, int));
288 int     key_acquire __P((u_int, SOCKADDR *, SOCKADDR *));
289 int     getassocbyspi __P((u_int, SOCKADDR *, SOCKADDR *,
290               u_int32, struct key_tblnode **));
291 int     getassocbysocket __P((u_int, SOCKADDR *, SOCKADDR *,
292              struct socket *, u_int, struct key_tblnode **));
293 void    key_free __P((struct key_tblnode *));
294 int key_parse __P((struct key_msghdr **km, struct socket *so, int *));
295 #endif /* __P */
296 #endif
```

```
/*-----------------------------------------------------------------------
  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
```

```
54      without specific prior written permission.
55
56  THE SOFTWARE PROVIDED BY NRL IS PROVIDED BY NRL AND CONTRIBUTORS ``AS
57  IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
58  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
59  PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NRL OR
60  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
61  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
62  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
63  PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
64  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
65  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
66  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
67
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  #ifdef linux
76  #include <netkey/osdep_linux.h>
77  #else /* linux */
78  #include <sys/param.h>
79  #include <sys/proc.h>
80  #include <sys/mbuf.h>
81  #include <sys/socket.h>
82  #include <sys/socketvar.h>
83  #include <sys/time.h>
84  #include <sys/kernel.h>
85  #include <net/raw_cb.h>
86  #include <net/if.h>
87  #include <net/if_types.h>
88  #include <net/if_dl.h>
89  #include <net/route.h>
90  #include <netkey/osdep_44bsd.h>
91  #include <netinet/in_var.h>
92  #include <netinet/if_ether.h>
93  #endif /* linux */
94
95  #ifdef INET6
96  #include <netinet6/in6.h>
97  #include <netinet6/in6_var.h>
98  #endif /* INET6 */
99
100 #include <netkey/key.h>
101
102 #include <netkey/key_debug.h>
103
104 #define MAXHASHKEYLEN (2 * sizeof(int) + 2 * sizeof(struct sockaddr_in6))
105
106
```

```
107  /*
108   *  Not clear whether these values should be
109   *  tweakable at kernel config time.
110   */
111  #define KEYTBLSIZE 61
112  #define KEYALLOCTBLSIZE 61
113  #define SO2SPITBLSIZE 61
114
115  /*
116   *  These values should be tweakable...
117   *  perhaps by using sysctl
118   */
119
120  #define MAXLARVALTIME 240;   /* Lifetime of a larval key table entry */
121  #define MAXKEYACQUIRE 1;     /* Max number of key acquire messages sent */
122                               /*   per destination address               */
123  #define MAXACQUIRETIME 15;   /* Lifetime of acquire message */
124
125  /*
126   *  Key engine tables and global variables
127   */
128
129  struct key_tblnode keytable[KEYTBLSIZE];
130  struct key_allocnode keyalloctbl[KEYALLOCTBLSIZE];
131  struct key_so2spinode so2spitbl[SO2SPITBLSIZE];
132
133  struct keyso_cb keyso_cb;
134  struct key_tblnode nullkeynode  = { 0, 0, 0, 0, 0 };
135  struct key_registry *keyregtable;
136  struct key_acquirelist *key_acquirelist;
137  u_long maxlarvallifetime = MAXLARVALTIME;
138  int maxkeyacquire = MAXKEYACQUIRE;
139  u_long maxacquiretime = MAXACQUIRETIME;
140
141  extern SOCKADDR key_addr;
142
143  #define ROUNDUP(a) \
144    ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
145  #define ADVANCE(x, n) \
146      { x += ROUNDUP(n); }
147
148  int my_addr __P((SOCKADDR *));
149  int key_sendup __P((struct socket *, struct key_msghdr *));
150
151  /*-------------------------------------------------------------------
152   * key_secassoc2msghdr():
153   *      Copy info from a security association into a key message buffer.
154   *      Assume message buffer is sufficiently large to hold all security
155   *      association information including src, dst, from, key and iv.
156   -------------------------------------------------------------------*/
157  int key_secassoc2msghdr DEFARGS((secassoc, km, keyinfo),
158  struct key_secassoc *secassoc AND
159  struct key_msghdr *km AND
```

```
160  struct key_msgdata *keyinfo)
161  {
162    char *cp;
163    DPRINTF(IDL_FINISHED, ("Entering key_secassoc2msghdr\n"));
164
165    if ((km == 0) || (keyinfo == 0) || (secassoc == 0))
166      return(-1);
167
168    km->type = secassoc->type;
169    km->state = secassoc->state;
170    km->label = secassoc->label;
171    km->spi = secassoc->spi;
172    km->keylen = secassoc->keylen;
173    km->ivlen = secassoc->ivlen;
174    km->algorithm = secassoc->algorithm;
175    km->lifetype = secassoc->lifetype;
176    km->lifetime1 = secassoc->lifetime1;
177    km->lifetime2 = secassoc->lifetime2;
178
179    /*
180     *  Stuff src/dst/from/key/iv in buffer after
181     *  the message header.
182     */
183    cp = (char *)(km + 1);
184
185    DPRINTF(IDL_FINISHED, ("sa2msghdr: 1\n"));
186    keyinfo->src = (SOCKADDR *)cp;
187    if (secassoc->src->sa_len) {
188      bcopy(secassoc->src, cp, secassoc->src->sa_len);
189      ADVANCE(cp, secassoc->src->sa_len);
190    } else {
191      bzero(cp, MAX_SOCKADDR_SZ);
192      ADVANCE(cp, MAX_SOCKADDR_SZ);
193    }
194
195    DPRINTF(IDL_FINISHED, ("sa2msghdr: 2\n"));
196    keyinfo->dst = (SOCKADDR *)cp;
197    if (secassoc->dst->sa_len) {
198      bcopy(secassoc->dst, cp, secassoc->dst->sa_len);
199      ADVANCE(cp, secassoc->dst->sa_len);
200    } else {
201      bzero(cp, MAX_SOCKADDR_SZ);
202      ADVANCE(cp, MAX_SOCKADDR_SZ);
203    }
204
205    DPRINTF(IDL_FINISHED, ("sa2msghdr: 3\n"));
206    keyinfo->from = (SOCKADDR *)cp;
207    if (secassoc->from->sa_len) {
208      bcopy(secassoc->from, cp, secassoc->from->sa_len);
209      ADVANCE(cp, secassoc->from->sa_len);
210    } else {
211      bzero(cp, MAX_SOCKADDR_SZ);
212      ADVANCE(cp, MAX_SOCKADDR_SZ);
```

```
213    }
214
215    DPRINTF(IDL_FINISHED, ("sa2msghdr: 4\n"));
216
217    keyinfo->key = cp;
218    keyinfo->keylen = secassoc->keylen;
219    if (secassoc->keylen) {
220      bcopy((char *)(secassoc->key), cp, secassoc->keylen);
221      ADVANCE(cp, secassoc->keylen);
222    }
223
224    DPRINTF(IDL_FINISHED, ("sa2msghdr: 5\n"));
225    keyinfo->iv = cp;
226    keyinfo->ivlen = secassoc->ivlen;
227    if (secassoc->ivlen) {
228      bcopy((char *)(secassoc->iv), cp, secassoc->ivlen);
229      ADVANCE(cp, secassoc->ivlen);
230    }
231
232    DDO(IDL_FINISHED,printf("msgbuf(len=%d):\n",(char *)cp - (char *)km));
233    DDO(IDL_FINISHED,dump_buf((char *)km, (char *)cp - (char *)km));
234    DPRINTF(IDL_FINISHED, ("sa2msghdr: 6\n"));
235    return(0);
236  }
237
238
239  /*-------------------------------------------------------------------------
240   * key_msghdr2secassoc():
241   *      Copy info from a key message buffer into a key_secassoc
242   *      structure
243   -------------------------------------------------------------------------*/
244  int key_msghdr2secassoc DEFARGS((secassoc, km, keyinfo),
245  struct key_secassoc *secassoc AND
246  struct key_msghdr *km AND
247  struct key_msgdata *keyinfo)
248  {
249    DPRINTF(IDL_FINISHED, ("Entering key_msghdr2secassoc\n"));
250
251    if ((km == 0) || (keyinfo == 0) || (secassoc == 0))
252      return(-1);
253
254    secassoc->len = sizeof(*secassoc);
255    secassoc->type = km->type;
256    secassoc->state = km->state;
257    secassoc->label = km->label;
258    secassoc->spi = km->spi;
259    secassoc->keylen = km->keylen;
260    secassoc->ivlen = km->ivlen;
261    secassoc->algorithm = km->algorithm;
262    secassoc->lifetype = km->lifetype;
263    secassoc->lifetime1 = km->lifetime1;
264    secassoc->lifetime2 = km->lifetime2;
265
```

```
266   if (keyinfo->src) {
267     KMALLOC(secassoc->src, SOCKADDR *, keyinfo->src->sa_len);
268     if (!secassoc->src) {
269       DPRINTF(IDL_ERROR,("msghdr2secassoc: can't allocate mem for src\n"));
270       return(-1);
271     }
272     bcopy((char *)keyinfo->src, (char *)secassoc->src,
273     keyinfo->src->sa_len);
274   } else
275     secassoc->src = NULL;
276
277   if (keyinfo->dst) {
278     KMALLOC(secassoc->dst, SOCKADDR *, keyinfo->dst->sa_len);
279     if (!secassoc->dst) {
280       DPRINTF(IDL_ERROR,("msghdr2secassoc: can't allocate mem for dst\n"));
281       return(-1);
282     }
283     bcopy((char *)keyinfo->dst, (char *)secassoc->dst,
284     keyinfo->dst->sa_len);
285   } else
286     secassoc->dst = NULL;
287
288   if (keyinfo->from) {
289     KMALLOC(secassoc->from, SOCKADDR *, keyinfo->from->sa_len);
290     if (!secassoc->from) {
291       DPRINTF(IDL_ERROR,("msghdr2secassoc: can't allocate mem for from\n"));
292       return(-1);
293     }
294     bcopy((char *)keyinfo->from, (char *)secassoc->from,
295     keyinfo->from->sa_len);
296   } else
297     secassoc->from = NULL;
298
299   /*
300    *  Make copies of key and iv
301    */
302   if (secassoc->ivlen) {
303     KMALLOC(secassoc->iv, caddr_t, secassoc->ivlen);
304     if (secassoc->iv == 0) {
305       DPRINTF(IDL_ERROR,("msghdr2secassoc: can't allocate mem for iv\n"));
306       return(-1);
307     }
308     bcopy((char *)keyinfo->iv, (char *)secassoc->iv, secassoc->ivlen);
309   } else
310     secassoc->iv = NULL;
311
312   if (secassoc->keylen) {
313     KMALLOC(secassoc->key, caddr_t, secassoc->keylen);
314     if (secassoc->key == 0) {
315       DPRINTF(IDL_ERROR,("msghdr2secassoc: can't allocate mem for key\n"));
316       if (secassoc->iv)
317   KFREE(secassoc->iv);
318       return(-1);
```

```
319       }
320       bcopy((char *)keyinfo->key, (char *)secassoc->key, secassoc->keylen);
321     } else
322       secassoc->key = NULL;
323     return(0);
324   }
325
326
327   /*------------------------------------------------------------------------
328    * addrpart_equal():
329    *      Determine if the address portion of two sockaddrs are equal.
330    *      Currently handles only AF_INET and AF_INET6 address families.
331    ------------------------------------------------------------------------*/
332   int addrpart_equal DEFARGS((sa1, sa2),
333   SOCKADDR *sa1 AND
334   SOCKADDR *sa2)
335   {
336     if ((sa1->sa_family != sa2->sa_family) ||
337         (sa1->sa_len != sa2->sa_len))
338       return 0;
339
340     switch(sa1->sa_family) {
341     case AF_INET:
342       return (((struct sockaddr_in *)sa1)->sin_addr.s_addr ==
343         ((struct sockaddr_in *)sa2)->sin_addr.s_addr);
344   #ifdef INET6
345     case AF_INET6:
346       return (IN6_ADDR_EQUAL(((struct sockaddr_in6 *)sa1)->sin6_addr,
347           ((struct sockaddr_in6 *)sa2)->sin6_addr));
348   #endif /* INET6 */
349     }
350     return(0);
351   }
352
353   /*------------------------------------------------------------------------
354    * key_inittables():
355    *      Allocate space and initialize key engine tables
356    ------------------------------------------------------------------------*/
357   int key_inittables __P((void))
358   {
359     int i;
360
361     KMALLOC(keyregtable, struct key_registry *, sizeof(struct key_registry));
362     if (!keyregtable)
363       return -1;
364     bzero((char *)keyregtable, sizeof(struct key_registry));
365     KMALLOC(key_acquirelist, struct key_acquirelist *,
366        sizeof(struct key_acquirelist));
367     if (!key_acquirelist)
368       return -1;
369     bzero((char *)key_acquirelist, sizeof(struct key_acquirelist));
370     for (i = 0; i < KEYTBLSIZE; i++)
371       bzero((char *)&keytable[i], sizeof(struct key_tblnode));
```

```
372   for (i = 0; i < KEYALLOCTBLSIZE; i++)
373     bzero((char *)&keyalloctbl[i], sizeof(struct key_allocnode));
374   for (i = 0; i < SO2SPITBLSIZE; i++)
375     bzero((char *)&so2spitbl[i], sizeof(struct key_so2spinode));
376
377   return 0;
378 }
379
380 int key_freetables __P((void))
381 {
382   KFREE(keyregtable);
383   keyregtable = NULL;
384   KFREE(key_acquirelist);
385   key_acquirelist = NULL;
386   return 0;
387 }
388
389 /*-------------------------------------------------------------------------
390  * key_gethashval():
391  *      Determine keytable hash value.
392  -------------------------------------------------------------------------*/
393 int key_gethashval DEFARGS((buf, len, tblsize),
394 char *buf AND
395 int len AND
396 int tblsize)
397 {
398   int i, j = 0;
399
400   /*
401    * Todo: Use word size xor and check for alignment
402    *       and zero pad if necessary.  Need to also pick
403    *       a good hash function and table size.
404    */
405   if (len <= 0) {
406     DPRINTF(IDL_ERROR,("key_gethashval got bogus len!\n"));
407     return(-1);
408   }
409   for(i = 0; i < len; i++) {
410     j ^=  (u_int8)(*(buf + i));
411   }
412   return (j % tblsize);
413 }
414
415
416 /*-------------------------------------------------------------------------
417  * key_createkey():
418  *      Create hash key for hash function
419  *      key is: type+src+dst if keytype = 1
420  *              type+src+dst+spi if keytype = 0
421  *      Uses only the address portion of the src and dst sockaddrs to
422  *      form key.  Currently handles only AF_INET and AF_INET6 sockaddrs
423  -------------------------------------------------------------------------*/
424 int key_createkey DEFARGS((buf, type, src, dst, spi, keytype),
```

```
425       char *buf AND
426       u_int type AND
427       SOCKADDR *src AND
428       SOCKADDR *dst AND
429       u_int32 spi AND
430       u_int keytype)
431  {
432    char *cp, *p;
433
434    DPRINTF(IDL_FINISHED,("Entering key_createkey\n"));
435
436    if (!buf || !src || !dst)
437      return(-1);
438
439    cp = buf;
440    bcopy((char *)&type, cp, sizeof(type));
441    cp += sizeof(type);
442
443  #ifdef INET6
444    /*
445     * Assume only IPv4 and IPv6 addresses.
446     */
447  #define ADDRPART(a) \
448      ((a)->sa_family == AF_INET6) ? \
449      (char *)&(((struct sockaddr_in6 *)(a))->sin6_addr) : \
450      (char *)&(((struct sockaddr_in *)(a))->sin_addr)
451
452  #define ADDRSIZE(a) \
453      ((a)->sa_family == AF_INET6) ? sizeof(struct in_addr6) : \
454      sizeof(struct in_addr)
455  #else /* INET6 */
456  #define ADDRPART(a) (char *)&(((struct sockaddr_in *)(a))->sin_addr)
457  #define ADDRSIZE(a) sizeof(struct in_addr)
458  #endif /* INET6 */
459
460    DPRINTF(IDL_FINISHED,("src addr:\n"));
461    DDO(IDL_FINISHED,dump_smart_sockaddr(src));
462    DPRINTF(IDL_FINISHED,("dst addr:\n"));
463    DDO(IDL_FINISHED,dump_smart_sockaddr(dst));
464
465    p = ADDRPART(src);
466    bcopy(p, cp, ADDRSIZE(src));
467    cp += ADDRSIZE(src);
468
469    p = ADDRPART(dst);
470    bcopy(p, cp, ADDRSIZE(dst));
471    cp += ADDRSIZE(dst);
472
473  #undef ADDRPART
474  #undef ADDRSIZE
475
476    if (keytype == 0) {
477      bcopy((char *)&spi, cp, sizeof(spi));
```

```
478       cp += sizeof(spi);
479     }
480
481     DPRINTF(IDL_FINISHED,("hash key:\n"));
482     DDO(IDL_FINISHED, dump_buf(buf, cp - buf));
483     return(cp - buf);
484   }
485
486
487   /*-----------------------------------------------------------------------
488    * key_sosearch():
489    *       Search the so2spi table for the security association allocated to
490    *       the socket.  Returns pointer to a struct key_so2spinode which can
491    *       be used to locate the security association entry in the keytable.
492    -----------------------------------------------------------------------*/
493   struct key_so2spinode *key_sosearch DEFARGS((type, src, dst, so),
494        u_int type AND
495        SOCKADDR *src AND
496        SOCKADDR *dst AND
497        struct socket *so)
498   {
499     struct key_so2spinode *np = 0;
500
501     if (!(src && dst)) {
502       DPRINTF(IDL_ERROR,("key_sosearch: got null src or dst pointer!\n"));
503       return(NULL);
504     }
505
506     for (np = so2spitbl[((u_int32)so) % SO2SPITBLSIZE].next; np; np = np->next) {
507       if ((so == np->socket) && (type == np->keynode->secassoc->type)
508     && addrpart_equal(src, np->keynode->secassoc->src)
509     && addrpart_equal(dst, np->keynode->secassoc->dst))
510         return(np);
511     }
512     return(NULL);
513   }
514
515
516   /*-----------------------------------------------------------------------
517    * key_sodelete():
518    *       Delete entries from the so2spi table.
519    *         flag = 1  purge all entries
520    *         flag = 0  delete entries with socket pointer matching socket
521    -----------------------------------------------------------------------*/
522   void key_sodelete DEFARGS((socket, flag),
523        struct socket *socket AND
524        int flag)
525   {
526     struct key_so2spinode *prevnp, *np;
527     CRITICAL_DCL
528
529     CRITICAL_START;
530
```

```
531    DPRINTF(IDL_EVENT,("Entering keysodelete w/so=0x%x flag=%d\n",
532            (unsigned int)socket,flag));
533
534    if (flag) {
535      int i;
536
537      for (i = 0; i < SO2SPITBLSIZE; i++)
538        for(np = so2spitbl[i].next; np; np = np->next) {
539    KFREE(np);
540        }
541      CRITICAL_END;
542      return;
543    }
544
545    prevnp = &so2spitbl[((u_int32)socket) % SO2SPITBLSIZE];
546    for(np = prevnp->next; np; np = np->next) {
547      if (np->socket == socket) {
548        struct socketlist *socklp, *prevsocklp;
549
550        (np->keynode->alloc_count)--;
551
552        /*
553         * If this socket maps to a unique secassoc,
554         * we go ahead and delete the secassoc, since it
555         * can no longer be allocated or used by any other
556         * socket.
557         */
558        if (np->keynode->secassoc->state & K_UNIQUE) {
559    if (key_delete(np->keynode->secassoc) != 0)
560      panic("key_sodelete");
561    np = prevnp;
562    continue;
563        }
564
565        /*
566         * We traverse the socketlist and remove the entry
567         * for this socket
568         */
569        DPRINTF(IDL_FINISHED,("keysodelete: deleting from socklist..."));
570        prevsocklp = np->keynode->solist;
571        for (socklp = prevsocklp->next; socklp; socklp = socklp->next) {
572    if (socklp->socket == socket) {
573      prevsocklp->next = socklp->next;
574      KFREE(socklp);
575      break;
576    }
577    prevsocklp = socklp;
578        }
579        DPRINTF(IDL_FINISHED,("done\n"));
580        prevnp->next = np->next;
581        KFREE(np);
582        np = prevnp;
583      }
```

```
584       prevnp = np;
585     }
586     CRITICAL_END;
587   }
588
589
590   /*-----------------------------------------------------------------------
591    * key_deleteacquire():
592    *       Delete an entry from the key_acquirelist
593    -----------------------------------------------------------------------*/
594   void key_deleteacquire DEFARGS((type, target),
595       u_int type AND
596       SOCKADDR *target)
597   {
598     struct key_acquirelist *ap, *prev;
599
600     prev = key_acquirelist;
601     for(ap = key_acquirelist->next; ap; ap = ap->next) {
602       if (addrpart_equal(target, (SOCKADDR *)&(ap->target)) &&
603     (type == ap->type)) {
604         DPRINTF(IDL_EVENT,("Deleting entry from acquire list!\n"));
605         prev->next = ap->next;
606         KFREE(ap);
607         ap = prev;
608       }
609       prev = ap;
610     }
611   }
612
613
614   /*-----------------------------------------------------------------------
615    * key_search():
616    *       Search the key table for an entry with same type, src addr, dest
617    *       addr, and spi.  Returns a pointer to struct key_tblnode if found
618    *       else returns null.
619    -----------------------------------------------------------------------*/
620   struct key_tblnode *key_search DEFARGS(
621               (type, src, dst, spi, indx, prevkeynode),
622       u_int type AND
623       SOCKADDR *src AND
624       SOCKADDR *dst AND
625       u_int32 spi AND
626       int indx AND
627       struct key_tblnode **prevkeynode)
628   {
629     struct key_tblnode *keynode, *prevnode;
630
631     if (indx > KEYTBLSIZE || indx < 0)
632       return (NULL);
633     if (!(&keytable[indx]))
634       return (NULL);
635
636   #define sec_type keynode->secassoc->type
```

```
637 #define sec_spi keynode->secassoc->spi
638 #define sec_src keynode->secassoc->src
639 #define sec_dst keynode->secassoc->dst
640
641   prevnode = &keytable[indx];
642   for (keynode = keytable[indx].next; keynode; keynode = keynode->next) {
643     if ((type == sec_type) && (spi == sec_spi) &&
644   addrpart_equal(src, sec_src)
645   && addrpart_equal(dst, sec_dst))
646       break;
647     prevnode = keynode;
648   }
649   *prevkeynode = prevnode;
650   return(keynode);
651 }
652
653
654 /*---------------------------------------------------------------------
655  * key_addnode():
656  *      Insert a key_tblnode entry into the key table.  Returns a pointer
657  *      to the newly created key_tblnode.
658  ---------------------------------------------------------------------*/
659 struct key_tblnode *key_addnode DEFARGS((indx, secassoc),
660      int indx AND
661      struct key_secassoc *secassoc)
662 {
663   struct key_tblnode *keynode;
664
665   DPRINTF(IDL_FINISHED,("Entering key_addnode w/indx=%d secassoc=0x%x\n",
666       indx, (unsigned int)secassoc));
667
668   if (!(&keytable[indx]))
669     return(NULL);
670   if (!secassoc) {
671     panic("key_addnode: Someone passed in a null secassoc!\n");
672   }
673
674   KMALLOC(keynode, struct key_tblnode *, sizeof(struct key_tblnode));
675   if (keynode == 0)
676     return(NULL);
677   bzero((char *)keynode, sizeof(struct key_tblnode));
678
679   KMALLOC(keynode->solist, struct socketlist *, sizeof(struct socketlist));
680   if (keynode->solist == 0) {
681     KFREE(keynode);
682     return(NULL);
683   }
684   bzero((char *)(keynode->solist), sizeof(struct socketlist));
685
686   keynode->secassoc = secassoc;
687   keynode->solist->next = NULL;
688   keynode->next = keytable[indx].next;
689   keytable[indx].next = keynode;
```

```
690     return(keynode);
691   }
692
693
694   /*-----------------------------------------------------------------------
695    * key_add():
696    *      Add a new security association to the key table.  Caller is
697    *      responsible for allocating memory for the key_secassoc as
698    *      well as the buffer space for the key and iv.  Assumes the security
699    *      association passed in is well-formed.
700    -----------------------------------------------------------------------*/
701   int
702   key_add DEFARGS((secassoc),
703        struct key_secassoc *secassoc)
704   {
705     char buf[MAXHASHKEYLEN];
706     int len, indx;
707     int inbound = 0;
708     int outbound = 0;
709     struct key_tblnode *keynode, *prevkeynode;
710     struct key_allocnode *np = NULL;
711     CRITICAL_DCL
712
713     DPRINTF(IDL_FINISHED, ("Entering key_add w/secassoc=0x%x\n",
714          (unsigned int)secassoc));
715
716     if (!secassoc) {
717       panic("key_add: who the hell is passing me a null pointer");
718     }
719
720     /*
721      * Should we allow a null key to be inserted into the table ?
722      * or can we use null key to indicate some policy action...
723      */
724
725   #if 0
726     /*
727      *  For esp using des-cbc or tripple-des we call
728      * des_set_odd_parity.
729      */
730     if (secassoc->key && (secassoc->type == KEY_TYPE_ESP) &&
731         ((secassoc->algorithm == IPSEC_ALGTYPE_ESP_DES_CBC) ||
732          (secassoc->algorithm == IPSEC_ALGTYPE_ESP_3DES)))
733       des_set_odd_parity(secassoc->key);
734   #endif /* 0 */
735
736     /*
737      *  Check if secassoc with same spi exists before adding
738      */
739     bzero((char *)&buf, sizeof(buf));
740     len = key_createkey((char *)&buf, secassoc->type, secassoc->src,
741             secassoc->dst, secassoc->spi, 0);
742     indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
```

```
743     DPRINTF(IDL_FINISHED,("keyadd: keytbl hash position=%d\n", indx));
744     keynode = key_search(secassoc->type, secassoc->src, secassoc->dst,
745             secassoc->spi, indx, &prevkeynode);
746     if (keynode) {
747       DPRINTF(IDL_EVENT,("keyadd: secassoc already exists!\n"));
748       return(-2);
749     }
750
751     inbound = my_addr(secassoc->dst);
752     outbound = my_addr(secassoc->src);
753     DPRINTF(IDL_FINISHED,("inbound=%d outbound=%d\n", inbound, outbound));
754
755     /*
756      * We allocate mem for an allocation entry if needed.
757      * This is done here instead of in the allocaton code
758      * segment so that we can easily recover/cleanup from a
759      * memory allocation error.
760      */
761     if (outbound || (!inbound && !outbound)) {
762       KMALLOC(np, struct key_allocnode *, sizeof(struct key_allocnode));
763       if (np == 0) {
764         DPRINTF(IDL_ERROR,("keyadd: can't allocate allocnode!\n"));
765         return(-1);
766       }
767     }
768
769     CRITICAL_START;
770
771     if ((keynode = key_addnode(indx, secassoc)) == NULL) {
772       DPRINTF(IDL_ERROR,("keyadd: key_addnode failed!\n"));
773       if (np)
774         KFREE(np);
775       CRITICAL_END;
776       return(-1);
777     }
778     DPRINTF(IDL_GROSS_EVENT,("Added new keynode:\n"));
779     DDO(IDL_FINISHED, dump_keytblnode(keynode));
780     DDO(IDL_FINISHED, dump_secassoc(keynode->secassoc));
781
782     /*
783      *  We add an entry to the allocation table for
784      *  this secassoc if the interfaces are up and
785      *  the secassoc is outbound.  In the case
786      *  where the interfaces are not up, we go ahead
787      *  and do it anyways.  This wastes an allocation
788      *  entry if the secassoc later turned out to be
789      *  inbound when the interfaces are ifconfig up.
790      */
791     if (outbound || (!inbound && !outbound)) {
792       len = key_createkey((char *)&buf, secassoc->type, secassoc->src,
793         secassoc->dst, 0, 1);
794       indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
795       DPRINTF(IDL_FINISHED,("keyadd: keyalloc hash position=%d\n", indx));
```

```
796       np->keynode = keynode;
797       np->next = keyalloctbl[indx].next;
798       keyalloctbl[indx].next = np;
799     }
800     if (inbound)
801       secassoc->state |= K_INBOUND;
802     if (outbound)
803       secassoc->state |= K_OUTBOUND;
804
805     key_deleteacquire(secassoc->type, secassoc->dst);
806
807     CRITICAL_END;
808     return 0;
809   }
810
811
812   /*-----------------------------------------------------------------------
813    * key_get():
814    *      Get a security association from the key table.
815    -----------------------------------------------------------------------*/
816   int key_get DEFARGS((type, src, dst, spi, secassoc),
817        u_int type AND
818        SOCKADDR *src AND
819        SOCKADDR *dst AND
820        u_int32 spi AND
821        struct key_secassoc **secassoc)
822   {
823     char buf[MAXHASHKEYLEN];
824     struct key_tblnode *keynode, *prevkeynode;
825     int len, indx;
826
827     bzero(&buf, sizeof(buf));
828     *secassoc = NULL;
829     len = key_createkey((char *)&buf, type, src, dst, spi, 0);
830     indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
831     DPRINTF(IDL_FINISHED,("keyget: indx=%d\n",indx));
832     keynode = key_search(type, src, dst, spi, indx, &prevkeynode);
833     if (keynode) {
834       DPRINTF(IDL_GROSS_EVENT,("keyget: found it! keynode=0x%x",
835              (unsigned int)keynode));
836       *secassoc = keynode->secassoc;
837       return(0);
838     } else
839       return(-1);  /* Not found */
840   }
841
842
843   /*-----------------------------------------------------------------------
844    * key_dump():
845    *      Dump all valid entries in the keytable to a pf_key socket.  Each
846    *      security associaiton is sent one at a time in a pf_key message.  A
847    *      message with seqno = 0 signifies the end of the dump transaction.
848    -----------------------------------------------------------------------*/
```

```
849  int key_dump DEFARGS((so),
850       struct socket *so)
851  {
852    int len, i;
853    int seq = 1;
854    struct key_msgdata keyinfo;
855    struct key_msghdr *km;
856    struct key_tblnode *keynode;
857    extern SOCKADDR key_addr;
858
859    /*
860     * Routine to dump the key table to a routing socket
861     * Use for debugging only!
862     */
863
864    KMALLOC(km, struct key_msghdr *, sizeof(struct key_msghdr) +
865      3 * MAX_SOCKADDR_SZ + MAX_KEY_SZ + MAX_IV_SZ);
866    if (!km)
867      return(ENOBUFS);
868
869    DPRINTF(IDL_FINISHED,("Entering key_dump()"));
870    /*
871     * We need to speed this up later.  Fortunately, key_dump
872     * messages are not sent often.
873     */
874    for (i = 0; i < KEYTBLSIZE; i++) {
875      for (keynode = keytable[i].next; keynode; keynode = keynode->next) {
876        /*
877         * We exclude dead/larval/zombie security associations for now
878         * but it may be useful to also send these up for debugging purposes
879         */
880        if (keynode->secassoc->state & (K_DEAD | K_LARVAL | K_ZOMBIE))
881    continue;
882
883        len = (sizeof(struct key_msghdr) +
884         ROUNDUP(keynode->secassoc->src->sa_len) +
885         ROUNDUP(keynode->secassoc->dst->sa_len) +
886         ROUNDUP(keynode->secassoc->from->sa_len) +
887         ROUNDUP(keynode->secassoc->keylen) +
888         ROUNDUP(keynode->secassoc->ivlen));
889
890        if (key_secassoc2msghdr(keynode->secassoc, km, &keyinfo) != 0)
891    panic("key_dump");
892
893        km->key_msglen = len;
894        km->key_msgvers = KEY_VERSION;
895        km->key_msgtype = KEY_DUMP;
896        km->key_pid = CURRENT_PID;
897        km->key_seq = seq++;
898        km->key_errno = 0;
899
900        key_sendup(so, km);
901      }
```

```
902     }
903     bzero((char *)km, sizeof(struct key_msghdr));
904     km->key_msglen = sizeof(struct key_msghdr);
905     km->key_msgvers = KEY_VERSION;
906     km->key_msgtype = KEY_DUMP;
907     km->key_pid = CURRENT_PID;
908     km->key_seq = 0;
909     km->key_errno = 0;
910
911     key_sendup(so, km);
912     KFREE(km);
913     DPRINTF(IDL_FINISHED,("Leaving key_dump()\n"));
914     return(0);
915   }
916
917   /*---------------------------------------------------------------------
918    * key_delete():
919    *      Delete a security association from the key table.
920    ---------------------------------------------------------------------*/
921   int key_delete DEFARGS((secassoc),
922       struct key_secassoc *secassoc)
923   {
924     char buf[MAXHASHKEYLEN];
925     int len, indx;
926     struct key_tblnode *keynode = 0;
927     struct key_tblnode *prevkeynode = 0;
928     struct socketlist *socklp, *deadsocklp;
929     struct key_so2spinode *np, *prevnp;
930     struct key_allocnode *ap, *prevap;
931     CRITICAL_DCL
932
933     DPRINTF(IDL_FINISHED,("Entering key_delete w/secassoc=0x%x\n",
934         (unsigned int)secassoc));
935
936     bzero((char *)&buf, sizeof(buf));
937     len = key_createkey((char *)&buf, secassoc->type, secassoc->src,
938             secassoc->dst, secassoc->spi, 0);
939     indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
940     DPRINTF(IDL_FINISHED,("keydelete: keytbl hash position=%d\n", indx));
941     keynode = key_search(secassoc->type, secassoc->src, secassoc->dst,
942              secassoc->spi, indx, &prevkeynode);
943
944     if (keynode) {
945       CRITICAL_START;
946       DPRINTF(IDL_GROSS_EVENT,("keydelete: found keynode to delete\n"));
947       keynode->secassoc->state |= K_DEAD;
948
949       if (keynode->ref_count > 0) {
950         DPRINTF(IDL_EVENT,("keydelete: secassoc still held, marking for deletion
          only!\n"));
951         CRITICAL_END;
952         return(0);
953       }
```

```
954
955       prevkeynode->next = keynode->next;
956
957       /*
958        *  Walk the socketlist and delete the
959        *  entries mapping sockets to this secassoc
960        *  from the so2spi table.
961        */
962       DPRINTF(IDL_FINISHED,("keydelete: deleting socklist..."));
963       for(socklp = keynode->solist->next; socklp; ) {
964         prevnp = &so2spitbl[((u_int32)(socklp->socket)) % SO2SPITBLSIZE];
965         for(np = prevnp->next; np; np = np->next) {
966     if ((np->socket == socklp->socket) && (np->keynode == keynode)) {
967       prevnp->next = np->next;
968       KFREE(np);
969       break;
970     }
971     prevnp = np;
972         }
973         deadsocklp = socklp;
974         socklp = socklp->next;
975         KFREE(deadsocklp);
976       }
977       DPRINTF(IDL_FINISHED,("done\n"));
978       /*
979        * If an allocation entry exist for this
980        * secassoc, delete it.
981        */
982       bzero((char *)&buf, sizeof(buf));
983       len = key_createkey((char *)&buf, secassoc->type, secassoc->src,
984         secassoc->dst, 0, 1);
985       indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
986       DPRINTF(IDL_FINISHED,("keydelete: alloctbl hash position=%d\n", indx));
987       prevap = &keyalloctbl[indx];
988       for (ap = prevap->next; ap; ap = ap->next) {
989         if (ap->keynode == keynode) {
990     prevap->next = ap->next;
991     KFREE(ap);
992     break;
993         }
994         prevap = ap;
995       }
996
997       if (keynode->secassoc->iv)
998         KFREE(keynode->secassoc->iv);
999       if (keynode->secassoc->key)
1000        KFREE(keynode->secassoc->key);
1001      KFREE(keynode->secassoc);
1002      if (keynode->solist)
1003        KFREE(keynode->solist);
1004      KFREE(keynode);
1005      CRITICAL_END;
1006      return(0);
```

```
1007   }
1008   return(-1);
1009 }
1010
1011
1012 /*------------------------------------------------------------------------
1013  * key_flush():
1014  *      Delete all entries from the key table.
1015  ------------------------------------------------------------------------*/
1016 void key_flush __P((void))
1017 {
1018   struct key_tblnode *keynode;
1019   int i;
1020
1021   /*
1022    * This is slow, but simple.
1023    */
1024   DPRINTF(IDL_FINISHED,("Flushing key table..."));
1025   for (i = 0; i < KEYTBLSIZE; i++) {
1026     while ((keynode = keytable[i].next))
1027       if (key_delete(keynode->secassoc) != 0)
1028   panic("key_flush");
1029   }
1030   DPRINTF(IDL_FINISHED,("done\n"));
1031 }
1032
1033
1034 /*------------------------------------------------------------------------
1035  * key_getspi():
1036  *      Get a unique spi value for a key management daemon/program.  The
1037  *      spi value, once assigned, cannot be assigned again (as long as the
1038  *      entry with that same spi value remains in the table).
1039  ------------------------------------------------------------------------*/
1040 int key_getspi DEFARGS((type, src, dst, lowval, highval, spi),
1041      u_int type AND
1042      SOCKADDR *src AND
1043      SOCKADDR *dst AND
1044      u_int32 lowval AND
1045      u_int32 highval AND
1046      u_int32 *spi)
1047 {
1048   struct key_secassoc *secassoc;
1049   struct key_tblnode *keynode, *prevkeynode;
1050   int count, done, len, indx;
1051   int maxcount = 1000;
1052   u_int32 val;
1053   char buf[MAXHASHKEYLEN];
1054   CRITICAL_DCL
1055
1056   DPRINTF(IDL_EVENT,("Entering getspi w/type=%d,low=%u,high=%u\n",
1057          type, lowval, highval));
1058   if (!(src && dst))
1059     return(EINVAL);
```

```
1060
1061    if ((lowval == 0) || (highval == 0))
1062      return(EINVAL);
1063
1064    if (lowval > highval) {
1065      u_int32 temp;
1066      temp = lowval;
1067      lowval = highval;
1068      highval = lowval;
1069    }
1070
1071    done = count = 0;
1072    do {
1073      count++;
1074      /*
1075       *  This may not be "random enough".
1076       */
1077      val = lowval + (random() % (highval - lowval + 1));
1078
1079      if (lowval == highval)
1080        count = maxcount;
1081      DPRINTF(IDL_FINISHED,("%u ",val));
1082      if (val) {
1083        DPRINTF(IDL_FINISHED,("\n"));
1084        bzero(&buf, sizeof(buf));
1085        len = key_createkey((char *)&buf, type, src, dst, val, 0);
1086        indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
1087        if (!key_search(type, src, dst, val, indx, &prevkeynode)) {
1088  CRITICAL_START;
1089  KMALLOC(secassoc, struct key_secassoc *, sizeof(struct key_secassoc));
1090  if (secassoc == 0) {
1091    DPRINTF(IDL_ERROR,("key_getspi: can't allocate memory\n"));
1092    CRITICAL_END;
1093    return(ENOBUFS);
1094  }
1095  bzero((char *)secassoc, sizeof(*secassoc));
1096
1097  DPRINTF(IDL_FINISHED,("getspi: indx=%d\n",indx));
1098  secassoc->len = sizeof(struct key_secassoc);
1099  secassoc->type = type;
1100  secassoc->spi = val;
1101  secassoc->state |= K_LARVAL;
1102  if (my_addr(secassoc->dst))
1103    secassoc->state |= K_INBOUND;
1104  if (my_addr(secassoc->src))
1105    secassoc->state |= K_OUTBOUND;
1106
1107  bcopy((char *)src, (char *)secassoc->src, src->sa_len);
1108  bcopy((char *)dst, (char *)secassoc->dst, dst->sa_len);
1109
1110  /* We fill this in with a plausable value now to insure
1111     that other routines don't break. These will get
1112     overwritten later with the correct values. */
```

```
1113 #ifdef INET6
1114   secassoc->from->sa_family = AF_INET6;
1115   secassoc->from->sa_len = sizeof(struct sockaddr_in6);
1116 #else /* INET6 */
1117   secassoc->from->sa_family = AF_INET;
1118   secassoc->from->sa_len = sizeof(struct sockaddr_in);
1119 #endif /* INET6 */
1120
1121   /*
1122    * We need to add code to age these larval key table
1123    * entries so they don't linger forever waiting for
1124    * a KEY_UPDATE message that may not come for various
1125    * reasons.  This is another task that key_reaper can
1126    * do once we have it coded.
1127    */
1128   secassoc->lifetime1 += TIME_SECONDS + maxlarvallifetime;
1129
1130   if (!(keynode = key_addnode(indx, secassoc))) {
1131     DPRINTF(IDL_ERROR,("key_getspi: can't add node\n"));
1132     CRITICAL_END;
1133     return(ENOBUFS);
1134   }
1135   DPRINTF(IDL_FINISHED,("key_getspi: added node 0x%x\n",
1136             (unsigned int)keynode));
1137   done++;
1138   CRITICAL_END;
1139       }
1140     }
1141   } while ((count < maxcount) && !done);
1142   DPRINTF(IDL_EVENT,("getspi returns w/spi=%u,count=%d\n",val,count));
1143   if (done) {
1144     *spi = val;
1145     return(0);
1146   } else {
1147     *spi = 0;
1148     return(EADDRNOTAVAIL);
1149   }
1150 }
1151
1152
1153 /*-------------------------------------------------------------------------
1154  * key_update():
1155  *      Update a keytable entry that has an spi value assigned but is
1156  *      incomplete (e.g. no key/iv).
1157  -------------------------------------------------------------------------*/
1158 int key_update DEFARGS((secassoc),
1159      struct key_secassoc *secassoc)
1160 {
1161   struct key_tblnode *keynode, *prevkeynode;
1162   struct key_allocnode *np = 0;
1163   u_int8 newstate;
1164   int len, indx, inbound, outbound;
1165   char buf[MAXHASHKEYLEN];
```

```
1166  CRITICAL_DCL
1167
1168  bzero(&buf, sizeof(buf));
1169  len = key_createkey((char *)&buf, secassoc->type, secassoc->src,
1170          secassoc->dst, secassoc->spi, 0);
1171  indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
1172  if(!(keynode = key_search(secassoc->type, secassoc->src, secassoc->dst,
1173          secassoc->spi, indx, &prevkeynode))) {
1174    return(ESRCH);
1175  }
1176  if (keynode->secassoc->state & K_DEAD)
1177    return(ESRCH);
1178
1179  /* Should we also restrict updating of only LARVAL entries ? */
1180
1181  CRITICAL_START;
1182
1183  inbound = my_addr(secassoc->dst);
1184  outbound = my_addr(secassoc->src);
1185
1186  newstate = keynode->secassoc->state;
1187  newstate &= ~K_LARVAL;
1188
1189  if (inbound)
1190    newstate |= K_INBOUND;
1191  if (outbound)
1192    newstate |= K_OUTBOUND;
1193
1194  if (outbound || (!inbound && !outbound)) {
1195    KMALLOC(np, struct key_allocnode *, sizeof(struct key_allocnode));
1196    if (np == 0) {
1197      DPRINTF(IDL_ERROR,("keyupdate: can't allocate allocnode!\n"));
1198      CRITICAL_END;
1199      return(ENOBUFS);
1200    }
1201  }
1202
1203  /*
1204   *  Free the old key and iv if they're there.
1205   */
1206  if (keynode->secassoc->key)
1207    KFREE(keynode->secassoc->key);
1208  if (keynode->secassoc->iv)
1209    KFREE(keynode->secassoc->iv);
1210
1211  /*
1212   *  We now copy the secassoc over. We don't need to copy
1213   *  the key and iv into new buffers since the calling routine
1214   *  does that already.
1215   */
1216
1217  *(keynode->secassoc) = *secassoc;
1218  keynode->secassoc->state = newstate;
```

```
1219
1220   /*
1221    * Should we allow a null key to be inserted into the table ?
1222    * or can we use null key to indicate some policy action...
1223    */
1224
1225 #if 0
1226   if (keynode->secassoc->key &&
1227        (keynode->secassoc->type == KEY_TYPE_ESP) &&
1228        ((keynode->secassoc->algorithm == IPSEC_ALGTYPE_ESP_DES_CBC) ||
1229   (keynode->secassoc->algorithm == IPSEC_ALGTYPE_ESP_3DES)))
1230       des_set_odd_parity(keynode->secassoc->key);
1231 #endif /* 0 */
1232
1233   /*
1234    *  We now add an entry to the allocation table for this
1235    *  updated key table entry.
1236    */
1237   if (outbound || (!inbound && !outbound)) {
1238     len = key_createkey((char *)&buf, secassoc->type, secassoc->src,
1239       secassoc->dst, 0, 1);
1240     indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
1241     DPRINTF(IDL_FINISHED,("keyupdate: keyalloc hash position=%d\n", indx));
1242     np->keynode = keynode;
1243     np->next = keyalloctbl[indx].next;
1244     keyalloctbl[indx].next = np;
1245   }
1246
1247   key_deleteacquire(secassoc->type, (SOCKADDR *)&(secassoc->dst));
1248
1249   CRITICAL_END;
1250   return(0);
1251 }
1252
1253 /*-------------------------------------------------------------------------
1254  * key_register():
1255  *      Register a socket as one capable of acquiring security associations
1256  *      for the kernel.
1257  -------------------------------------------------------------------------*/
1258 int key_register DEFARGS((socket, type),
1259      struct socket *socket AND
1260      u_int type)
1261 {
1262   struct key_registry *p, *new;
1263   CRITICAL_DCL
1264
1265   CRITICAL_START;
1266
1267   DPRINTF(IDL_EVENT,("Entering key_register w/so=0x%x,type=%d\n",
1268          (unsigned int)socket,type));
1269
1270   if (!(keyregtable && socket))
1271     panic("key_register");
```

```
1272
1273    /*
1274     * Make sure entry is not already in table
1275     */
1276    for(p = keyregtable->next; p; p = p->next) {
1277      if ((p->type == type) && (p->socket == socket)) {
1278        CRITICAL_END;
1279        return(EEXIST);
1280      }
1281    }
1282
1283    KMALLOC(new, struct key_registry *, sizeof(struct key_registry));
1284    if (new == 0) {
1285      CRITICAL_END;
1286      return(ENOBUFS);
1287    }
1288    new->type = type;
1289    new->socket = socket;
1290    new->next = keyregtable->next;
1291    keyregtable->next = new;
1292    CRITICAL_END;
1293    return(0);
1294  }
1295
1296  /*-----------------------------------------------------------------------
1297   * key_unregister():
1298   *      Delete entries from the registry list.
1299   *         allflag = 1 : delete all entries with matching socket
1300   *         allflag = 0 : delete only the entry matching socket and type
1301   -----------------------------------------------------------------------*/
1302  void key_unregister DEFARGS((socket, type, allflag),
1303       struct socket *socket AND
1304       u_int type AND
1305       int allflag)
1306  {
1307    struct key_registry *p, *prev;
1308    CRITICAL_DCL
1309
1310    CRITICAL_START;
1311
1312    DPRINTF(IDL_EVENT,("Entering key_unregister w/so=0x%x,type=%d,flag=%d\n",
1313           (unsigned int)socket, type, allflag));
1314
1315    if (!(keyregtable && socket))
1316      panic("key_register");
1317    prev = keyregtable;
1318    for(p = keyregtable->next; p; p = p->next) {
1319      if ((allflag && (p->socket == socket)) ||
1320    ((p->type == type) && (p->socket == socket))) {
1321        prev->next = p->next;
1322        KFREE(p);
1323        p = prev;
1324      }
```

```
1325      prev = p;
1326    }
1327    CRITICAL_END;
1328  }
1329
1330
1331  /*----------------------------------------------------------------------
1332   * key_acquire():
1333   *      Send a key_acquire message to all registered key mgnt daemons
1334   *      capable of acquire security association of type type.
1335   *
1336   *      Return: 0 if succesfully called key mgnt. daemon(s)
1337   *              -1 if not successfull.
1338   ----------------------------------------------------------------------*/
1339  int key_acquire DEFARGS((type, src, dst),
1340       u_int type AND
1341       SOCKADDR *src AND
1342       SOCKADDR *dst)
1343  {
1344    struct key_registry *p;
1345    struct key_acquirelist *ap, *prevap;
1346    int success = 0, created = 0;
1347    u_int etype;
1348    struct key_msghdr *km = NULL;
1349    int len;
1350
1351    DPRINTF(IDL_EVENT,("Entering key_acquire()\n"));
1352
1353    if (!keyregtable || !src || !dst)
1354      return (-1);
1355
1356    /*
1357     * We first check the acquirelist to see if a key_acquire
1358     * message has been sent for this destination.
1359     */
1360    etype = type;
1361    prevap = key_acquirelist;
1362    for(ap = key_acquirelist->next; ap; ap = ap->next) {
1363      if (addrpart_equal(dst, ap->target) &&
1364    (etype == ap->type)) {
1365        DPRINTF(IDL_EVENT,("acquire message previously sent!\n"));
1366        if (ap->expiretime < TIME_SECONDS) {
1367    DPRINTF(IDL_EVENT,("acquire message has expired!\n"));
1368    ap->count = 0;
1369    break;
1370        }
1371        if (ap->count < maxkeyacquire) {
1372    DPRINTF(IDL_EVENT,("max acquire messages not yet exceeded!\n"));
1373    break;
1374        }
1375        return(0);
1376      } else if (ap->expiretime < TIME_SECONDS) {
1377        /*
```

```
1378        *  Since we're already looking at the list, we may as
1379        *  well delete expired entries as we scan through the list.
1380        *  This should really be done by a function like key_reaper()
1381        *  but until we code key_reaper(), this is a quick and dirty
1382        *  hack.
1383        */
1384       DPRINTF(IDL_EVENT,("found an expired entry...deleting it!\n"));
1385       prevap->next = ap->next;
1386       KFREE(ap);
1387       ap = prevap;
1388     }
1389     prevap = ap;
1390   }
1391
1392   /*
1393    * Scan registry and send KEY_ACQUIRE message to
1394    * appropriate key management daemons.
1395    */
1396   for(p = keyregtable->next; p; p = p->next) {
1397     if (p->type != type)
1398       continue;
1399
1400     if (!created) {
1401       len = sizeof(struct key_msghdr) + ROUNDUP(src->sa_len) +
1402   ROUNDUP(dst->sa_len);
1403       KMALLOC(km, struct key_msghdr *, len);
1404       if (!km) {
1405   DPRINTF(IDL_ERROR,("key_acquire: no memory\n"));
1406   return(-1);
1407       }
1408       DPRINTF(IDL_FINISHED,("key_acquire/created: 1\n"));
1409       bzero((char *)km, len);
1410       km->key_msglen = len;
1411       km->key_msgvers = KEY_VERSION;
1412       km->key_msgtype = KEY_ACQUIRE;
1413       km->type = type;
1414       DPRINTF(IDL_FINISHED,("key_acquire/created: 2\n"));
1415       /*
1416        * This is inefficient and slow.
1417        */
1418
1419       /*
1420        * We zero out sin_zero here for AF_INET addresses because
1421        * ip_output() currently does not do it for performance reasons.
1422        */
1423       if (src->sa_family == AF_INET)
1424   bzero((char *)(((struct sockaddr_in *)src)->sin_zero),
1425         sizeof(((struct sockaddr_in *)src)->sin_zero));
1426       if (dst->sa_family == AF_INET)
1427   bzero((char *)(((struct sockaddr_in *)dst)->sin_zero),
1428         sizeof(((struct sockaddr_in *)dst)->sin_zero));
1429
1430       bcopy((char *)src, (char *)(km + 1), src->sa_len);
```

```
1431        bcopy((char *)dst, (char *)((int)(km + 1) + ROUNDUP(src->sa_len)),
1432        dst->sa_len);
1433        DPRINTF(IDL_FINISHED,("key_acquire/created: 3\n"));
1434        created++;
1435      }
1436      if (key_sendup(p->socket, km))
1437        success++;
1438    }
1439
1440    if (km)
1441      KFREE(km);
1442
1443    /*
1444     *  Update the acquirelist
1445     */
1446    if (success) {
1447      if (!ap) {
1448        DPRINTF(IDL_EVENT,("Adding new entry in acquirelist\n"));
1449        KMALLOC(ap, struct key_acquirelist *, sizeof(struct key_acquirelist));
1450        if (ap == 0)
1451    return(success ? 0 : -1);
1452        bzero((char *)ap, sizeof(struct key_acquirelist));
1453        bcopy((char *)dst, (char *)ap->target, dst->sa_len);
1454        ap->type = etype;
1455        ap->next = key_acquirelist->next;
1456        key_acquirelist->next = ap;
1457      }
1458      DPRINTF(IDL_GROSS_EVENT,("Updating acquire counter and expiration time\n"));
1459      ap->count++;
1460      ap->expiretime = TIME_SECONDS + maxacquiretime;
1461    }
1462    DPRINTF(IDL_EVENT,("key_acquire: done! success=%d\n",success));
1463    return(success ? 0 : -1);
1464  }
1465
1466  /*----------------------------------------------------------------------
1467   * key_alloc():
1468   *      Allocate a security association to a socket.  A socket requesting
1469   *      unique keying (per-socket keying) is assigned a security assocation
1470   *      exclusively for its use.  Sockets not requiring unique keying are
1471   *      assigned the first security association which may or may not be
1472   *      used by another socket.
1473   ----------------------------------------------------------------------*/
1474  int key_alloc DEFARGS((type, src, dst, socket, unique_key, keynodep),
1475       u_int type AND
1476       SOCKADDR *src AND
1477       SOCKADDR *dst AND
1478       struct socket *socket AND
1479       u_int  unique_key AND
1480       struct key_tblnode **keynodep)
1481  {
1482    struct key_tblnode *keynode;
1483    char buf[MAXHASHKEYLEN];
```

```
1484   struct key_allocnode *np, *prevnp;
1485   struct key_so2spinode *newnp;
1486   int len;
1487   int indx;
1488
1489   DPRINTF(IDL_FINISHED,("Entering key_alloc w/type=%u!\n",type));
1490   if (!(src && dst)) {
1491     DPRINTF(IDL_ERROR,("key_alloc: received null src or dst!\n"));
1492     return(-1);
1493   }
1494
1495   /*
1496    * Search key allocation table
1497    */
1498   bzero((char *)&buf, sizeof(buf));
1499   len = key_createkey((char *)&buf, type, src, dst, 0, 1);
1500   indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
1501
1502 #define np_type np->keynode->secassoc->type
1503 #define np_state np->keynode->secassoc->state
1504 #define np_src np->keynode->secassoc->src
1505 #define np_dst np->keynode->secassoc->dst
1506
1507   prevnp = &keyalloctbl[indx];
1508   for (np = keyalloctbl[indx].next; np; np = np->next) {
1509     if ((type == np_type) && addrpart_equal(src, np_src) &&
1510   addrpart_equal(dst, np_dst) &&
1511   !(np_state & (K_LARVAL | K_DEAD | K_UNIQUE))) {
1512       if (!(unique_key))
1513   break;
1514       if (!(np_state & K_USED))
1515   break;
1516     }
1517     prevnp = np;
1518   }
1519
1520   if (np) {
1521     struct socketlist *newsp;
1522     CRITICAL_DCL
1523
1524     CRITICAL_START;
1525
1526     DPRINTF(IDL_EVENT,("key_alloc: found node to allocate\n"));
1527     keynode = np->keynode;
1528
1529     KMALLOC(newnp, struct key_so2spinode *, sizeof(struct key_so2spinode));
1530     if (newnp == 0) {
1531       DPRINTF(IDL_ERROR,("key_alloc: Can't alloc mem for so2spi node!\n"));
1532       CRITICAL_END;
1533       return(ENOBUFS);
1534     }
1535     KMALLOC(newsp, struct socketlist *, sizeof(struct socketlist));
1536     if (newsp == 0) {
```

```
1537        DPRINTF(IDL_ERROR,("key_alloc: Can't alloc mem for socketlist!\n"));
1538        if (newnp)
1539    KFREE(newnp);
1540        CRITICAL_END;
1541        return(ENOBUFS);
1542      }
1543
1544      /*
1545       * Add a hash entry into the so2spi table to
1546       * map socket to allocated secassoc.
1547       */
1548      DPRINTF(IDL_FINISHED,("key_alloc: adding entry to so2spi table..."));
1549      newnp->keynode = keynode;
1550      newnp->socket = socket;
1551      newnp->next = so2spitbl[((u_int32)socket) % SO2SPITBLSIZE].next;
1552      so2spitbl[((u_int32)socket) % SO2SPITBLSIZE].next = newnp;
1553      DPRINTF(IDL_FINISHED,("done\n"));
1554
1555      if (unique_key) {
1556        /*
1557         * Need to remove the allocation entry
1558         * since the secassoc is now unique and
1559         * can't be allocated to any other socket
1560         */
1561        DPRINTF(IDL_EVENT,("key_alloc: making keynode unique..."));
1562        keynode->secassoc->state |= K_UNIQUE;
1563        prevnp->next = np->next;
1564        KFREE(np);
1565        DPRINTF(IDL_EVENT,("done\n"));
1566      }
1567      keynode->secassoc->state |= K_USED;
1568      keynode->secassoc->state |= K_OUTBOUND;
1569      keynode->alloc_count++;
1570
1571      /*
1572       * Add socket to list of socket using secassoc.
1573       */
1574      DPRINTF(IDL_FINISHED,("key_alloc: adding so to solist..."));
1575      newsp->socket = socket;
1576      newsp->next = keynode->solist->next;
1577      keynode->solist->next = newsp;
1578      DPRINTF(IDL_FINISHED,("done\n"));
1579      *keynodep = keynode;
1580      CRITICAL_END;
1581      return(0);
1582    }
1583    *keynodep = NULL;
1584    return(0);
1585  }
1586
1587
1588  /*-------------------------------------------------------------------
1589   * key_free():
```

```
1590  *      Decrement the refcount for a key table entry.  If the entry is
1591  *      marked dead, and the refcount is zero, we go ahead and delete it.
1592  ----------------------------------------------------------------------*/
1593 void key_free DEFARGS((keynode),
1594      struct key_tblnode *keynode)
1595 {
1596   DPRINTF(IDL_GROSS_EVENT,("Entering key_free w/keynode=0x%x\n",
1597          (unsigned int)keynode));
1598   if (!keynode) {
1599     DPRINTF(IDL_ERROR,("Warning: key_free got null pointer\n"));
1600     return;
1601   }
1602   (keynode->ref_count)--;
1603   if (keynode->ref_count < 0) {
1604     DPRINTF(IDL_ERROR,("Warning: key_free decremented refcount to
      %d\n",keynode->ref_count));
1605   }
1606   if ((keynode->secassoc->state & K_DEAD) && (keynode->ref_count <= 0)) {
1607     DPRINTF(IDL_GROSS_EVENT,("key_free: calling key_delete\n"));
1608     key_delete(keynode->secassoc);
1609   }
1610 }
1611
1612 /*----------------------------------------------------------------------
1613  * getassocbyspi():
1614  *      Get a security association for a given type, src, dst, and spi.
1615  *
1616  *      Returns: 0 if sucessfull
1617  *               -1 if error/not found
1618  *
1619  *      Caller must convert spi to host order.  Function assumes spi is
1620  *      in host order!
1621  ----------------------------------------------------------------------*/
1622 int getassocbyspi DEFARGS((type, src, dst, spi, keyentry),
1623      u_int type AND
1624      SOCKADDR *src AND
1625      SOCKADDR *dst AND
1626      u_int32 spi AND
1627      struct key_tblnode **keyentry)
1628 {
1629   char buf[MAXHASHKEYLEN];
1630   int len, indx;
1631   struct key_tblnode *keynode, *prevkeynode = 0;
1632
1633   DPRINTF(IDL_FINISHED,("Entering getassocbyspi w/type=%u spi=%u\n",type,spi));
1634
1635   *keyentry = NULL;
1636   bzero(&buf, sizeof(buf));
1637   len = key_createkey((char *)&buf, type, src, dst, spi, 0);
1638   indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
1639   DPRINTF(IDL_FINISHED,("getassocbyspi: indx=%d\n",indx));
1640   DDO(IDL_FINISHED,dump_sockaddr(src);dump_sockaddr(dst));
1641   keynode = key_search(type, src, dst, spi, indx, &prevkeynode);
```

```
1642   DPRINTF(IDL_FINISHED,("getassocbyspi: keysearch ret=0x%x\n",
1643       (unsigned int)keynode));
1644   if (keynode && !(keynode->secassoc->state & (K_DEAD | K_LARVAL))) {
1645     DPRINTF(IDL_GROSS_EVENT,("getassocbyspi: found secassoc!\n"));
1646     (keynode->ref_count)++;
1647     keynode->secassoc->state |= K_USED;
1648     *keyentry = keynode;
1649   } else {
1650     DPRINTF(IDL_EVENT,("getassocbyspi: secassoc not found!\n"));
1651     return (-1);
1652   }
1653   return(0);
1654 }
1655
1656
1657 /*-------------------------------------------------------------------------
1658  * getassocbysocket():
1659  *      Get a security association for a given type, src, dst, and socket.
1660  *      If not found, try to allocate one.
1661  *      Returns: 0 if successfull
1662  *               -1 if error condition/secassoc not found (*keyentry = NULL)
1663  *                1 if secassoc temporarily unavailable (*keynetry = NULL)
1664  *                  (e.g., key mgnt. daemon(s) called)
1665  -------------------------------------------------------------------------*/
1666 int getassocbysocket DEFARGS((type, src, dst, socket, unique_key, keyentry),
1667      u_int type AND
1668      SOCKADDR *src AND
1669      SOCKADDR *dst AND
1670      struct socket *socket AND
1671      u_int unique_key AND
1672      struct key_tblnode **keyentry)
1673 {
1674   struct key_tblnode *keynode = 0;
1675   struct key_so2spinode *np;
1676   u_int realtype;
1677
1678   DPRINTF(IDL_FINISHED,("Entering getassocbysocket w/type=%u so=0x%x\n",
1679       type,(unsigned int)socket));
1680
1681   /*
1682    *  We treat esp-transport mode and esp-tunnel mode
1683    *  as a single type in the keytable.  This has a side
1684    *  effect that socket using both esp-transport and
1685    *  esp-tunnel will use the same security association
1686    *  for both modes.  Is this a problem?
1687    */
1688   realtype = type;
1689   if ((np = key_sosearch(type, src, dst, socket))) {
1690     if (np->keynode && np->keynode->secassoc &&
1691   !(np->keynode->secassoc->state & (K_DEAD | K_LARVAL))) {
1692       DPRINTF(IDL_FINISHED,("getassocbysocket: found secassoc!\n"));
1693       (np->keynode->ref_count)++;
1694       *keyentry = np->keynode;
```

```
1695        return(0);
1696      }
1697    }
1698
1699    /*
1700     * No secassoc has been allocated to socket,
1701     * so allocate one, if available
1702     */
1703    DPRINTF(IDL_GROSS_EVENT,("getassocbyso: can't find it, trying to allocate!\n"));
1704    if (key_alloc(realtype, src, dst, socket, unique_key, &keynode) == 0) {
1705      if (keynode) {
1706        DPRINTF(IDL_GROSS_EVENT,("getassocbyso: key_alloc found secassoc!\n"));
1707        keynode->ref_count++;
1708        *keyentry = keynode;
1709        return(0);
1710      } else {
1711        /*
1712         * Kick key mgnt. daemon(s)
1713         * (this should be done in ipsec_output_policy() instead or
1714         * selectively called based on a flag value)
1715         */
1716        DPRINTF(IDL_FINISHED,("getassocbyso: calling key mgnt daemons!\n"));
1717        *keyentry = NULL;
1718        if (key_acquire(realtype, src, dst) == 0)
1719    return (1);
1720        else
1721    return(-1);
1722      }
1723    }
1724    *keyentry = NULL;
1725    return(-1);
1726  }
1727
1728  /*-----------------------------------------------------------------------
1729   * key_xdata():
1730   *      Parse message buffer for src/dst/from/iv/key if parseflag = 0
1731   *      else parse for src/dst only.
1732   -----------------------------------------------------------------------*/
1733  int key_xdata DEFARGS((km, kip, parseflag),
1734       struct key_msghdr *km AND
1735       struct key_msgdata *kip AND
1736       int parseflag)
1737  {
1738    char *cp, *cpmax;
1739
1740    if (!km || (km->key_msglen <= 0))
1741      return (-1);
1742
1743    cp = (caddr_t)(km + 1);
1744    cpmax = (caddr_t)km + km->key_msglen;
1745
1746    /*
1747     * Assumes user process passes message with
```

```
1748     * correct word alignment.
1749     */
1750
1751    /*
1752     * Need to clean up this code later.
1753     */
1754
1755    /* Grab src addr */
1756    kip->src = (SOCKADDR *)cp;
1757    if (!kip->src->sa_len) {
1758      DPRINTF(IDL_MAJOR_EVENT,("key_xdata couldn't parse src addr\n"));
1759      return(-1);
1760    }
1761
1762    ADVANCE(cp, kip->src->sa_len);
1763
1764    /* Grab dest addr */
1765    kip->dst = (SOCKADDR *)cp;
1766    if (!kip->dst->sa_len) {
1767      DPRINTF(IDL_MAJOR_EVENT,("key_xdata couldn't parse dest addr\n"));
1768      return(-1);
1769    }
1770
1771    ADVANCE(cp, kip->dst->sa_len);
1772    if (parseflag == 1) {
1773      kip->from = 0;
1774      kip->key = kip->iv = 0;
1775      kip->keylen = kip->ivlen = 0;
1776      return(0);
1777    }
1778
1779    /* Grab from addr */
1780    kip->from = (SOCKADDR *)cp;
1781    if (!kip->from->sa_len) {
1782      DPRINTF(IDL_MAJOR_EVENT,("key_xdata couldn't parse from addr\n"));
1783      return(-1);
1784    }
1785
1786    ADVANCE(cp, kip->from->sa_len);
1787
1788    /* Grab key */
1789    if ((kip->keylen = km->keylen)) {
1790      kip->key = cp;
1791      ADVANCE(cp, km->keylen);
1792    } else
1793      kip->key = 0;
1794
1795    /* Grab iv */
1796    if ((kip->ivlen = km->ivlen))
1797      kip->iv = cp;
1798    else
1799      kip->iv = 0;
1800
```

```
1801   return (0);
1802 }
1803
1804
1805 int key_parse DEFARGS((kmp, so, dstfamily),
1806 struct key_msghdr **kmp AND
1807 struct socket *so AND
1808 int *dstfamily)
1809 {
1810   int error = 0, keyerror = 0;
1811   struct key_msgdata keyinfo;
1812   struct key_secassoc *secassoc = NULL;
1813   struct key_msghdr *km = *kmp;
1814
1815   DPRINTF(IDL_MAJOR_EVENT, ("Entering key_parse\n"));
1816
1817 #define senderr(e) \
1818   { error = (e); goto flush; }
1819
1820   if (km->key_msgvers != KEY_VERSION) {
1821     DPRINTF(IDL_CRITICAL,("keyoutput: Unsupported key message version!\n"));
1822     senderr(EPROTONOSUPPORT);
1823   }
1824
1825   km->key_pid = CURRENT_PID;
1826
1827   DDO(IDL_MAJOR_EVENT, printf("keymsghdr:\n"); dump_keymsghdr(km));
1828
1829   /*
1830    * Parse buffer for src addr, dest addr, from addr, key, iv
1831    */
1832   bzero((char *)&keyinfo, sizeof(keyinfo));
1833
1834   switch (km->key_msgtype) {
1835   case KEY_ADD:
1836     DPRINTF(IDL_MAJOR_EVENT,("key_output got KEY_ADD msg\n"));
1837
1838     if (key_xdata(km, &keyinfo, 0) < 0)
1839       goto parsefail;
1840
1841     /*
1842      * Allocate the secassoc structure to insert
1843      * into key table here.
1844      */
1845     KMALLOC(secassoc, struct key_secassoc *, sizeof(struct key_secassoc));
1846     if (secassoc == 0) {
1847       DPRINTF(IDL_CRITICAL,("keyoutput: No more memory!\n"));
1848       senderr(ENOBUFS);
1849     }
1850
1851     if (key_msghdr2secassoc(secassoc, km, &keyinfo) < 0) {
1852       DPRINTF(IDL_CRITICAL,("keyoutput: key_msghdr2secassoc failed!\n"));
1853       KFREE(secassoc);
```

```
1854        senderr(EINVAL);
1855      }
1856      DPRINTF(IDL_MAJOR_EVENT,("secassoc to add:\n"));
1857      DDO(IDL_MAJOR_EVENT,dump_secassoc(secassoc));
1858
1859      if ((keyerror = key_add(secassoc)) != 0) {
1860        DPRINTF(IDL_CRITICAL,("keyoutput: key_add failed\n"));
1861        if (secassoc->key)
1862    KFREE(secassoc->key);
1863        if (secassoc->iv)
1864    KFREE(secassoc->iv);
1865        KFREE(secassoc);
1866        if (keyerror == -2) {
1867    senderr(EEXIST);
1868        } else {
1869    senderr(ENOBUFS);
1870        }
1871      }
1872      break;
1873    case KEY_DELETE:
1874      DPRINTF(IDL_MAJOR_EVENT,("key_output got KEY_DELETE msg\n"));
1875
1876      if (key_xdata(km, &keyinfo, 1) < 0)
1877        goto parsefail;
1878
1879      KMALLOC(secassoc, struct key_secassoc *, sizeof(struct key_secassoc));
1880      if (secassoc == 0) {
1881        senderr(ENOBUFS);
1882      }
1883      if (key_msghdr2secassoc(secassoc, km, &keyinfo) < 0) {
1884        KFREE(secassoc);
1885        senderr(EINVAL);
1886      }
1887      if (key_delete(secassoc) != 0) {
1888        if (secassoc->iv)
1889    KFREE(secassoc->iv);
1890        if (secassoc->key)
1891    KFREE(secassoc->key);
1892        KFREE(secassoc);
1893        senderr(ESRCH);
1894      }
1895      if (secassoc->iv)
1896        KFREE(secassoc->iv);
1897      if (secassoc->key)
1898        KFREE(secassoc->key);
1899      KFREE(secassoc);
1900      break;
1901    case KEY_UPDATE:
1902      DPRINTF(IDL_EVENT,("key_output got KEY_UPDATE msg\n"));
1903
1904      if (key_xdata(km, &keyinfo, 0) < 0)
1905        goto parsefail;
1906
```

```
1907      KMALLOC(secassoc, struct key_secassoc *, sizeof(struct key_secassoc));
1908      if (secassoc == 0) {
1909        senderr(ENOBUFS);
1910      }
1911      if (key_msghdr2secassoc(secassoc, km, &keyinfo) < 0) {
1912        KFREE(secassoc);
1913        senderr(EINVAL);
1914      }
1915      if ((keyerror = key_update(secassoc)) != 0) {
1916        DPRINTF(IDL_CRITICAL,("Error updating key entry\n"));
1917        if (secassoc->iv)
1918    KFREE(secassoc->iv);
1919        if (secassoc->key)
1920    KFREE(secassoc->key);
1921        KFREE(secassoc);
1922        senderr(keyerror);
1923      }
1924      KFREE(secassoc);
1925      break;
1926    case KEY_GET:
1927      DPRINTF(IDL_EVENT,("key_output got KEY_GET msg\n"));
1928
1929      if (key_xdata(km, &keyinfo, 1) < 0)
1930        goto parsefail;
1931
1932      if (key_get(km->type, (SOCKADDR *)keyinfo.src,
1933      (SOCKADDR *)keyinfo.dst,
1934      km->spi, &secassoc) != 0) {
1935        DPRINTF(IDL_EVENT,("keyoutput: can't get key\n"));
1936        senderr(ESRCH);
1937      }
1938
1939      if (secassoc) {
1940        int newlen;
1941
1942        DPRINTF(IDL_EVENT,("keyoutput: Found secassoc!\n"));
1943        newlen = sizeof(struct key_msghdr) + ROUNDUP(secassoc->src->sa_len) +
1944    ROUNDUP(secassoc->dst->sa_len) + ROUNDUP(secassoc->from->sa_len) +
1945      ROUNDUP(secassoc->keylen) + ROUNDUP(secassoc->ivlen);
1946        DPRINTF(IDL_EVENT,("keyoutput: newlen=%d\n", newlen));
1947        if (newlen > km->key_msglen) {
1948    struct key_msghdr *newkm;
1949
1950    DPRINTF(IDL_EVENT,("keyoutput: Allocating new buffer!\n"));
1951    KMALLOC(newkm, struct key_msghdr *, newlen);
1952    if (newkm == 0) {
1953      senderr(ENOBUFS);
1954    }
1955    bcopy((char *)km, (char *)newkm, km->key_msglen);
1956    DPRINTF(IDL_FINISHED,("keyoutput: 1\n"));
1957    KFREE(km);
1958    *kmp = km = newkm;
1959    DPRINTF(IDL_CRITICAL, ("km->key_msglen = %d, newlen = %d\n",
```

```
1960                km->key_msglen, newlen));
1961    km->key_msglen = newlen;
1962        }
1963        DPRINTF(IDL_FINISHED,("keyoutput: 2\n"));
1964        if (key_secassoc2msghdr(secassoc, km, &keyinfo)) {
1965    DPRINTF(IDL_CRITICAL,("keyoutput: Can't create msghdr!\n"));
1966    senderr(EINVAL);
1967        }
1968        DPRINTF(IDL_FINISHED,("keyoutput: 3\n"));
1969      }
1970      break;
1971    case KEY_GETSPI:
1972      DPRINTF(IDL_EVENT,("key_output got KEY_GETSPI msg\n"));
1973
1974      if (key_xdata(km, &keyinfo, 1) < 0)
1975        goto parsefail;
1976
1977      if ((keyerror = key_getspi(km->type, keyinfo.src, keyinfo.dst,
1978                km->lifetime1, km->lifetime2,
1979                &(km->spi))) != 0) {
1980        DPRINTF(IDL_CRITICAL,("keyoutput: getspi failed error=%d\n", keyerror));
1981        senderr(keyerror);
1982      }
1983      break;
1984    case KEY_REGISTER:
1985      DPRINTF(IDL_EVENT,("key_output got KEY_REGISTER msg\n"));
1986      key_register(so, km->type);
1987      break;
1988    case KEY_DUMP:
1989      DPRINTF(IDL_EVENT,("key_output got KEY_DUMP msg\n"));
1990      error = key_dump(so);
1991      return(error);
1992      break;
1993    case KEY_FLUSH:
1994      DPRINTF(IDL_EVENT,("key_output got KEY_FLUSH msg\n"));
1995      key_flush();
1996      break;
1997    default:
1998      DPRINTF(IDL_CRITICAL,("key_output got unsupported msg type=%d\n",
1999              km->key_msgtype));
2000      senderr(EOPNOTSUPP);
2001    }
2002
2003    goto flush;
2004
2005  parsefail:
2006    keyinfo.dst = NULL;
2007    error = EINVAL;
2008
2009  flush:
2010    if (km)
2011      km->key_errno = error;
2012
```

```
2013   if (dstfamily)
2014     *dstfamily = keyinfo.dst ? keyinfo.dst->sa_family : 0;
2015
2016   DPRINTF(IDL_MAJOR_EVENT, ("key_parse exiting with error=%d\n", error));
2017   return error;
2018 }
```