****
1274 Tue May 20 20:20:10 2014
new/usr/src/cmd/cmd-inet/usr.lib/wanboot/Makefile.com
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
****

```
   1 #
   2 # CDDL HEADER START
   3 #
   4 # The contents of this file are subject to the terms of the
   5 # Common Development and Distribution License, Version 1.0 only
   6 # (the "License").  You may not use this file except in compliance
   7 # with the License.
   8 #
   9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10 # or http://www.opensolaris.org/os/licensing.
  11 # See the License for the specific language governing permissions
  12 # and limitations under the License.
  13 #
  14 # When distributing Covered Code, include this CDDL HEADER in each
  15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16 # If applicable, add the following below this CDDL HEADER, with the
  17 # fields enclosed by brackets "[]" replaced with your own identifying
  18 # information: Portions Copyright [yyyy] [name of copyright owner]
  19 #
  20 # CDDL HEADER END
  21 #
  22 #
  23 # Copyright 2003 Sun Microsystems, Inc.  All rights reserved.
  24 # Use is subject to license terms.
  25 #

  27 include $(SRC)/cmd/Makefile.cmd
  28 ROOTCMDDIR = $(ROOT)/usr/lib/inet/wanboot

  30 CMNCRYPTDIR = ../../../../../common/net/wanboot/crypt

  32 CERRWARN += -_gcc=-Wno-uninitialized

  34 # OpenSSL 1.0 and 0.9.8 produce different lint warnings
  35 LINTFLAGS   +=    -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED
  36 LINTFLAGS64 +=    -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED

  38 #endif /* ! codereview */
  39 .KEEP_STATE:
```

```
**********************************************************
   15554 Tue May 20 20:20:10 2014
new/usr/src/cmd/cmd-inet/usr.lib/wanboot/p12split/p12split.c
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
**********************************************************
______unchanged_portion_omitted_

 219 static int
 220 do_certs(void)
 221 {
 222         char *bufp;
 223         STACK_OF(X509) *ta_in = NULL;
 224         EVP_PKEY *pkey_in = NULL;
 225         X509 *xcert_in = NULL;

 227         sunw_crypto_init();

 229         if (read_files(&ta_in, &xcert_in, &pkey_in) < 0)
 230                 return (-1);

 232         if (verbose) {
 233                 if (xcert_in != NULL) {
 234                         (void) printf(gettext("\nMain cert:\n"));

 236                         /*
 237                          * sunw_subject_attrs() returns a pointer to
 238                          * memory allocated on our behalf. The same
 239                          * behavior is exhibited by sunw_issuer_attrs().
 240                          */
 241                         bufp = sunw_subject_attrs(xcert_in, NULL, 0);
 242                         if (bufp != NULL) {
 243                                 (void) printf(gettext("  Subject: %s\n"),
 244                                     bufp);
 245                                 OPENSSL_free(bufp);
 246                         }

 248                         bufp = sunw_issuer_attrs(xcert_in, NULL, 0);
 249                         if (bufp != NULL) {
 250                                 (void) printf(gettext("  Issuer: %s\n"), bufp);
 251                                 OPENSSL_free(bufp);
 252                         }

 254                         (void) sunw_print_times(stdout, PRNT_BOTH, NULL,
 255                             xcert_in);
 256                 }

 258                 if (ta_in != NULL) {
 259                         X509 *x;
 260                         int i;

 262                         for (i = 0; i < sk_X509_num(ta_in); i++) {
 263                                 /* LINTED */
 264 #endif /* ! codereview */
 265                                 x = sk_X509_value(ta_in, i);
 266                                 (void) printf(
 267                                     gettext("\nTrust Anchor cert %d:\n"), i);

 269                                 /*
 270                                  * sunw_subject_attrs() returns a pointer to
 271                                  * memory allocated on our behalf. We get the
 272                                  * same behavior from sunw_issuer_attrs().
 273                                  */
 274                                 bufp = sunw_subject_attrs(x, NULL, 0);
 275                                 if (bufp != NULL) {
 276                                         (void) printf(
 277                                             gettext("  Subject: %s\n"), bufp);
```



```
 278                                         OPENSSL_free(bufp);
 279                                 }

 281                                 bufp = sunw_issuer_attrs(x, NULL, 0);
 282                                 if (bufp != NULL) {
 283                                         (void) printf(
 284                                             gettext("  Issuer: %s\n"), bufp);
 285                                         OPENSSL_free(bufp);
 286                                 }

 288                                 (void) sunw_print_times(stdout, PRNT_BOTH,
 289                                     NULL, x);
 290                         }
 291                 }
 292         }

 294         check_certs(ta_in, &xcert_in);
 295         if (xcert_in != NULL && pkey_in != NULL) {
 296                 if (sunw_check_keys(xcert_in, pkey_in) == 0) {
 297                         wbku_printerr("warning: key and certificate do "
 298                             "not match\n");
 299                 }
 300         }

 302         return (write_files(ta_in, xcert_in, pkey_in));
 303 }

 305 static int
 306 read_files(STACK_OF(X509) **t_in, X509 **c_in, EVP_PKEY **k_in)
 307 {
 308         char *i_pass;

 310         i_pass = getpassphrase(gettext("Enter key password: "));

 312         if (get_ifile(input, i_pass, k_in, c_in, t_in) < 0)
 313                 return (-1);

 315         /*
 316          * If we are only interested in getting a trust anchor, and if there
 317          * is no trust anchor but is a regular cert, use it instead.  Do this
 318          * to handle the insanity with openssl, which requires a matching cert
 319          * and key in order to write a PKCS12 file.
 320          */
 321         if (outfiles == IO_TRUSTFILE) {
 322                 if (c_in != NULL && *c_in != NULL && t_in != NULL) {
 323                         if (*t_in == NULL) {
 324                                 if ((*t_in = sk_X509_new_null()) == NULL) {
 325                                         wbku_printerr("out of memory\n");
 326                                         return (-1);
 327                                 }
 328                         }

 330                         if (sk_X509_num(*t_in) == 0) {
 331                                 if (sk_X509_push(*t_in, *c_in) == 0) {
 332                                         wbku_printerr("out of memory\n");
 333                                         return (-1);
 334                                 }
 335                                 *c_in = NULL;
 336                         }
 337                 }
 338         }

 340         if ((outfiles & IO_KEYFILE) && *k_in == NULL) {
 341                 wbku_printerr("no matching key found\n");
 342                 return (-1);
 343         }
```

```
344         if ((outfiles & IO_CERTFILE) && *c_in == NULL) {
345                 wbku_printerr("no matching certificate found\n");
346                 return (-1);
347         }
348         if ((outfiles & IO_TRUSTFILE) && *t_in == NULL) {
349                 wbku_printerr("no matching trust anchor found\n");
350                 return (-1);
351         }

353         return (0);
354 }

356 static void
357 check_certs(STACK_OF(X509) *ta_in, X509 **c_in)
358 {
359         X509 *curr;
360         time_errs_t ret;
361         int i;
362         int del_expired = (outfiles != 0);

364         if (c_in != NULL && *c_in != NULL) {
365                 ret = time_check_print(*c_in);
366                 if ((ret != CHK_TIME_OK && ret != CHK_TIME_IS_BEFORE) &&
367                     del_expired) {
368                         (void) fprintf(stderr, gettext("  Removing cert\n"));
369                         X509_free(*c_in);
370                         *c_in = NULL;
371                 }
372         }

374         if (ta_in == NULL)
375                 return;

377         for (i = 0; i < sk_X509_num(ta_in); ) {
378                 /* LINTED */
379 #endif /* ! codereview */
380                 curr = sk_X509_value(ta_in, i);
381                 ret = time_check_print(curr);
382                 if ((ret != CHK_TIME_OK && ret != CHK_TIME_IS_BEFORE) &&
383                     del_expired) {
384                         (void) fprintf(stderr, gettext("  Removing cert\n"));
385                         /* LINTED */
386 #endif /* ! codereview */
387                         curr = sk_X509_delete(ta_in, i);
388                         X509_free(curr);
389                         continue;
390                 }
391                 i++;
392         }
393 }

395 static time_errs_t
396 time_check_print(X509 *cert)
397 {
398         char buf[256];
399         int ret;

401         ret = time_check(cert);
402         if (ret == CHK_TIME_OK)
403                 return (CHK_TIME_OK);

405         (void) fprintf(stderr, gettext("  Subject: %s"),
406             sunw_subject_attrs(cert, buf, sizeof (buf)));
407         (void) fprintf(stderr, gettext("  Issuer:  %s"),
408             sunw_issuer_attrs(cert, buf, sizeof (buf)));
```



```
410         switch (ret) {
411         case CHK_TIME_BEFORE_BAD:
412                 (void) fprintf(stderr,
413                     gettext("\n  Invalid cert 'not before' field\n"));
414                 break;

416         case CHK_TIME_AFTER_BAD:
417                 (void) fprintf(stderr,
418                     gettext("\n  Invalid cert 'not after' field\n"));
419                 break;

421         case CHK_TIME_HAS_EXPIRED:
422                 (void) sunw_print_times(stderr, PRNT_NOT_AFTER,
423                     gettext("\n  Cert has expired\n"), cert);
424                 break;

426         case CHK_TIME_IS_BEFORE:
427                 (void) sunw_print_times(stderr, PRNT_NOT_BEFORE,
428                     gettext("\n  Warning: cert not yet valid\n"), cert);
429                 break;

431         default:
432                 break;
433         }

435         return (ret);
436 }

438 static time_errs_t
439 time_check(X509 *cert)
440 {
441         int i;

443         i = X509_cmp_time(X509_get_notBefore(cert), NULL);
444         if (i == 0)
445                 return (CHK_TIME_BEFORE_BAD);
446         if (i > 0)
447                 return (CHK_TIME_IS_BEFORE);
448         /* After 'not before' time */

450         i = X509_cmp_time(X509_get_notAfter(cert), NULL);
451         if (i == 0)
452                 return (CHK_TIME_AFTER_BAD);
453         if (i < 0)
454                 return (CHK_TIME_HAS_EXPIRED);
455         return (CHK_TIME_OK);
456 }

458 static int
459 write_files(STACK_OF(X509) *t_out, X509 *c_out, EVP_PKEY *k_out)
460 {
461         if (key_out != NULL) {
462                 if (verbose)
463                         (void) printf(gettext("%s: writing key\n"), progname);
464                 if (do_ofile(key_out, k_out, NULL, NULL) < 0)
465                         return (-1);
466         }

468         if (cert_out != NULL) {
469                 if (verbose)
470                         (void) printf(gettext("%s: writing cert\n"), progname);
471                 if (do_ofile(cert_out, NULL, c_out, NULL) < 0)
472                         return (-1);
473         }

475         if (trust_out != NULL) {
```

```
476                 if (verbose)
477                         (void) printf(gettext("%s: writing trust\n"),
478                             progname);
479                 if (do_ofile(trust_out, NULL, NULL, t_out) < 0)
480                         return (-1);
481         }

483         return (0);
484 }

486 static int
487 get_ifile(char *name, char *pass, EVP_PKEY **tmp_k, X509 **tmp_c,
488     STACK_OF(X509) **tmp_t)
489 {
490         PKCS12          *p12;
491         FILE            *fp;
492         int             ret;
493         struct stat     sbuf;

495         if (stat(name, &sbuf) == 0 && !S_ISREG(sbuf.st_mode)) {
496                 wbku_printerr("%s is not a regular file\n", name);
497                 return (-1);
498         }

500         if ((fp = fopen(name, "r")) == NULL) {
501                 wbku_printerr("cannot open input file %s", name);
502                 return (-1);
503         }

505         p12 = d2i_PKCS12_fp(fp, NULL);
506         if (p12 == NULL) {
507                 wbku_printerr("cannot read file %s: %s\n", name, cryptoerr());
508                 (void) fclose(fp);
509                 return (-1);
510         }
511         (void) fclose(fp);

513         ret = sunw_PKCS12_parse(p12, pass, matchty, k_matchval, k_len,
514             NULL, tmp_k, tmp_c, tmp_t);
515         if (ret <= 0) {
516                 if (ret == 0)
517                         wbku_printerr("cannot find matching cert and key\n");
518                 else
519                         wbku_printerr("cannot parse %s: %s\n", name,
520                             cryptoerr());
521                 PKCS12_free(p12);
522                 return (-1);
523         }
524         return (0);
525 }

527 static int
528 do_ofile(char *name, EVP_PKEY *pkey, X509 *cert, STACK_OF(X509) *ta)
529 {
530         STACK_OF(EVP_PKEY) *klist = NULL;
531         STACK_OF(X509)  *clist = NULL;
532         PKCS12          *p12 = NULL;
533         int             ret = 0;
534         FILE            *fp;
535         struct stat     sbuf;

537         if (stat(name, &sbuf) == 0 && !S_ISREG(sbuf.st_mode)) {
538                 wbku_printerr("%s is not a regular file\n", name);
539                 return (-1);
540         }
```



```
542         if ((fp = fopen(name, "w")) == NULL) {
543                 wbku_printerr("cannot open output file %s", name);
544                 return (-1);
545         }

547         if ((clist = sk_X509_new_null()) == NULL ||
548             (klist = sk_EVP_PKEY_new_null()) == NULL) {
549                 wbku_printerr("out of memory\n");
550                 ret = -1;
551                 goto cleanup;
552         }

554         if (cert != NULL && sk_X509_push(clist, cert) == 0) {
555                 wbku_printerr("out of memory\n");
556                 ret = -1;
557                 goto cleanup;
558         }

560         if (pkey != NULL && sk_EVP_PKEY_push(klist, pkey) == 0) {
561                 wbku_printerr("out of memory\n");
562                 ret = -1;
563                 goto cleanup;
564         }

566         p12 = sunw_PKCS12_create(WANBOOT_PASSPHRASE, klist, clist, ta);
567         if (p12 == NULL) {
568                 wbku_printerr("cannot create %s: %s\n", name, cryptoerr());
569                 ret = -1;
570                 goto cleanup;
571         }

573         if (i2d_PKCS12_fp(fp, p12) == 0) {
574                 wbku_printerr("cannot write %s: %s\n", name, cryptoerr());
575                 ret = -1;
576                 goto cleanup;
577         }

579 cleanup:
580         (void) fclose(fp);
581         if (p12 != NULL)
582                 PKCS12_free(p12);
583         /*
584          * Put the cert and pkey off of the stack so that they won't
585          * be freed two times.  (If they get left in the stack then
586          * they will be freed with the stack.)
587          */
588         if (clist != NULL) {
589                 if (cert != NULL && sk_X509_num(clist) == 1) {
590                         /* LINTED */
591 #endif /* ! codereview */
592                         (void) sk_X509_delete(clist, 0);
593                 }
594                 sk_X509_pop_free(clist, X509_free);
595         }
596         if (klist != NULL) {
597                 if (pkey != NULL && sk_EVP_PKEY_num(klist) == 1) {
598                         /* LINTED */
599 #endif /* ! codereview */
600                         (void) sk_EVP_PKEY_delete(klist, 0);
601                 }
602                 sk_EVP_PKEY_pop_free(klist, sunw_evp_pkey_free);
603         }

605         return (ret);
606 }
```

```
608 static void
609 usage(void)
610 {
611         (void) fprintf(stderr,
612             gettext("usage:\n"
613             "     %s -i <file> -c <file> -k <file> -t <file> [-l <keyid> -v]\n"
614             "\n"),
615             progname);
616         (void) fprintf(stderr,
617             gettext(" where:\n"
618             "  -i - input file to be split into component parts and put in\n"
619             "       files given by -c, -k and -t\n"
620             "  -c - output file for the client certificate\n"
621             "  -k - output file for the client private key\n"
622             "  -t - output file for the remaining certificates (assumed\n"
623             "       to be trust anchors)\n"
624             "\n Files are assumed to be pkcs12-format files.\n\n"
625             "  -v - verbose\n"
626             "  -l - value of 'localkeyid' attribute in client cert and\n"
627             "       private key to be selected from the input file.\n\n"));
628         exit(EXIT_FAILURE);
629 }

631 /*
632  * Return a pointer to a static buffer that contains a listing of crypto
633  * errors.  We presume that the user doesn't want more than 8KB of error
634  * messages :-)
635  */
636 static const char *
637 cryptoerr(void)
638 {
639         static char     errbuf[8192];
640         ulong_t         err;
641         const char      *pfile;
642         int             line;
643         unsigned int    nerr = 0;

645         errbuf[0] = '\0';
646         while ((err = ERR_get_error_line(&pfile, &line)) != 0) {
647                 if (++nerr > 1)
648                         (void) strlcat(errbuf, "\n\t", sizeof (errbuf));

650                 if (err == (ulong_t)-1) {
651                         (void) strlcat(errbuf, strerror(errno),
652                             sizeof (errbuf));
653                         break;
654                 }
655                 (void) strlcat(errbuf, ERR_reason_error_string(err),
656                     sizeof (errbuf));
657         }

659         return (errbuf);
660 }
```

```
**********************************************************
   45047 Tue May 20 20:20:10 2014
new/usr/src/cmd/cmd-inet/usr.lib/wanboot/wanboot-cgi/wanboot-cgi.c
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
**********************************************************
______unchanged_portion_omitted_

 773 /*
 774  * Add the certs found in the trustfile found in path (a trust store) to
 775  * the file found at bootfs_dir/truststore.  If necessary, create the
 776  * output file.
 777  */
 778 static int
 779 build_trustfile(const char *path, void *truststorepath)
 780 {
 781         int             ret = WBCGI_FTW_CBERR;
 782         STACK_OF(X509)  *i_anchors = NULL;
 783         STACK_OF(X509)  *o_anchors = NULL;
 784         char            message[WBCGI_MAXBUF];
 785         PKCS12          *p12 = NULL;
 786         FILE            *rfp = NULL;
 787         FILE            *wfp = NULL;
 788         struct stat     i_st;
 789         struct stat     o_st;
 790         X509            *x = NULL;
 791         int             errtype = 0;
 792         int             wfd = -1;
 793         int             chars;
 794         int             i;

 796         if (!WBCGI_FILE_EXISTS(path, i_st)) {
 797                 goto cleanup;
 798         }

 800         if (WBCGI_FILE_EXISTS((char *)truststorepath, o_st)) {
 801                 /*
 802                  * If we are inadvertantly writing to the input file.
 803                  * return success.
 804                  * XXX Pete: how can this happen, and why success?
 805                  */
 806                 if (i_st.st_ino == o_st.st_ino) {
 807                         ret = WBCGI_FTW_CBCONT;
 808                         goto cleanup;
 809                 }
 810                 if ((wfp = fopen((char *)truststorepath, "r+")) == NULL) {
 811                         goto cleanup;
 812                 }
 813                 /*
 814                  * Read what's already there, so that new information
 815                  * can be added.
 816                  */
 817                 if ((p12 = d2i_PKCS12_fp(wfp, NULL)) == NULL) {
 818                         errtype = 1;
 819                         goto cleanup;
 820                 }
 821                 i = sunw_PKCS12_parse(p12, WANBOOT_PASSPHRASE, DO_NONE, NULL,
 822                     0, NULL, NULL, NULL, &o_anchors);
 823                 if (i <= 0) {
 824                         errtype = 1;
 825                         goto cleanup;
 826                 }

 828                 PKCS12_free(p12);
 829                 p12 = NULL;
 830         } else {
 831                 if (errno != ENOENT) {
```



```
 832                         chars = snprintf(message, sizeof (message),
 833                             "(error accessing file %s, error %s)",
 834                             path, strerror(errno));
 835                         if (chars > 0 && chars < sizeof (message))
 836                                 print_status(500, message);
 837                         else
 838                                 print_status(500, NULL);
 839                         return (WBCGI_FTW_CBERR);
 840                 }

 842                 /*
 843                  * Note: We could copy the file to the new trustfile, but
 844                  * we can't verify the password that way.  Therefore, copy
 845                  * it by reading it.
 846                  */
 847                 if ((wfd = open((char *)truststorepath,
 848                     O_CREAT|O_EXCL|O_RDWR, 0700)) < 0) {
 849                         goto cleanup;
 850                 }
 851                 if ((wfp = fdopen(wfd, "w+")) == NULL) {
 852                         goto cleanup;
 853                 }
 854                 o_anchors = sk_X509_new_null();
 855                 if (o_anchors == NULL) {
 856                         goto cleanup;
 857                 }
 858         }

 860         if ((rfp = fopen(path, "r")) == NULL) {
 861                 goto cleanup;
 862         }
 863         if ((p12 = d2i_PKCS12_fp(rfp, NULL)) == NULL) {
 864                 errtype = 1;
 865                 goto cleanup;
 866         }
 867         i = sunw_PKCS12_parse(p12, WANBOOT_PASSPHRASE, DO_NONE, NULL, 0, NULL,
 868             NULL, NULL, &i_anchors);
 869         if (i <= 0) {
 870                 errtype = 1;
 871                 goto cleanup;
 872         }
 873         PKCS12_free(p12);
 874         p12 = NULL;

 876         /*
 877          * Merge the two stacks of pkcs12 certs.
 878          */
 879         for (i = 0; i < sk_X509_num(i_anchors); i++) {
 880                 /* LINTED */
 881 #endif /* ! codereview */
 882                 x = sk_X509_delete(i_anchors, i);
 883                 (void) sk_X509_push(o_anchors, x);
 884         }

 886         /*
 887          * Create the pkcs12 structure from the modified input stack and
 888          * then write out that structure.
 889          */
 890         p12 = sunw_PKCS12_create((const char *)WANBOOT_PASSPHRASE, NULL, NULL,
 891             o_anchors);
 892         if (p12 == NULL) {
 893                 goto cleanup;
 894         }
 895         rewind(wfp);
 896         if (i2d_PKCS12_fp(wfp, p12) == 0) {
 897                 goto cleanup;
```

```
898         }

900         ret = WBCGI_FTW_CBCONT;
901 cleanup:
902         if (ret == WBCGI_FTW_CBERR) {
903                 if (errtype == 1) {
904                         chars = snprintf(message, sizeof (message),
905                             "(internal PKCS12 error while copying %s to %s)",
906                             path, (char *)truststorepath);
907                 } else {
908                         chars = snprintf(message, sizeof (message),
909                             "(error copying %s to %s)",
910                             path, (char *)truststorepath);
911                 }
912                 if (chars > 0 && chars <= sizeof (message)) {
913                         print_status(500, message);
914                 } else {
915                         print_status(500, NULL);
916                 }
917         }
918         if (rfp != NULL) {
919                 (void) fclose(rfp);
920         }
921         if (wfp != NULL) {
922                 /* Will also close wfd */
923                 (void) fclose(wfp);
924         }
925         if (p12 != NULL) {
926                 PKCS12_free(p12);
927         }
928         if (i_anchors != NULL) {
929                 sk_X509_pop_free(i_anchors, X509_free);
930         }
931         if (o_anchors != NULL) {
932                 sk_X509_pop_free(o_anchors, X509_free);
933         }

935         return (ret);
936 }

938 static boolean_t
939 check_key_type(const char *keyfile, const char *keytype, int flag)
940 {
941         boolean_t       ret = B_FALSE;
942         FILE            *key_fp = NULL;
943         wbku_key_attr_t ka;

945         /*
946          * Map keytype into the ka structure
947          */
948         if (wbku_str_to_keyattr(keytype, &ka, flag) != WBKU_SUCCESS) {
949                 goto cleanup;
950         }

952         /*
953          * Open the key file for reading.
954          */
955         if ((key_fp = fopen(keyfile, "r")) == NULL) {
956                 goto cleanup;
957         }

959         /*
960          * Find the valid client key, if it exists.
961          */
962         if (wbku_find_key(key_fp, NULL, &ka, NULL, B_FALSE) != WBKU_SUCCESS) {
963                 goto cleanup;
```



```
964         }

966         ret = B_TRUE;
967 cleanup:
968         if (key_fp != NULL) {
969                 (void) fclose(key_fp);
970         }

972         return (ret);
973 }

975 static boolean_t
976 resolve_hostname(const char *hostname, nvlist_t *nvl, boolean_t may_be_crap)
977 {
978         struct sockaddr_in      sin;
979         struct hostent          *hp;
980         struct utsname          un;
981         static char             myname[SYS_NMLN] = { '\0' };
982         char                    *cp = NULL;
983         char                    msg[WBCGI_MAXBUF];

985         /*
986          *  Initialize cached nodename
987          */
988         if (strlen(myname) == 0) {
989                 if (uname(&un) == -1) {
990                         (void) snprintf(msg, sizeof (msg),
991                             "(unable to retrieve uname, errno %d)", errno);
992                         print_status(500, msg);
993                         return (B_FALSE);
994                 }
995                 (void) strcpy(myname, un.nodename);
996         }

998         /*
999          * If hostname is local node name, return the address this
1000         * request came in on, which is supplied as SERVER_ADDR in the
1001         * cgi environment.  This ensures we don't send back a possible
1002         * alternate address that may be unreachable from the client's
1003         * network.  Otherwise, just resolve with nameservice.
1004         */
1005        if ((strcmp(hostname, myname) != 0) ||
1006            ((cp = getenv("SERVER_ADDR")) == NULL)) {
1007                if (((hp = gethostbyname(hostname)) == NULL) ||
1008                    (hp->h_addrtype != AF_INET) ||
1009                    (hp->h_length != sizeof (struct in_addr))) {
1010                        if (!may_be_crap) {
1011                                print_status(500, "(error resolving hostname)");
1012                        }
1013                        return (may_be_crap);
1014                }
1015                (void) memcpy(&sin.sin_addr, hp->h_addr, hp->h_length);
1016                cp = inet_ntoa(sin.sin_addr);
1017        }

1019        if (nvlist_add_string(nvl, (char *)hostname, cp) != 0) {
1020                print_status(500, "(error adding hostname to nvlist)");
1021                return (B_FALSE);
1022        }

1024        return (B_TRUE);
1025 }

1027 /*
1028  * one_name() is called for each certificate found and is passed the string
1029  * that X509_NAME_oneline() returns.  Its job is to find the common name and
```

```
 * determine whether it is a host name; if it is then a line suitable for
 * inclusion in /etc/inet/hosts is written to that file.
 */
static boolean_t
one_name(const char *namestr, nvlist_t *nvl)
{
        boolean_t       ret = B_TRUE;
        char            *p;
        char            *q;
        char            c;

        if (namestr != NULL &&
            (p = strstr(namestr, WBCGI_CNSTR)) != NULL) {
                p += WBCGI_CNSTR_LEN;

                if ((q = strpbrk(p, WBCGI_NAMESEP)) != NULL) {
                        c = *q;
                        *q = '\0';
                        ret = resolve_hostname(p, nvl, B_TRUE);
                        *q = c;
                } else {
                        ret = resolve_hostname(p, nvl, B_TRUE);
                }
        }

        return (ret);
}

/*
 * Loop through the certificates in a file
 */
static int
get_hostnames(const char *path, void *nvl)
{
        int             ret = WBCGI_FTW_CBERR;
        STACK_OF(X509)  *certs = NULL;
        PKCS12          *p12 = NULL;
        char            message[WBCGI_MAXBUF];
        char            buf[WBCGI_MAXBUF + 1];
        FILE            *rfp = NULL;
        X509            *x = NULL;
        int             errtype = 0;
        int             chars;
        int             i;

        if ((rfp = fopen(path, "r")) == NULL) {
                goto cleanup;
        }

        if ((p12 = d2i_PKCS12_fp(rfp, NULL)) == NULL) {
                errtype = 1;
                goto cleanup;
        }
        i = sunw_PKCS12_parse(p12, WANBOOT_PASSPHRASE, DO_NONE, NULL, 0, NULL,
            NULL, NULL, &certs);
        if (i <= 0) {
                errtype = 1;
                goto cleanup;
        }

        PKCS12_free(p12);
        p12 = NULL;

        for (i = 0; i < sk_X509_num(certs); i++) {
                /* LINTED */
#endif /* ! codereview */
```



```
                x = sk_X509_value(certs, i);
                if (!one_name(sunw_issuer_attrs(x, buf, sizeof (buf) - 1),
                    nvl)) {
                        goto cleanup;
                }
        }

        ret = WBCGI_FTW_CBCONT;
cleanup:
        if (ret == WBCGI_FTW_CBERR) {
                if (errtype == 1) {
                        chars = snprintf(message, sizeof (message),
                            "(internal PKCS12 error reading %s)", path);
                } else {
                        chars = snprintf(message, sizeof (message),
                            "error reading %s", path);
                }
                if (chars > 0 && chars <= sizeof (message)) {
                        print_status(500, message);
                } else {
                        print_status(500, NULL);
                }
        }
        if (rfp != NULL) {
                (void) fclose(rfp);
        }
        if (p12 != NULL) {
                PKCS12_free(p12);
        }
        if (certs != NULL) {
                sk_X509_pop_free(certs, X509_free);
        }

        return (ret);
}

/*
 * Create a hosts file by extracting hosts from client and truststore
 * files.  Use the CN. Then we should copy that file to the inet dir.
 */
static boolean_t
create_hostsfile(const char *hostsfile, const char *net, const char *cid)
{
        boolean_t       ret = B_FALSE;
        nvlist_t        *nvl;
        nvpair_t        *nvp;
        FILE            *hostfp = NULL;
        int             hostfd = -1;
        int             i;
        char            *hostslist;
        const char      *bc_urls[] = { BC_ROOT_SERVER, BC_BOOT_LOGGER, NULL };

        /*
         * Allocate nvlist handle to store our hostname/IP pairs.
         */
        if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
                print_status(500, "(error allocating hostname nvlist)");
                goto cleanup;
        }

        /*
         * Extract and resolve hostnames from CNs.
         */
        if (netboot_ftw(NB_CLIENT_CERT, net, cid,
            get_hostnames, nvl) == WBCGI_FTW_CBERR ||
            netboot_ftw(NB_CA_CERT, net, cid,
```

```
1162             get_hostnames, nvl) == WBCGI_FTW_CBERR) {
1163                 goto cleanup;
1164         }

1166         /*
1167          * Extract and resolve hostnames from any URLs in bootconf.
1168          */
1169         for (i = 0; bc_urls[i] != NULL; ++i) {
1170                 char    *urlstr;
1171                 url_t   url;

1173                 if ((urlstr = bootconf_get(&bc_handle, bc_urls[i])) != NULL &&
1174                     url_parse(urlstr, &url) == URL_PARSE_SUCCESS) {
1175                         if (!resolve_hostname(url.hport.hostname,
1176                             nvl, B_FALSE)) {
1177                                 goto cleanup;
1178                         }
1179                 }
1180         }

1182         /*
1183          * If there is a resolve-hosts list in bootconf, resolve those
1184          * hostnames too.
1185          */
1186         if ((hostslist = bootconf_get(&bc_handle, BC_RESOLVE_HOSTS)) != NULL) {
1187                 char    *hostname;

1189                 for (hostname = strtok(hostslist, ","); hostname != NULL;
1190                     hostname = strtok(NULL, ",")) {
1191                         if (!resolve_hostname(hostname, nvl, B_FALSE)) {
1192                                 goto cleanup;
1193                         }
1194                 }
1195         }

1197         /*
1198          * Now write the hostname/IP pairs gathered to the hosts file.
1199          */
1200         if ((hostfd = open(hostsfile,
1201             O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR)) == -1 ||
1202             (hostfp = fdopen(hostfd, "w+")) == NULL) {
1203                 print_status(500, "(error creating hosts file)");
1204                 goto cleanup;
1205         }
1206         for (nvp = nvlist_next_nvpair(nvl, NULL); nvp != NULL;
1207             nvp = nvlist_next_nvpair(nvl, nvp)) {
1208                 char    *hostname;
1209                 char    *ipstr;

1211                 hostname = nvpair_name(nvp);
1212                 if (nvpair_value_string(nvp, &ipstr) != 0) {
1213                         print_status(500, "(nvl error writing hosts file)");
1214                         goto cleanup;
1215                 }

1217                 if (fprintf(hostfp, "%s\t%s\n", ipstr, hostname) < 0) {
1218                         print_status(500, "(error writing hosts file)");
1219                         goto cleanup;
1220                 }
1221         }

1223         ret = B_TRUE;
1224 cleanup:
1225         if (nvl != NULL) {
1226                 nvlist_free(nvl);
1227         }
```



```
1228         if (hostfp != NULL) {
1229                 /*
1230                  * hostfd is automatically closed as well.
1231                  */
1232                 (void) fclose(hostfp);
1233         }

1235         return (ret);
1236 }

1238 static boolean_t
1239 bootfile_payload(const char *docroot, char **bootpathp)
1240 {
1241         boolean_t       ret = B_FALSE;
1242         char            *boot_file;
1243         struct stat     sbuf;

1245         if ((boot_file = bootconf_get(&bc_handle, BC_BOOT_FILE)) == NULL) {
1246                 print_status(500, "(boot_file must be specified)");
1247                 goto cleanup;
1248         }
1249         if ((*bootpathp = make_path(docroot, boot_file)) == NULL) {
1250                 goto cleanup;
1251         }
1252         if (!WBCGI_FILE_EXISTS(*bootpathp, sbuf)) {
1253                 print_status(500, "(boot_file missing)");
1254                 goto cleanup;
1255         }

1257         ret = B_TRUE;
1258 cleanup:
1259         return (ret);
1260 }

1262 /*
1263  * Create the wanboot file system whose contents are determined by the
1264  * security configuration specified in bootconf.
1265  */
1266 static boolean_t
1267 wanbootfs_payload(const char *net, const char *cid, const char *nonce,
1268     const char *bootconf, char **wanbootfs_imagep)
1269 {
1270         int             ret = B_FALSE;

1272         char            *server_authentication;
1273         char            *client_authentication;
1274         char            *scf;

1276         char            *bootfs_dir = NULL;
1277         char            *bootfs_etc_dir = NULL;
1278         char            *bootfs_etc_inet_dir = NULL;
1279         char            *bootfs_dev_dir = NULL;

1281         char            *systemconf = NULL;
1282         char            *keystorepath = NULL;
1283         char            *certstorepath = NULL;
1284         char            *truststorepath = NULL;
1285         char            *bootconfpath = NULL;
1286         char            *systemconfpath = NULL;
1287         char            *urandompath = NULL;
1288         char            *noncepath = NULL;
1289         char            *hostspath = NULL;
1290         char            *etc_hostspath = NULL;
1291         char            *timestamppath = NULL;

1293         boolean_t       authenticate_client;
```

```
1294        boolean_t       authenticate_server;

1296        struct stat     sbuf;

1298        /*
1299         * Initialize SSL stuff.
1300         */
1301        sunw_crypto_init();

1303        /*
1304         * Get the security strategy values.
1305         */
1306        client_authentication = bootconf_get(&bc_handle,
1307            BC_CLIENT_AUTHENTICATION);
1308        authenticate_client = (client_authentication != NULL &&
1309            strcmp(client_authentication, "yes") == 0);
1310        server_authentication = bootconf_get(&bc_handle,
1311            BC_SERVER_AUTHENTICATION);
1312        authenticate_server = (server_authentication != NULL &&
1313            strcmp(server_authentication, "yes") == 0);

1315        /*
1316         * Make a temporary directory structure for the wanboot file system.
1317         */
1318        if ((bootfs_dir = gen_tmppath("bootfs_dir", net, cid)) == NULL ||
1319            (bootfs_etc_dir = make_path(bootfs_dir, "etc")) == NULL ||
1320            (bootfs_etc_inet_dir = make_path(bootfs_etc_dir, "inet")) == NULL ||
1321            (bootfs_dev_dir = make_path(bootfs_dir, "dev")) == NULL) {
1322                goto cleanup;
1323        }
1324        if (mkdirp(bootfs_dir, 0700) ||
1325            mkdirp(bootfs_etc_dir, 0700) ||
1326            mkdirp(bootfs_etc_inet_dir, 0700) ||
1327            mkdirp(bootfs_dev_dir, 0700)) {
1328                print_status(500, "(error creating wanbootfs dir structure)");
1329                goto cleanup;
1330        }

1332        if (authenticate_client) {
1333                /*
1334                 * Add the client private key.
1335                 */
1336                if ((keystorepath = make_path(bootfs_dir,
1337                    NB_CLIENT_KEY)) == NULL ||
1338                    netboot_ftw(NB_CLIENT_KEY, net, cid,
1339                    create_keystore, keystorepath) != WBCGI_FTW_CBOK) {
1340                        goto cleanup;
1341                }

1343                /*
1344                 * Add the client certificate.
1345                 */
1346                if ((certstorepath = make_path(bootfs_dir,
1347                    NB_CLIENT_CERT)) == NULL ||
1348                    netboot_ftw(NB_CLIENT_CERT, net, cid,
1349                    copy_certstore, certstorepath) != WBCGI_FTW_CBOK) {
1350                        goto cleanup;
1351                }
1352        }

1354        if (authenticate_client || authenticate_server) {
1355                /*
1356                 * Add the trustfile; at least one truststore must exist.
1357                 */
1358                if ((truststorepath = make_path(bootfs_dir,
1359                    NB_CA_CERT)) == NULL) {
```



```
1360                        goto cleanup;
1361                }
1362                if (netboot_ftw(NB_CA_CERT, net, cid,
1363                    noact_cb, NULL) != WBCGI_FTW_CBOK) {
1364                        print_status(500, "(truststore not found)");
1365                }
1366                if (netboot_ftw(NB_CA_CERT, net, cid,
1367                    build_trustfile, truststorepath) == WBCGI_FTW_CBERR) {
1368                        goto cleanup;
1369                }

1371                /*
1372                 * Create the /dev/urandom file.
1373                 */
1374                if ((urandompath = make_path(bootfs_dev_dir,
1375                    "urandom")) == NULL ||
1376                    !create_urandom(urandompath)) {
1377                        goto cleanup;
1378                }
1379        }

1381        /*
1382         * Add the wanboot.conf(4) file.
1383         */
1384        if ((bootconfpath = make_path(bootfs_dir, NB_WANBOOT_CONF)) == NULL ||
1385            !copy_file(bootconf, bootconfpath)) {
1386                goto cleanup;
1387        }

1389        /*
1390         * Add the system_conf file if present.
1391         */
1392        if ((scf = bootconf_get(&bc_handle, BC_SYSTEM_CONF)) != NULL) {
1393                if (netboot_ftw(scf, net, cid,
1394                    set_pathname, &systemconf) != WBCGI_FTW_CBOK) {
1395                        print_status(500, "(system_conf file not found)");
1396                        goto cleanup;
1397                }
1398                if ((systemconfpath = make_path(bootfs_dir,
1399                    NB_SYSTEM_CONF)) == NULL ||
1400                    !copy_file(systemconf, systemconfpath)) {
1401                        goto cleanup;
1402                }
1403        }

1405        /*
1406         * Create the /nonce file.
1407         */
1408        if ((noncepath = make_path(bootfs_dir, "nonce")) == NULL ||
1409            !create_nonce(noncepath, nonce)) {
1410                goto cleanup;
1411        }

1413        /*
1414         * Create an /etc/inet/hosts file by extracting hostnames from CN,
1415         * URLs in bootconf and resolve-hosts in bootconf.
1416         */
1417        if ((hostspath = make_path(bootfs_etc_inet_dir, "hosts")) == NULL ||
1418            !create_hostsfile(hostspath, net, cid)) {
1419                goto cleanup;
1420        }

1422        /*
1423         * We would like to create a symbolic link etc/hosts -> etc/inet/hosts,
1424         * but unfortunately the HSFS support in the standalone doesn't handle
1425         * symlinks.
```

```
1426          */
1427         if ((etc_hostspath = make_path(bootfs_etc_dir, "hosts")) == NULL ||
1428             !copy_file(hostspath, etc_hostspath)) {
1429                 goto cleanup;
1430         }
1431 
1432         /*
1433          * Create the /timestamp file.
1434          */
1435         if ((timestamppath = make_path(bootfs_dir, "timestamp")) == NULL ||
1436             !create_timestamp(timestamppath, "timestamp")) {
1437                 goto cleanup;
1438         }
1439 
1440         /*
1441          * Create an HSFS file system for the directory.
1442          */
1443         if ((*wanbootfs_imagep = gen_tmppath("wanbootfs", net, cid)) == NULL ||
1444             !mkisofs(bootfs_dir, *wanbootfs_imagep)) {
1445                 goto cleanup;
1446         }
1447 
1448         ret = B_TRUE;
1449 cleanup:
1450         /*
1451          * Clean up temporary files and directories.
1452          */
1453         if (keystorepath != NULL &&
1454             WBCGI_FILE_EXISTS(keystorepath, sbuf)) {
1455                 (void) unlink(keystorepath);
1456         }
1457         if (certstorepath != NULL &&
1458             WBCGI_FILE_EXISTS(certstorepath, sbuf)) {
1459                 (void) unlink(certstorepath);
1460         }
1461         if (truststorepath != NULL &&
1462             WBCGI_FILE_EXISTS(truststorepath, sbuf)) {
1463                 (void) unlink(truststorepath);
1464         }
1465         if (bootconfpath != NULL &&
1466             WBCGI_FILE_EXISTS(bootconfpath, sbuf)) {
1467                 (void) unlink(bootconfpath);
1468         }
1469         if (systemconfpath != NULL &&
1470             WBCGI_FILE_EXISTS(systemconfpath, sbuf)) {
1471                 (void) unlink(systemconfpath);
1472         }
1473         if (urandompath != NULL &&
1474             WBCGI_FILE_EXISTS(urandompath, sbuf)) {
1475                 (void) unlink(urandompath);
1476         }
1477         if (noncepath != NULL &&
1478             WBCGI_FILE_EXISTS(noncepath, sbuf)) {
1479                 (void) unlink(noncepath);
1480         }
1481         if (hostspath != NULL &&
1482             WBCGI_FILE_EXISTS(hostspath, sbuf)) {
1483                 (void) unlink(hostspath);
1484         }
1485         if (etc_hostspath != NULL &&
1486             WBCGI_FILE_EXISTS(etc_hostspath, sbuf)) {
1487                 (void) unlink(etc_hostspath);
1488         }
1489         if (timestamppath != NULL &&
1490             WBCGI_FILE_EXISTS(timestamppath, sbuf)) {
1491                 (void) unlink(timestamppath);
```



```
1492         }
1493 
1494         if (bootfs_etc_inet_dir != NULL &&
1495             WBCGI_DIR_EXISTS(bootfs_etc_inet_dir, sbuf)) {
1496                 (void) rmdir(bootfs_etc_inet_dir);
1497         }
1498         if (bootfs_etc_dir != NULL &&
1499             WBCGI_DIR_EXISTS(bootfs_etc_dir, sbuf)) {
1500                 (void) rmdir(bootfs_etc_dir);
1501         }
1502         if (bootfs_dev_dir != NULL &&
1503             WBCGI_DIR_EXISTS(bootfs_dev_dir, sbuf)) {
1504                 (void) rmdir(bootfs_dev_dir);
1505         }
1506         if (bootfs_dir != NULL &&
1507             WBCGI_DIR_EXISTS(bootfs_dir, sbuf)) {
1508                 (void) rmdir(bootfs_dir);
1509         }
1510 
1511         /*
1512          * Free allocated memory.
1513          */
1514         free_path(&bootfs_dir);
1515         free_path(&bootfs_etc_dir);
1516         free_path(&bootfs_etc_inet_dir);
1517         free_path(&bootfs_dev_dir);
1518 
1519         free_path(&systemconf);
1520         free_path(&keystorepath);
1521         free_path(&certstorepath);
1522         free_path(&truststorepath);
1523         free_path(&bootconfpath);
1524         free_path(&systemconfpath);
1525         free_path(&urandompath);
1526         free_path(&noncepath);
1527         free_path(&hostspath);
1528         free_path(&etc_hostspath);
1529         free_path(&timestamppath);
1530 
1531         return (ret);
1532 }
1533 
1534 static boolean_t
1535 miniroot_payload(const char *net, const char *cid, const char *docroot,
1536     char **rootpathp, char **rootinfop, boolean_t *https_rootserverp)
1537 {
1538         boolean_t       ret = B_FALSE;
1539         char            *root_server;
1540         char            *root_file;
1541         url_t           url;
1542         struct stat     sbuf;
1543         char            sizebuf[WBCGI_MAXBUF];
1544         int             chars;
1545         int             fd = -1;
1546 
1547         if ((root_server = bootconf_get(&bc_handle, BC_ROOT_SERVER)) == NULL) {
1548                 print_status(500, "(root_server must be specified)");
1549                 goto cleanup;
1550         }
1551         if (url_parse(root_server, &url) != URL_PARSE_SUCCESS) {
1552                 print_status(500, "(root_server URL is invalid)");
1553         }
1554         *https_rootserverp = url.https;
1555 
1556         if ((root_file = bootconf_get(&bc_handle, BC_ROOT_FILE)) == NULL) {
1557                 print_status(500, "(rootfile must be specified)");
```

```
1558                goto cleanup;
1559        }
1560        if ((*rootpathp = make_path(docroot, root_file)) == NULL) {
1561                goto cleanup;
1562        }
1563        if (!WBCGI_FILE_EXISTS(*rootpathp, sbuf)) {
1564                print_status(500, "(root filesystem image missing)");
1565                goto cleanup;
1566        }

1568        if ((*rootinfop = gen_tmppath("mrinfo", net, cid)) == NULL) {
1569                goto cleanup;
1570        }
1571        if ((chars = snprintf(sizebuf, sizeof (sizebuf), "%ld",
1572            sbuf.st_size)) < 0 || chars > sizeof (sizebuf) ||
1573            (fd = open(*rootinfop,
1574            O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR)) == -1 ||
1575            !write_buffer(fd, sizebuf, strlen(sizebuf))) {
1576                print_status(500, "(error creating miniroot info file)");
1577                goto cleanup;
1578        }

1580        ret = B_TRUE;
1581 cleanup:
1582        if (fd != -1) {
1583                (void) close(fd);
1584        }

1586        return (ret);
1587 }

1589 static boolean_t
1590 deliver_payload(const char *payload, const char *payload_hash)
1591 {
1592        int             fd = fileno(stdout);
1593        struct stat     payload_buf, hash_buf;
1594        int             chars;
1595        char            main_header[WBCGI_MAXBUF];
1596        char            multi_header[WBCGI_MAXBUF];
1597        char            multi_header1[WBCGI_MAXBUF];
1598        char            multi_header2[WBCGI_MAXBUF];
1599        char            multi_end[WBCGI_MAXBUF];
1600        size_t          msglen;

1602        if (!WBCGI_FILE_EXISTS(payload, payload_buf) ||
1603            !WBCGI_FILE_EXISTS(payload_hash, hash_buf)) {
1604                print_status(500, "(payload/hash file(s) missing)");
1605                return (B_FALSE);
1606        }

1608        /*
1609         * Multi-part header.
1610         */
1611        if ((chars = snprintf(multi_header, sizeof (multi_header),
1612            "%s--%s%s%sapplication/octet-stream%s%s", WBCGI_CRNL,
1613            WBCGI_WANBOOT_BNDTXT, WBCGI_CRNL, WBCGI_CONTENT_TYPE, WBCGI_CRNL,
1614            WBCGI_CONTENT_LENGTH)) < 0 || chars > sizeof (multi_header)) {
1615                print_status(500, "(error creating multi_header)");
1616                return (B_FALSE);
1617        }

1619        /*
1620         * Multi-part header for part one.
1621         */
1622        if ((chars = snprintf(multi_header1, sizeof (multi_header1),
1623            "%s%ld%s%s", multi_header, payload_buf.st_size, WBCGI_CRNL,
```



```
1624            WBCGI_CRNL)) < 0 || chars > sizeof (multi_header1)) {
1625                print_status(500, "(error creating multi_header1)");
1626                return (B_FALSE);
1627        }

1629        /*
1630         * Multi-part header for part two.
1631         */
1632        if ((chars = snprintf(multi_header2, sizeof (multi_header2),
1633            "%s%ld%s%s", multi_header, hash_buf.st_size, WBCGI_CRNL,
1634            WBCGI_CRNL)) < 0 || chars > sizeof (multi_header2)) {
1635                print_status(500, "(error creating multi_header2)");
1636                return (B_FALSE);
1637        }

1639        /*
1640         * End-of-parts Trailer.
1641         */
1642        if ((chars = snprintf(multi_end, sizeof (multi_end),
1643            "%s--%s--%s", WBCGI_CRNL, WBCGI_WANBOOT_BNDTXT,
1644            WBCGI_CRNL)) < 0 || chars > sizeof (multi_end)) {
1645                print_status(500, "(error creating multi_end)");
1646                return (B_FALSE);
1647        }

1649        /*
1650         * Message header.
1651         */
1652        msglen = payload_buf.st_size +  hash_buf.st_size +
1653            strlen(multi_header1) + strlen(multi_header2) + strlen(multi_end);

1655        if ((chars = snprintf(main_header, sizeof (main_header),
1656            "%s%u%s%smultipart/mixed; boundary=%s%s%s", WBCGI_CONTENT_LENGTH,
1657            msglen, WBCGI_CRNL, WBCGI_CONTENT_TYPE, WBCGI_WANBOOT_BNDTXT,
1658            WBCGI_CRNL, WBCGI_CRNL)) < 0 || chars > sizeof (main_header)) {
1659                print_status(500, "(error creating main_header)");
1660                return (B_FALSE);
1661        }

1663        /*
1664         * Write the message out.  If things fall apart during this then
1665         * there's no way to report the error back to the client.
1666         */
1667        if (!write_buffer(fd, main_header, strlen(main_header)) ||
1668            !write_buffer(fd, multi_header1, strlen(multi_header1)) ||
1669            !write_file(fd, payload, payload_buf.st_size) ||
1670            !write_buffer(fd, multi_header2, strlen(multi_header2)) ||
1671            !write_file(fd, payload_hash, hash_buf.st_size) ||
1672            !write_buffer(fileno(stdout), multi_end, strlen(multi_end))) {
1673                return (B_FALSE);
1674        }

1676        return (B_TRUE);
1677 }


1680 /*ARGSUSED*/
1681 int
1682 main(int argc, char **argv)
1683 {
1684        int             ret = WBCGI_STATUS_ERR;
1685        struct stat     sbuf;
1686        int             content;
1687        char            *net;
1688        char            *cid;
1689        char            *nonce;
```

```
1690        char            *docroot;
1691        char            *payload;
1692        char            *signature_type;
1693        char            *encryption_type;
1694        char            *bootconf = NULL;
1695        char            *keyfile = NULL;
1696        char            *bootpath = NULL;
1697        char            *wanbootfs_image = NULL;
1698        char            *rootpath = NULL;
1699        char            *miniroot_info = NULL;
1700        char            *encr_payload = NULL;
1701        char            *payload_hash = NULL;
1702        boolean_t       https_rootserver;

1704        /*
1705         * Process the query string.
1706         */
1707        if (!get_request_info(&content, &net, &cid, &nonce, &docroot)) {
1708                goto cleanup;
1709        }

1711        /*
1712         * Sanity check that the netboot directory exists.
1713         */
1714        if (!WBCGI_DIR_EXISTS(NB_NETBOOT_ROOT, sbuf)) {
1715                print_status(500, "(" NB_NETBOOT_ROOT " does not exist)");
1716                goto cleanup;
1717        }

1719        /*
1720         * Get absolute bootconf pathname.
1721         */
1722        if (netboot_ftw(NB_WANBOOT_CONF, net, cid,
1723            set_pathname, &bootconf) != WBCGI_FTW_CBOK) {
1724                print_status(500, "(wanboot.conf not found)");
1725                goto cleanup;
1726        }

1728        /*
1729         * Initialize bc_handle from the given wanboot.conf file.
1730         */
1731        if (bootconf_init(&bc_handle, bootconf) != BC_SUCCESS) {
1732                char    message[WBCGI_MAXBUF];
1733                int     chars;

1735                chars = snprintf(message, sizeof (message),
1736                    "(wanboot.conf error: %s)", bootconf_errmsg(&bc_handle));
1737                if (chars > 0 && chars < sizeof (message))
1738                        print_status(500, message);
1739                else
1740                        print_status(500, "(wanboot.conf error)");
1741                goto cleanup;
1742        }

1744        /*
1745         * Get and check signature and encryption types,
1746         * presence of helper utilities, keystore, etc.
1747         */
1748        if ((signature_type = bootconf_get(&bc_handle,
1749            BC_SIGNATURE_TYPE)) != NULL) {
1750                if (!WBCGI_FILE_EXISTS(WBCGI_HMAC_PATH, sbuf)) {
1751                        print_status(500, "(hmac utility not found)");
1752                        goto cleanup;
1753                }
1754                if (keyfile == NULL && netboot_ftw(NB_CLIENT_KEY, net, cid,
1755                    set_pathname, &keyfile) != WBCGI_FTW_CBOK) {
```



```
1756                        print_status(500, "(keystore not found)");
1757                        goto cleanup;
1758                }
1759                if (!check_key_type(keyfile, signature_type, WBKU_HASH_KEY)) {
1760                        print_status(500, "(hash key not found)");
1761                        goto cleanup;
1762                }
1763        }
1764        if ((encryption_type = bootconf_get(&bc_handle,
1765            BC_ENCRYPTION_TYPE)) != NULL) {
1766                if (signature_type == NULL) {
1767                        print_status(500, "(encrypted but not signed)");
1768                        goto cleanup;
1769                }
1770                if (!WBCGI_FILE_EXISTS(WBCGI_ENCR_PATH, sbuf)) {
1771                        print_status(500, "(encr utility not found)");
1772                        goto cleanup;
1773                }
1774                if (keyfile == NULL && netboot_ftw(NB_CLIENT_KEY, net, cid,
1775                    set_pathname, &keyfile) != WBCGI_FTW_CBOK) {
1776                        print_status(500, "(keystore not found)");
1777                        goto cleanup;
1778                }
1779                if (!check_key_type(keyfile, encryption_type, WBKU_ENCR_KEY)) {
1780                        print_status(500, "(encr key not found)");
1781                        goto cleanup;
1782                }
1783        }

1785        /*
1786         * Determine/create our payload.
1787         */
1788        switch (content) {
1789        case WBCGI_CONTENT_BOOTFILE:
1790                if (!bootfile_payload(docroot, &bootpath)) {
1791                        goto cleanup;
1792                }
1793                payload = bootpath;

1795                break;

1797        case WBCGI_CONTENT_BOOTFS:
1798                if (!wanbootfs_payload(net, cid, nonce,
1799                    bootconf, &wanbootfs_image)) {
1800                        goto cleanup;
1801                }
1802                payload = wanbootfs_image;

1804                break;

1806        case WBCGI_CONTENT_ROOTFS:
1807                if (!miniroot_payload(net, cid, docroot,
1808                    &rootpath, &miniroot_info, &https_rootserver)) {
1809                        goto cleanup;
1810                }
1811                payload = rootpath;

1813                break;
1814        }

1816        /*
1817         * Encrypt the payload if necessary.
1818         */
1819        if (content != WBCGI_CONTENT_BOOTFILE &&
1820            content != WBCGI_CONTENT_ROOTFS &&
1821            encryption_type != NULL) {
```

```
1822                 if ((encr_payload = gen_tmppath("encr", net, cid)) == NULL) {
1823                         goto cleanup;
1824                 }

1826                 if (!encrypt_payload(payload, encr_payload, keyfile,
1827                     encryption_type)) {
1828                         goto cleanup;
1829                 }

1831                 payload = encr_payload;
1832         }

1834         /*
1835          * Compute the hash (actual or null).
1836          */
1837         if ((payload_hash = gen_tmppath("hash", net, cid)) == NULL) {
1838                 goto cleanup;
1839         }

1841         if (signature_type != NULL &&
1842             (content != WBCGI_CONTENT_ROOTFS || !https_rootserver)) {
1843                 if (!hash_payload(payload, payload_hash, keyfile)) {
1844                         goto cleanup;
1845                 }
1846         } else {
1847                 if (!create_null_hash(payload_hash)) {
1848                         goto cleanup;
1849                 }
1850         }

1852         /*
1853          * For the rootfs the actual payload transmitted is the file
1854          * containing the size of the rootfs (as a string of ascii digits);
1855          * point payload at this instead.
1856          */
1857         if (content == WBCGI_CONTENT_ROOTFS) {
1858                 payload = miniroot_info;
1859         }

1861         /*
1862          * Finally, deliver the payload and hash as a multipart message.
1863          */
1864         if (!deliver_payload(payload, payload_hash)) {
1865                 goto cleanup;
1866         }

1868         ret = WBCGI_STATUS_OK;
1869 cleanup:
1870         /*
1871          * Clean up temporary files.
1872          */
1873         if (wanbootfs_image != NULL &&
1874             WBCGI_FILE_EXISTS(wanbootfs_image, sbuf)) {
1875                 (void) unlink(wanbootfs_image);
1876         }
1877         if (miniroot_info != NULL &&
1878             WBCGI_FILE_EXISTS(miniroot_info, sbuf)) {
1879                 (void) unlink(miniroot_info);
1880         }
1881         if (encr_payload != NULL &&
1882             WBCGI_FILE_EXISTS(encr_payload, sbuf)) {
1883                 (void) unlink(encr_payload);
1884         }
1885         if (payload_hash != NULL &&
1886             WBCGI_FILE_EXISTS(payload_hash, sbuf)) {
1887                 (void) unlink(payload_hash);
```



```
1888         }

1890         /*
1891          * Free up any allocated strings.
1892          */
1893         free_path(&bootconf);
1894         free_path(&keyfile);
1895         free_path(&bootpath);
1896         free_path(&wanbootfs_image);
1897         free_path(&rootpath);
1898         free_path(&miniroot_info);
1899         free_path(&encr_payload);
1900         free_path(&payload_hash);

1902         bootconf_end(&bc_handle);

1904         return (ret);
1905 }
```

```
**********************************************************
    2110 Tue May 20 20:20:11 2014
new/usr/src/lib/libkmf/plugins/kmf_openssl/Makefile.com
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# Makefile for KMF Plugins
#

LIBRARY=        kmf_openssl.a
VERS=           .1

OBJECTS=        openssl_spi.o

include $(SRC)/lib/Makefile.lib

LIBLINKS=       $(DYNLIB:.so.1=.so)
KMFINC=         -I../../../include -I../../../ber_der/inc

BERLIB=         -lkmf -lkmfberder
BERLIB64=       $(BERLIB)

OPENSSLLIBS=    $(BERLIB) -lcrypto -lcryptoutil -lc
OPENSSLLIBS64=  $(BERLIB64) -lcrypto -lcryptoutil -lc

LINTSSLLIBS     = $(BERLIB) -lcryptoutil -lc
LINTSSLLIBS64   = $(BERLIB64) -lcryptoutil -lc

# OpenSSL 1.0 and 0.9.8 produce different lint warnings
LINTFLAGS   +=    -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED
LINTFLAGS64 +=    -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED

#endif /* ! codereview */
SRCDIR=         ../common
INCDIR=         ../../include

CFLAGS          +=      $(CCVERBOSE)
CPPFLAGS        +=      -D_REENTRANT $(KMFINC) \
                        -I$(INCDIR) -I$(ADJUNCT_PROTO)/usr/include/libxml2

CERRWARN        +=      -_gcc=-Wno-unused-label
CERRWARN        +=      -_gcc=-Wno-unused-value
CERRWARN        +=      -_gcc=-Wno-uninitialized
```



```
PICS=   $(OBJECTS:%=pics/%)

lint:=  OPENSSLLIBS=    $(LINTSSLLIBS)
lint:=  OPENSSLLIBS64=  $(LINTSSLLIBS64)

LDLIBS32        +=      $(OPENSSLLIBS)

ROOTLIBDIR=     $(ROOTFS_LIBDIR)/crypto
ROOTLIBDIR64=   $(ROOTFS_LIBDIR)/crypto/$(MACH64)

.KEEP_STATE:

LIBS    =       $(DYNLIB)
all:    $(DYNLIB) $(LINTLIB)

lint: lintcheck

FRC:

include $(SRC)/lib/Makefile.targ
```

```
**********************************************************
  133868 Tue May 20 20:20:11 2014
new/usr/src/lib/libkmf/plugins/kmf_openssl/common/openssl_spi.c
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
**********************************************************
______unchanged_portion_omitted_

2483 /* ocsp_find_signer_sk() is copied from openssl source */
2484 static X509 *ocsp_find_signer_sk(STACK_OF(X509) *certs, OCSP_RESPID *id)
2485 {
2486         int i;
2487         unsigned char tmphash[SHA_DIGEST_LENGTH], *keyhash;

2489         /* Easy if lookup by name */
2490         if (id->type == V_OCSP_RESPID_NAME)
2491                 return (X509_find_by_subject(certs, id->value.byName));

2493         /* Lookup by key hash */

2495         /* If key hash isn't SHA1 length then forget it */
2496         if (id->value.byKey->length != SHA_DIGEST_LENGTH)
2497                 return (NULL);

2499         keyhash = id->value.byKey->data;
2500         /* Calculate hash of each key and compare */
2501         for (i = 0; i < sk_X509_num(certs); i++) {
2502                 /* LINTED E_BAD_PTR_CAST_ALIGN */
2503 #endif /* ! codereview */
2504                 X509 *x = sk_X509_value(certs, i);
2505                 /* Use pubkey_digest to get the key ID value */
2506                 (void) X509_pubkey_digest(x, EVP_sha1(), tmphash, NULL);
2507                 if (!memcmp(keyhash, tmphash, SHA_DIGEST_LENGTH))
2508                         return (x);
2509         }
2510         return (NULL);
2511 }

2513 /* ocsp_find_signer() is copied from openssl source */
2514 /* ARGSUSED2 */
2515 static int
2516 ocsp_find_signer(X509 **psigner, OCSP_BASICRESP *bs, STACK_OF(X509) *certs,
2517     X509_STORE *st, unsigned long flags)
2518 {
2519         X509 *signer;
2520         OCSP_RESPID *rid = bs->tbsResponseData->responderId;
2521         if ((signer = ocsp_find_signer_sk(certs, rid))) {
2522                 *psigner = signer;
2523                 return (2);
2524         }
2525         if (!(flags & OCSP_NOINTERN) &&
2526             (signer = ocsp_find_signer_sk(bs->certs, rid))) {
2527                 *psigner = signer;
2528                 return (1);
2529         }
2530         /* Maybe lookup from store if by subject name */

2532         *psigner = NULL;
2533         return (0);
2534 }

2536 /*
2537  * This function will verify the signature of a basic response, using
2538  * the public key from the OCSP responder certificate.
2539  */
2540 static KMF_RETURN
2541 check_response_signature(KMF_HANDLE_T handle, OCSP_BASICRESP *bs,
```



```
2542     KMF_DATA *signer_cert, KMF_DATA *issuer_cert)
2543 {
2544         KMF_RETURN ret = KMF_OK;
2545         KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
2546         STACK_OF(X509) *cert_stack = NULL;
2547         X509 *signer = NULL;
2548         X509 *issuer = NULL;
2549         EVP_PKEY *skey = NULL;
2550         unsigned char *ptmp;


2553         if (bs == NULL || issuer_cert == NULL)
2554                 return (KMF_ERR_BAD_PARAMETER);

2556         /*
2557          * Find the certificate that signed the basic response.
2558          *
2559          * If signer_cert is not NULL, we will use that as the signer cert.
2560          * Otherwise, we will check if the issuer cert is actually the signer.
2561          * If we still do not find a signer, we will look for it from the
2562          * certificate list came with the response file.
2563          */
2564         if (signer_cert != NULL) {
2565                 ptmp = signer_cert->Data;
2566                 signer = d2i_X509(NULL, (const uchar_t **)&ptmp,
2567                     signer_cert->Length);
2568                 if (signer == NULL) {
2569                         SET_ERROR(kmfh, ERR_get_error());
2570                         ret = KMF_ERR_OCSP_BAD_SIGNER;
2571                         goto end;
2572                 }
2573         } else {
2574                 /*
2575                  * Convert the issuer cert into X509 and push it into a
2576                  * stack to be used by ocsp_find_signer().
2577                  */
2578                 ptmp = issuer_cert->Data;
2579                 issuer = d2i_X509(NULL, (const uchar_t **)&ptmp,
2580                     issuer_cert->Length);
2581                 if (issuer == NULL) {
2582                         SET_ERROR(kmfh, ERR_get_error());
2583                         ret = KMF_ERR_OCSP_BAD_ISSUER;
2584                         goto end;
2585                 }

2587                 if ((cert_stack = sk_X509_new_null()) == NULL) {
2588                         ret = KMF_ERR_INTERNAL;
2589                         goto end;
2590                 }

2592                 if (sk_X509_push(cert_stack, issuer) == NULL) {
2593                         ret = KMF_ERR_INTERNAL;
2594                         goto end;
2595                 }

2597                 ret = ocsp_find_signer(&signer, bs, cert_stack, NULL, 0);
2598                 if (!ret) {
2599                         /* can not find the signer */
2600                         ret = KMF_ERR_OCSP_BAD_SIGNER;
2601                         goto end;
2602                 }
2603         }

2605         /* Verify the signature of the response */
2606         skey = X509_get_pubkey(signer);
2607         if (skey == NULL) {
```

```
2608                 ret = KMF_ERR_OCSP_BAD_SIGNER;
2609                 goto end;
2610         }

2612         ret = OCSP_BASICRESP_verify(bs, skey, 0);
2613         if (ret == 0) {
2614                 ret = KMF_ERR_OCSP_RESPONSE_SIGNATURE;
2615                 goto end;
2616         }

2618 end:
2619         if (issuer != NULL) {
2620                 X509_free(issuer);
2621         }

2623         if (signer != NULL) {
2624                 X509_free(signer);
2625         }

2627         if (skey != NULL) {
2628                 EVP_PKEY_free(skey);
2629         }

2631         if (cert_stack != NULL) {
2632                 sk_X509_free(cert_stack);
2633         }

2635         return (ret);
2636 }



2640 KMF_RETURN
2641 OpenSSL_GetOCSPStatusForCert(KMF_HANDLE_T handle,
2642         int numattr, KMF_ATTRIBUTE *attrlist)
2643 {
2644         KMF_RETURN ret = KMF_OK;
2645         BIO *derbio = NULL;
2646         OCSP_RESPONSE *resp = NULL;
2647         OCSP_BASICRESP *bs = NULL;
2648         OCSP_CERTID *id = NULL;
2649         OCSP_SINGLERESP *single = NULL;
2650         ASN1_GENERALIZEDTIME *rev, *thisupd, *nextupd;
2651         int index, status, reason;
2652         KMF_DATA *issuer_cert;
2653         KMF_DATA *user_cert;
2654         KMF_DATA *signer_cert;
2655         KMF_DATA *response;
2656         int *response_reason, *response_status, *cert_status;
2657         boolean_t ignore_response_sign = B_FALSE;       /* default is FALSE */
2658         uint32_t response_lifetime;

2660         issuer_cert = kmf_get_attr_ptr(KMF_ISSUER_CERT_DATA_ATTR,
2661             attrlist, numattr);
2662         if (issuer_cert == NULL)
2663                 return (KMF_ERR_BAD_PARAMETER);

2665         user_cert = kmf_get_attr_ptr(KMF_USER_CERT_DATA_ATTR,
2666             attrlist, numattr);
2667         if (user_cert == NULL)
2668                 return (KMF_ERR_BAD_PARAMETER);

2670         response = kmf_get_attr_ptr(KMF_OCSP_RESPONSE_DATA_ATTR,
2671             attrlist, numattr);
2672         if (response == NULL)
2673                 return (KMF_ERR_BAD_PARAMETER);
```



```
2675         response_status = kmf_get_attr_ptr(KMF_OCSP_RESPONSE_STATUS_ATTR,
2676             attrlist, numattr);
2677         if (response_status == NULL)
2678                 return (KMF_ERR_BAD_PARAMETER);

2680         response_reason = kmf_get_attr_ptr(KMF_OCSP_RESPONSE_REASON_ATTR,
2681             attrlist, numattr);
2682         if (response_reason == NULL)
2683                 return (KMF_ERR_BAD_PARAMETER);

2685         cert_status = kmf_get_attr_ptr(KMF_OCSP_RESPONSE_CERT_STATUS_ATTR,
2686             attrlist, numattr);
2687         if (cert_status == NULL)
2688                 return (KMF_ERR_BAD_PARAMETER);

2690         /* Read in the response */
2691         derbio = BIO_new_mem_buf(response->Data, response->Length);
2692         if (!derbio) {
2693                 ret = KMF_ERR_MEMORY;
2694                 return (ret);
2695         }

2697         resp = d2i_OCSP_RESPONSE_bio(derbio, NULL);
2698         if (resp == NULL) {
2699                 ret = KMF_ERR_OCSP_MALFORMED_RESPONSE;
2700                 goto end;
2701         }

2703         /* Check the response status */
2704         status = OCSP_response_status(resp);
2705         *response_status = status;
2706         if (status != OCSP_RESPONSE_STATUS_SUCCESSFUL) {
2707                 ret = KMF_ERR_OCSP_RESPONSE_STATUS;
2708                 goto end;
2709         }

2711 #ifdef DEBUG
2712         printf("Successfully checked the response file status.\n");
2713 #endif /* DEBUG */

2715         /* Extract basic response */
2716         bs = OCSP_response_get1_basic(resp);
2717         if (bs == NULL) {
2718                 ret = KMF_ERR_OCSP_NO_BASIC_RESPONSE;
2719                 goto end;
2720         }

2722 #ifdef DEBUG
2723         printf("Successfully retrieved the basic response.\n");
2724 #endif /* DEBUG */

2726         /* Check the basic response signature if required */
2727         ret = kmf_get_attr(KMF_IGNORE_RESPONSE_SIGN_ATTR, attrlist, numattr,
2728             (void *)&ignore_response_sign, NULL);
2729         if (ret != KMF_OK)
2730                 ret = KMF_OK;

2732         signer_cert = kmf_get_attr_ptr(KMF_SIGNER_CERT_DATA_ATTR,
2733             attrlist, numattr);

2735         if (ignore_response_sign == B_FALSE) {
2736                 ret = check_response_signature(handle, bs,
2737                     signer_cert, issuer_cert);
2738                 if (ret != KMF_OK)
2739                         goto end;
```

```
2740         }

2742 #ifdef DEBUG
2743         printf("Successfully verified the response signature.\n");
2744 #endif /* DEBUG */

2746         /* Create a certid for the certificate in question */
2747         ret = create_certid(handle, issuer_cert, user_cert, &id);
2748         if (ret != KMF_OK) {
2749                 ret = KMF_ERR_OCSP_CERTID;
2750                 goto end;
2751         }

2753 #ifdef DEBUG
2754         printf("successfully created a certid for the cert.\n");
2755 #endif /* DEBUG */

2757         /* Find the index of the single response for the certid */
2758         index = OCSP_resp_find(bs, id, -1);
2759         if (index < 0) {
2760                 /* cound not find this certificate in the response */
2761                 ret = KMF_ERR_OCSP_UNKNOWN_CERT;
2762                 goto end;
2763         }

2765 #ifdef DEBUG
2766         printf("Successfully found the single response index for the cert.\n");
2767 #endif /* DEBUG */

2769         /* Retrieve the single response and get the cert status */
2770         single = OCSP_resp_get0(bs, index);
2771         status = OCSP_single_get0_status(single, &reason, &rev, &thisupd,
2772             &nextupd);
2773         if (status == V_OCSP_CERTSTATUS_GOOD) {
2774                 *cert_status = OCSP_GOOD;
2775         } else if (status == V_OCSP_CERTSTATUS_UNKNOWN) {
2776                 *cert_status = OCSP_UNKNOWN;
2777         } else { /* revoked */
2778                 *cert_status = OCSP_REVOKED;
2779                 *response_reason = reason;
2780         }
2781         ret = KMF_OK;

2783         /* resp. time is optional, so we don't care about the return code. */
2784         (void) kmf_get_attr(KMF_RESPONSE_LIFETIME_ATTR, attrlist, numattr,
2785             (void *)&response_lifetime, NULL);

2787         if (!OCSP_check_validity(thisupd, nextupd, 300,
2788             response_lifetime)) {
2789                 ret = KMF_ERR_OCSP_STATUS_TIME_INVALID;
2790                 goto end;
2791         }

2793 #ifdef DEBUG
2794         printf("Successfully verify the time.\n");
2795 #endif /* DEBUG */

2797 end:
2798         if (derbio != NULL)
2799                 (void) BIO_free(derbio);

2801         if (resp != NULL)
2802                 OCSP_RESPONSE_free(resp);

2804         if (bs != NULL)
2805                 OCSP_BASICRESP_free(bs);
```



```
2807         if (id != NULL)
2808                 OCSP_CERTID_free(id);

2810         return (ret);
2811 }

2813 static KMF_RETURN
2814 fetch_key(KMF_HANDLE_T handle, char *path,
2815         KMF_KEY_CLASS keyclass, KMF_KEY_HANDLE *key)
2816 {
2817         KMF_RETURN rv = KMF_OK;
2818         EVP_PKEY *pkey = NULL;
2819         KMF_RAW_SYM_KEY *rkey = NULL;

2821         if (keyclass == KMF_ASYM_PRI ||
2822             keyclass == KMF_ASYM_PUB) {
2823                 pkey = openssl_load_key(handle, path);
2824                 if (pkey == NULL) {
2825                         return (KMF_ERR_KEY_NOT_FOUND);
2826                 }
2827                 if (key != NULL) {
2828                         if (pkey->type == EVP_PKEY_RSA)
2829                                 key->keyalg = KMF_RSA;
2830                         else if (pkey->type == EVP_PKEY_DSA)
2831                                 key->keyalg = KMF_DSA;

2833                         key->kstype = KMF_KEYSTORE_OPENSSL;
2834                         key->keyclass = keyclass;
2835                         key->keyp = (void *)pkey;
2836                         key->israw = FALSE;
2837                         if (path != NULL &&
2838                             ((key->keylabel = strdup(path)) == NULL)) {
2839                                 EVP_PKEY_free(pkey);
2840                                 return (KMF_ERR_MEMORY);
2841                         }
2842                 } else {
2843                         EVP_PKEY_free(pkey);
2844                         pkey = NULL;
2845                 }
2846         } else if (keyclass == KMF_SYMMETRIC) {
2847                 KMF_ENCODE_FORMAT fmt;
2848                 /*
2849                  * If the file is a recognized format,
2850                  * then it is NOT a symmetric key.
2851                  */
2852                 rv = kmf_get_file_format(path, &fmt);
2853                 if (rv == KMF_OK || fmt != 0) {
2854                         return (KMF_ERR_KEY_NOT_FOUND);
2855                 } else if (rv == KMF_ERR_ENCODING) {
2856                         /*
2857                          * If we don't know the encoding,
2858                          * it is probably  a symmetric key.
2859                          */
2860                         rv = KMF_OK;
2861                 } else if (rv == KMF_ERR_OPEN_FILE) {
2862                         return (KMF_ERR_KEY_NOT_FOUND);
2863                 }

2865                 if (key != NULL) {
2866                         KMF_DATA keyvalue;
2867                         rkey = malloc(sizeof (KMF_RAW_SYM_KEY));
2868                         if (rkey == NULL) {
2869                                 rv = KMF_ERR_MEMORY;
2870                                 goto out;
2871                         }
```

```
2873                        (void) memset(rkey, 0, sizeof (KMF_RAW_SYM_KEY));
2874                        rv = kmf_read_input_file(handle, path, &keyvalue);
2875                        if (rv != KMF_OK)
2876                                goto out;

2878                        rkey->keydata.len = keyvalue.Length;
2879                        rkey->keydata.val = keyvalue.Data;

2881                        key->kstype = KMF_KEYSTORE_OPENSSL;
2882                        key->keyclass = keyclass;
2883                        key->israw = TRUE;
2884                        key->keyp = (void *)rkey;
2885                        if (path != NULL &&
2886                            ((key->keylabel = strdup(path)) == NULL)) {
2887                                rv = KMF_ERR_MEMORY;
2888                        }
2889                }
2890        }
2891 out:
2892        if (rv != KMF_OK) {
2893                if (rkey != NULL) {
2894                        kmf_free_raw_sym_key(rkey);
2895                }
2896                if (pkey != NULL)
2897                        EVP_PKEY_free(pkey);

2899                if (key != NULL) {
2900                        key->keyalg = KMF_KEYALG_NONE;
2901                        key->keyclass = KMF_KEYCLASS_NONE;
2902                        key->keyp = NULL;
2903                }
2904        }

2906        return (rv);
2907 }

2909 KMF_RETURN
2910 OpenSSL_FindKey(KMF_HANDLE_T handle,
2911        int numattr, KMF_ATTRIBUTE *attrlist)
2912 {
2913        KMF_RETURN rv = KMF_OK;
2914        char *fullpath = NULL;
2915        uint32_t maxkeys;
2916        KMF_KEY_HANDLE *key;
2917        uint32_t *numkeys;
2918        KMF_KEY_CLASS keyclass;
2919        KMF_RAW_KEY_DATA *rawkey;
2920        char *dirpath;
2921        char *keyfile;

2923        if (handle == NULL)
2924                return (KMF_ERR_BAD_PARAMETER);

2926        numkeys = kmf_get_attr_ptr(KMF_COUNT_ATTR, attrlist, numattr);
2927        if (numkeys == NULL)
2928                return (KMF_ERR_BAD_PARAMETER);

2930        rv = kmf_get_attr(KMF_KEYCLASS_ATTR, attrlist, numattr,
2931            (void *)&keyclass, NULL);
2932        if (rv != KMF_OK)
2933                return (KMF_ERR_BAD_PARAMETER);

2935        if (keyclass != KMF_ASYM_PUB &&
2936            keyclass != KMF_ASYM_PRI &&
2937            keyclass != KMF_SYMMETRIC)
```



```
2938                return (KMF_ERR_BAD_KEY_CLASS);

2940        dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);
2941        keyfile = kmf_get_attr_ptr(KMF_KEY_FILENAME_ATTR, attrlist, numattr);

2943        fullpath = get_fullpath(dirpath, keyfile);

2945        if (fullpath == NULL)
2946                return (KMF_ERR_BAD_PARAMETER);

2948        maxkeys = *numkeys;
2949        if (maxkeys == 0)
2950                maxkeys = 0xFFFFFFFF;
2951        *numkeys = 0;

2953        key = kmf_get_attr_ptr(KMF_KEY_HANDLE_ATTR, attrlist, numattr);
2954        /* it is okay to have "keys" contains NULL */

2956        /*
2957         * The caller may want a list of the raw key data as well.
2958         * Useful for importing keys from a file into other keystores.
2959         */
2960        rawkey = kmf_get_attr_ptr(KMF_RAW_KEY_ATTR, attrlist, numattr);

2962        if (isdir(fullpath)) {
2963                DIR *dirp;
2964                struct dirent *dp;
2965                int n = 0;

2967                /* open all files in the directory and attempt to read them */
2968                if ((dirp = opendir(fullpath)) == NULL) {
2969                        return (KMF_ERR_BAD_PARAMETER);
2970                }
2971                rewinddir(dirp);
2972                while ((dp = readdir(dirp)) != NULL && n < maxkeys) {
2973                        if (strcmp(dp->d_name, ".") &&
2974                            strcmp(dp->d_name, "..")) {
2975                                char *fname;

2977                                fname = get_fullpath(fullpath,
2978                                    (char *)&dp->d_name);

2980                                rv = fetch_key(handle, fname,
2981                                    keyclass, key ? &key[n] : NULL);

2983                                if (rv == KMF_OK) {
2984                                        if (key != NULL && rawkey != NULL)
2985                                                rv = convertToRawKey(
2986                                                    key[n].keyp, &rawkey[n]);
2987                                        n++;
2988                                }

2990                                if (rv != KMF_OK || key == NULL)
2991                                        free(fname);
2992                        }
2993                }
2994                (void) closedir(dirp);
2995                free(fullpath);
2996                (*numkeys) = n;
2997        } else {
2998                rv = fetch_key(handle, fullpath, keyclass, key);
2999                if (rv == KMF_OK)
3000                        (*numkeys) = 1;

3002                if (rv != KMF_OK || key == NULL)
3003                        free(fullpath);
```

```
3005                 if (rv == KMF_OK && key != NULL && rawkey != NULL) {
3006                         rv = convertToRawKey(key->keyp, rawkey);
3007                 }
3008         }

3010         if (rv == KMF_OK && (*numkeys) == 0)
3011                 rv = KMF_ERR_KEY_NOT_FOUND;
3012         else if (rv == KMF_ERR_KEY_NOT_FOUND && (*numkeys) > 0)
3013                 rv = KMF_OK;

3015         return (rv);
3016 }

3018 #define HANDLE_PK12_ERROR { \
3019         SET_ERROR(kmfh, ERR_get_error()); \
3020         rv = KMF_ERR_ENCODING; \
3021         goto out; \
3022 }

3024 static int
3025 add_alias_to_bag(PKCS12_SAFEBAG *bag, X509 *xcert)
3026 {
3027         if (xcert != NULL && xcert->aux != NULL &&
3028             xcert->aux->alias != NULL) {
3029                 if (PKCS12_add_friendlyname_asc(bag,
3030                     (const char *)xcert->aux->alias->data,
3031                     xcert->aux->alias->length) == 0)
3032                         return (0);
3033         }
3034         return (1);
3035 }

3037 static PKCS7 *
3038 add_cert_to_safe(X509 *sslcert, KMF_CREDENTIAL *cred,
3039         uchar_t *keyid, unsigned int keyidlen)
3040 {
3041         PKCS12_SAFEBAG *bag = NULL;
3042         PKCS7 *cert_authsafe = NULL;
3043         STACK_OF(PKCS12_SAFEBAG) *bag_stack;

3045         bag_stack = sk_PKCS12_SAFEBAG_new_null();
3046         if (bag_stack == NULL)
3047                 return (NULL);

3049         /* Convert cert from X509 struct to PKCS#12 bag */
3050         bag = PKCS12_x5092certbag(sslcert);
3051         if (bag == NULL) {
3052                 goto out;
3053         }

3055         /* Add the key id to the certificate bag. */
3056         if (keyidlen > 0 && !PKCS12_add_localkeyid(bag, keyid, keyidlen)) {
3057                 goto out;
3058         }

3060         if (!add_alias_to_bag(bag, sslcert))
3061                 goto out;

3063         /* Pile it on the bag_stack. */
3064         if (!sk_PKCS12_SAFEBAG_push(bag_stack, bag)) {
3065                 goto out;
3066         }
3067         /* Turn bag_stack of certs into encrypted authsafe. */
3068         cert_authsafe = PKCS12_pack_p7encdata(
3069             NID_pbe_WithSHA1And40BitRC2_CBC,
```



```
3070             cred->cred, cred->credlen, NULL, 0,
3071             PKCS12_DEFAULT_ITER, bag_stack);

3073 out:
3074         if (bag_stack != NULL)
3075                 sk_PKCS12_SAFEBAG_pop_free(bag_stack, PKCS12_SAFEBAG_free);

3077         return (cert_authsafe);
3078 }

3080 static PKCS7 *
3081 add_key_to_safe(EVP_PKEY *pkey, KMF_CREDENTIAL *cred,
3082         uchar_t *keyid,  unsigned int keyidlen,
3083         char *label, int label_len)
3084 {
3085         PKCS8_PRIV_KEY_INFO *p8 = NULL;
3086         STACK_OF(PKCS12_SAFEBAG) *bag_stack = NULL;
3087         PKCS12_SAFEBAG *bag = NULL;
3088         PKCS7 *key_authsafe = NULL;

3090         p8 = EVP_PKEY2PKCS8(pkey);
3091         if (p8 == NULL) {
3092                 return (NULL);
3093         }
3094         /* Put the shrouded key into a PKCS#12 bag. */
3095         bag = PKCS12_MAKE_SHKEYBAG(
3096             NID_pbe_WithSHA1And3_Key_TripleDES_CBC,
3097             cred->cred, cred->credlen,
3098             NULL, 0, PKCS12_DEFAULT_ITER, p8);

3100         /* Clean up the PKCS#8 shrouded key, don't need it now. */
3101         PKCS8_PRIV_KEY_INFO_free(p8);
3102         p8 = NULL;

3104         if (bag == NULL) {
3105                 return (NULL);
3106         }
3107         if (keyidlen && !PKCS12_add_localkeyid(bag, keyid, keyidlen))
3108                 goto out;
3109         if (label != NULL && !PKCS12_add_friendlyname(bag, label, label_len))
3110                 goto out;

3112         /* Start a PKCS#12 safebag container for the private key. */
3113         bag_stack = sk_PKCS12_SAFEBAG_new_null();
3114         if (bag_stack == NULL)
3115                 goto out;

3117         /* Pile on the private key on the bag_stack. */
3118         if (!sk_PKCS12_SAFEBAG_push(bag_stack, bag))
3119                 goto out;

3121         key_authsafe = PKCS12_pack_p7data(bag_stack);

3123 out:
3124         if (bag_stack != NULL)
3125                 sk_PKCS12_SAFEBAG_pop_free(bag_stack, PKCS12_SAFEBAG_free);
3126         bag_stack = NULL;
3127         return (key_authsafe);
3128 }

3130 static EVP_PKEY *
3131 ImportRawRSAKey(KMF_RAW_RSA_KEY *key)
3132 {
3133         RSA             *rsa = NULL;
3134         EVP_PKEY        *newkey = NULL;
```

```
3136         if ((rsa = RSA_new()) == NULL)
3137                 return (NULL);
3139         if ((rsa->n = BN_bin2bn(key->mod.val, key->mod.len, rsa->n)) == NULL)
3140                 return (NULL);
3142         if ((rsa->e = BN_bin2bn(key->pubexp.val, key->pubexp.len, rsa->e)) ==
3143             NULL)
3144                 return (NULL);
3146         if (key->priexp.val != NULL)
3147                 if ((rsa->d = BN_bin2bn(key->priexp.val, key->priexp.len,
3148                     rsa->d)) == NULL)
3149                         return (NULL);
3151         if (key->prime1.val != NULL)
3152                 if ((rsa->p = BN_bin2bn(key->prime1.val, key->prime1.len,
3153                     rsa->p)) == NULL)
3154                         return (NULL);
3156         if (key->prime2.val != NULL)
3157                 if ((rsa->q = BN_bin2bn(key->prime2.val, key->prime2.len,
3158                     rsa->q)) == NULL)
3159                         return (NULL);
3161         if (key->exp1.val != NULL)
3162                 if ((rsa->dmp1 = BN_bin2bn(key->exp1.val, key->exp1.len,
3163                     rsa->dmp1)) == NULL)
3164                         return (NULL);
3166         if (key->exp2.val != NULL)
3167                 if ((rsa->dmq1 = BN_bin2bn(key->exp2.val, key->exp2.len,
3168                     rsa->dmq1)) == NULL)
3169                         return (NULL);
3171         if (key->coef.val != NULL)
3172                 if ((rsa->iqmp = BN_bin2bn(key->coef.val, key->coef.len,
3173                     rsa->iqmp)) == NULL)
3174                         return (NULL);
3176         if ((newkey = EVP_PKEY_new()) == NULL)
3177                 return (NULL);
3179         (void) EVP_PKEY_set1_RSA(newkey, rsa);
3181         /* The original key must be freed once here or it leaks memory */
3182         RSA_free(rsa);
3184         return (newkey);
3185 }
3187 static EVP_PKEY *
3188 ImportRawDSAKey(KMF_RAW_DSA_KEY *key)
3189 {
3190         DSA             *dsa = NULL;
3191         EVP_PKEY        *newkey = NULL;
3193         if ((dsa = DSA_new()) == NULL)
3194                 return (NULL);
3196         if ((dsa->p = BN_bin2bn(key->prime.val, key->prime.len,
3197             dsa->p)) == NULL)
3198                 return (NULL);
3200         if ((dsa->q = BN_bin2bn(key->subprime.val, key->subprime.len,
3201             dsa->q)) == NULL)
```



```
3202                 return (NULL);
3204         if ((dsa->g = BN_bin2bn(key->base.val, key->base.len,
3205             dsa->g)) == NULL)
3206                 return (NULL);
3208         if ((dsa->priv_key = BN_bin2bn(key->value.val, key->value.len,
3209             dsa->priv_key)) == NULL)
3210                 return (NULL);
3212         if (key->pubvalue.val != NULL) {
3213                 if ((dsa->pub_key = BN_bin2bn(key->pubvalue.val,
3214                     key->pubvalue.len, dsa->pub_key)) == NULL)
3215                         return (NULL);
3216         }
3218         if ((newkey = EVP_PKEY_new()) == NULL)
3219                 return (NULL);
3221         (void) EVP_PKEY_set1_DSA(newkey, dsa);
3223         /* The original key must be freed once here or it leaks memory */
3224         DSA_free(dsa);
3225         return (newkey);
3226 }
3228 static EVP_PKEY *
3229 raw_key_to_pkey(KMF_KEY_HANDLE *key)
3230 {
3231         EVP_PKEY *pkey = NULL;
3232         KMF_RAW_KEY_DATA *rawkey;
3233         ASN1_TYPE *attr = NULL;
3234         KMF_RETURN ret;
3236         if (key == NULL || !key->israw)
3237                 return (NULL);
3239         rawkey = (KMF_RAW_KEY_DATA *)key->keyp;
3240         if (rawkey->keytype == KMF_RSA) {
3241                 pkey = ImportRawRSAKey(&rawkey->rawdata.rsa);
3242         } else if (rawkey->keytype == KMF_DSA) {
3243                 pkey = ImportRawDSAKey(&rawkey->rawdata.dsa);
3244         } else if (rawkey->keytype == KMF_ECDSA) {
3245                 /*
3246                  * OpenSSL in Solaris does not support EC for
3247                  * legal reasons
3248                  */
3249                 return (NULL);
3250         } else {
3251                 /* wrong kind of key */
3252                 return (NULL);
3253         }
3255         if (rawkey->label != NULL) {
3256                 if ((attr = ASN1_TYPE_new()) == NULL) {
3257                         EVP_PKEY_free(pkey);
3258                         return (NULL);
3259                 }
3260                 attr->value.bmpstring = ASN1_STRING_type_new(V_ASN1_BMPSTRING);
3261                 (void) ASN1_STRING_set(attr->value.bmpstring, rawkey->label,
3262                     strlen(rawkey->label));
3263                 attr->type = V_ASN1_BMPSTRING;
3264                 attr->value.ptr = (char *)attr->value.bmpstring;
3265                 ret = set_pkey_attrib(pkey, attr, NID_friendlyName);
3266                 if (ret != KMF_OK) {
3267                         EVP_PKEY_free(pkey);
```

```
3268                        ASN1_TYPE_free(attr);
3269                        return (NULL);
3270                }
3271        }
3272        if (rawkey->id.Data != NULL) {
3273                if ((attr = ASN1_TYPE_new()) == NULL) {
3274                        EVP_PKEY_free(pkey);
3275                        return (NULL);
3276                }
3277                attr->value.octet_string =
3278                    ASN1_STRING_type_new(V_ASN1_OCTET_STRING);
3279                attr->type = V_ASN1_OCTET_STRING;
3280                (void) ASN1_STRING_set(attr->value.octet_string,
3281                    rawkey->id.Data, rawkey->id.Length);
3282                attr->value.ptr = (char *)attr->value.octet_string;
3283                ret = set_pkey_attrib(pkey, attr, NID_localKeyID);
3284                if (ret != KMF_OK) {
3285                        EVP_PKEY_free(pkey);
3286                        ASN1_TYPE_free(attr);
3287                        return (NULL);
3288                }
3289        }
3290        return (pkey);
3291 }

3293 /*
3294  * Search a list of private keys to find one that goes with the certificate.
3295  */
3296 static EVP_PKEY *
3297 find_matching_key(X509 *xcert, int numkeys, KMF_KEY_HANDLE *keylist)
3298 {
3299        int i;
3300        EVP_PKEY *pkey = NULL;

3302        if (numkeys == 0 || keylist == NULL || xcert == NULL)
3303                return (NULL);
3304        for (i = 0; i < numkeys; i++) {
3305                if (keylist[i].israw)
3306                        pkey = raw_key_to_pkey(&keylist[i]);
3307                else
3308                        pkey = (EVP_PKEY *)keylist[i].keyp;
3309                if (pkey != NULL) {
3310                        if (X509_check_private_key(xcert, pkey)) {
3311                                return (pkey);
3312                        } else {
3313                                EVP_PKEY_free(pkey);
3314                                pkey = NULL;
3315                        }
3316                }
3317        }
3318        return (pkey);
3319 }

3321 static KMF_RETURN
3322 local_export_pk12(KMF_HANDLE_T handle,
3323        KMF_CREDENTIAL *cred,
3324        int numcerts, KMF_X509_DER_CERT *certlist,
3325        int numkeys, KMF_KEY_HANDLE *keylist,
3326        char *filename)
3327 {
3328        KMF_RETURN rv = KMF_OK;
3329        KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
3330        BIO *bio = NULL;
3331        PKCS7 *cert_authsafe = NULL;
3332        PKCS7 *key_authsafe = NULL;
3333        STACK_OF(PKCS7) *authsafe_stack = NULL;
```



```
3334        PKCS12 *p12_elem = NULL;
3335        int i;

3337        if (numcerts == 0 && numkeys == 0)
3338                return (KMF_ERR_BAD_PARAMETER);

3340        /*
3341         * Open the output file.
3342         */
3343        if ((bio = BIO_new_file(filename, "wb")) == NULL) {
3344                SET_ERROR(kmfh, ERR_get_error());
3345                rv = KMF_ERR_OPEN_FILE;
3346                goto cleanup;
3347        }

3349        /* Start a PKCS#7 stack. */
3350        authsafe_stack = sk_PKCS7_new_null();
3351        if (authsafe_stack == NULL) {
3352                rv = KMF_ERR_MEMORY;
3353                goto cleanup;
3354        }
3355        if (numcerts > 0) {
3356                for (i = 0; rv == KMF_OK && i < numcerts; i++) {
3357                        const uchar_t *p = certlist[i].certificate.Data;
3358                        long len = certlist[i].certificate.Length;
3359                        X509 *xcert = NULL;
3360                        EVP_PKEY *pkey = NULL;
3361                        unsigned char keyid[EVP_MAX_MD_SIZE];
3362                        unsigned int keyidlen = 0;

3364                        xcert = d2i_X509(NULL, &p, len);
3365                        if (xcert == NULL) {
3366                                SET_ERROR(kmfh, ERR_get_error());
3367                                rv = KMF_ERR_ENCODING;
3368                        }
3369                        if (certlist[i].kmf_private.label != NULL) {
3370                                /* Set alias attribute */
3371                                (void) X509_alias_set1(xcert,
3372                                    (uchar_t *)certlist[i].kmf_private.label,
3373                                    strlen(certlist[i].kmf_private.label));
3374                        }
3375                        /* Check if there is a key corresponding to this cert */
3376                        pkey = find_matching_key(xcert, numkeys, keylist);

3378                        /*
3379                         * If key is found, get fingerprint and create a
3380                         * safebag.
3381                         */
3382                        if (pkey != NULL) {
3383                                (void) X509_digest(xcert, EVP_sha1(),
3384                                    keyid, &keyidlen);
3385                                key_authsafe = add_key_to_safe(pkey, cred,
3386                                    keyid, keyidlen,
3387                                    certlist[i].kmf_private.label,
3388                                    (certlist[i].kmf_private.label ?
3389                                    strlen(certlist[i].kmf_private.label) : 0));

3391                                if (key_authsafe == NULL) {
3392                                        X509_free(xcert);
3393                                        EVP_PKEY_free(pkey);
3394                                        goto cleanup;
3395                                }
3396                                /* Put the key safe into the Auth Safe */
3397                                if (!sk_PKCS7_push(authsafe_stack,
3398                                    key_authsafe)) {
3399                                        X509_free(xcert);
```

```
3400                                    EVP_PKEY_free(pkey);
3401                                    goto cleanup;
3402                            }
3403                    }

3405                    /* create a certificate safebag */
3406                    cert_authsafe = add_cert_to_safe(xcert, cred, keyid,
3407                        keyidlen);
3408                    if (cert_authsafe == NULL) {
3409                            X509_free(xcert);
3410                            EVP_PKEY_free(pkey);
3411                            goto cleanup;
3412                    }
3413                    if (!sk_PKCS7_push(authsafe_stack, cert_authsafe)) {
3414                            X509_free(xcert);
3415                            EVP_PKEY_free(pkey);
3416                            goto cleanup;
3417                    }

3419                    X509_free(xcert);
3420                    if (pkey)
3421                            EVP_PKEY_free(pkey);
3422            }
3423    } else if (numcerts == 0 && numkeys > 0) {
3424            /*
3425             * If only adding keys to the file.
3426             */
3427            for (i = 0; i < numkeys; i++) {
3428                    EVP_PKEY *pkey = NULL;

3430                    if (keylist[i].israw)
3431                            pkey = raw_key_to_pkey(&keylist[i]);
3432                    else
3433                            pkey = (EVP_PKEY *)keylist[i].keyp;

3435                    if (pkey == NULL)
3436                            continue;

3438                    key_authsafe = add_key_to_safe(pkey, cred,
3439                        NULL, 0, NULL, 0);

3441                    if (key_authsafe == NULL) {
3442                            EVP_PKEY_free(pkey);
3443                            goto cleanup;
3444                    }
3445                    if (!sk_PKCS7_push(authsafe_stack, key_authsafe)) {
3446                            EVP_PKEY_free(pkey);
3447                            goto cleanup;
3448                    }
3449            }
3450    }
3451    p12_elem = PKCS12_init(NID_pkcs7_data);
3452    if (p12_elem == NULL) {
3453            goto cleanup;
3454    }

3456    /* Put the PKCS#7 stack into the PKCS#12 element. */
3457    if (!PKCS12_pack_authsafes(p12_elem, authsafe_stack)) {
3458            goto cleanup;
3459    }

3461    /* Set the integrity MAC on the PKCS#12 element. */
3462    if (!PKCS12_set_mac(p12_elem, cred->cred, cred->credlen,
3463        NULL, 0, PKCS12_DEFAULT_ITER, NULL)) {
3464            goto cleanup;
3465    }
```



```
3467    /* Write the PKCS#12 element to the export file. */
3468    if (!i2d_PKCS12_bio(bio, p12_elem)) {
3469            goto cleanup;
3470    }
3471    PKCS12_free(p12_elem);

3473 cleanup:
3474    /* Clear away the PKCS#7 stack, we're done with it. */
3475    if (authsafe_stack)
3476            sk_PKCS7_pop_free(authsafe_stack, PKCS7_free);

3478    if (bio != NULL)
3479            (void) BIO_free_all(bio);

3481    return (rv);
3482 }

3484 KMF_RETURN
3485 openssl_build_pk12(KMF_HANDLE_T handle, int numcerts,
3486     KMF_X509_DER_CERT *certlist, int numkeys, KMF_KEY_HANDLE *keylist,
3487     KMF_CREDENTIAL *p12cred, char *filename)
3488 {
3489    KMF_RETURN rv;

3491    if (certlist == NULL && keylist == NULL)
3492            return (KMF_ERR_BAD_PARAMETER);

3494    rv = local_export_pk12(handle, p12cred, numcerts, certlist,
3495        numkeys, keylist, filename);

3497    return (rv);
3498 }

3500 KMF_RETURN
3501 OpenSSL_ExportPK12(KMF_HANDLE_T handle, int numattr, KMF_ATTRIBUTE *attrlist)
3502 {
3503    KMF_RETURN rv;
3504    KMF_HANDLE *kmfh = (KMF_HANDLE  *)handle;
3505    char *fullpath = NULL;
3506    char *dirpath = NULL;
3507    char *certfile = NULL;
3508    char *keyfile = NULL;
3509    char *filename = NULL;
3510    KMF_CREDENTIAL *p12cred = NULL;
3511    KMF_X509_DER_CERT certdata;
3512    KMF_KEY_HANDLE key;
3513    int gotkey = 0;
3514    int gotcert = 0;

3516    if (handle == NULL)
3517            return (KMF_ERR_BAD_PARAMETER);

3519    /*
3520     *  First, find the certificate.
3521     */
3522    dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);
3523    certfile = kmf_get_attr_ptr(KMF_CERT_FILENAME_ATTR, attrlist, numattr);
3524    if (certfile != NULL) {
3525            fullpath = get_fullpath(dirpath, certfile);
3526            if (fullpath == NULL)
3527                    return (KMF_ERR_BAD_PARAMETER);

3529            if (isdir(fullpath)) {
3530                    free(fullpath);
3531                    return (KMF_ERR_AMBIGUOUS_PATHNAME);
```

```
3532                 }

3534                 (void) memset(&certdata, 0, sizeof (certdata));
3535                 rv = kmf_load_cert(kmfh, NULL, NULL, NULL, NULL,
3536                     fullpath, &certdata.certificate);
3537                 if (rv != KMF_OK)
3538                         goto end;

3540                 gotcert++;
3541                 certdata.kmf_private.keystore_type = KMF_KEYSTORE_OPENSSL;
3542                 free(fullpath);
3543         }

3545         /*
3546          * Now find the private key.
3547          */
3548         keyfile = kmf_get_attr_ptr(KMF_KEY_FILENAME_ATTR, attrlist, numattr);
3549         if (keyfile != NULL) {
3550                 fullpath = get_fullpath(dirpath, keyfile);
3551                 if (fullpath == NULL)
3552                         return (KMF_ERR_BAD_PARAMETER);

3554                 if (isdir(fullpath)) {
3555                         free(fullpath);
3556                         return (KMF_ERR_AMBIGUOUS_PATHNAME);
3557                 }

3559                 (void) memset(&key, 0, sizeof (KMF_KEY_HANDLE));
3560                 rv = fetch_key(handle, fullpath, KMF_ASYM_PRI, &key);
3561                 if (rv != KMF_OK)
3562                         goto end;
3563                 gotkey++;
3564         }

3566         /*
3567          * Open the output file.
3568          */
3569         filename = kmf_get_attr_ptr(KMF_OUTPUT_FILENAME_ATTR, attrlist,
3570             numattr);
3571         if (filename == NULL) {
3572                 rv = KMF_ERR_BAD_PARAMETER;
3573                 goto end;
3574         }

3576         /* Stick the key and the cert into a PKCS#12 file */
3577         p12cred = kmf_get_attr_ptr(KMF_PK12CRED_ATTR, attrlist, numattr);
3578         if (p12cred == NULL) {
3579                 rv = KMF_ERR_BAD_PARAMETER;
3580                 goto end;
3581         }

3583         rv = local_export_pk12(handle, p12cred, 1, &certdata,
3584             1, &key, filename);

3586 end:
3587         if (fullpath)
3588                 free(fullpath);

3590         if (gotcert)
3591                 kmf_free_kmf_cert(handle, &certdata);
3592         if (gotkey)
3593                 kmf_free_kmf_key(handle, &key);
3594         return (rv);
3595 }

3597 /*
```



```
3598  * Helper function to extract keys and certificates from
3599  * a single PEM file.  Typically the file should contain a
3600  * private key and an associated public key wrapped in an x509 cert.
3601  * However, the file may be just a list of X509 certs with no keys.
3602  */
3603 static KMF_RETURN
3604 extract_pem(KMF_HANDLE *kmfh,
3605         char *issuer, char *subject, KMF_BIGINT *serial,
3606         char *filename, CK_UTF8CHAR *pin,
3607         CK_ULONG pinlen, EVP_PKEY **priv_key, KMF_DATA **certs,
3608         int *numcerts)
3609 /* ARGSUSED6 */
3610 {
3611         KMF_RETURN rv = KMF_OK;
3612         FILE *fp;
3613         STACK_OF(X509_INFO) *x509_info_stack = NULL;
3614         int i, ncerts = 0, matchcerts = 0;
3615         EVP_PKEY *pkey = NULL;
3616         X509_INFO *info;
3617         X509 *x;
3618         X509_INFO **cert_infos = NULL;
3619         KMF_DATA *certlist = NULL;

3621         if (priv_key)
3622                 *priv_key = NULL;
3623         if (certs)
3624                 *certs = NULL;
3625         fp = fopen(filename, "r");
3626         if (fp == NULL)
3627                 return (KMF_ERR_OPEN_FILE);

3629         x509_info_stack = PEM_X509_INFO_read(fp, NULL, NULL, pin);
3630         if (x509_info_stack == NULL) {
3631                 (void) fclose(fp);
3632                 return (KMF_ERR_ENCODING);
3633         }
3634         cert_infos = (X509_INFO **)malloc(sk_X509_INFO_num(x509_info_stack) *
3635             sizeof (X509_INFO *));
3636         if (cert_infos == NULL) {
3637                 (void) fclose(fp);
3638                 rv = KMF_ERR_MEMORY;
3639                 goto err;
3640         }

3642         for (i = 0; i < sk_X509_INFO_num(x509_info_stack); i++) {
3643                 /* LINTED E_BAD_PTR_CAST_ALIGN */
3644 #endif /* ! codereview */
3645                 cert_infos[ncerts] = sk_X509_INFO_value(x509_info_stack, i);
3646                 ncerts++;
3647         }

3649         if (ncerts == 0) {
3650                 (void) fclose(fp);
3651                 rv = KMF_ERR_CERT_NOT_FOUND;
3652                 goto err;
3653         }

3655         if (priv_key != NULL) {
3656                 rewind(fp);
3657                 pkey = PEM_read_PrivateKey(fp, NULL, NULL, pin);
3658         }
3659         (void) fclose(fp);

3661         x = cert_infos[ncerts - 1]->x509;
3662         /*
3663          * Make sure the private key matchs the last cert in the file.
```

```
3664          */
3665         if (pkey != NULL && !X509_check_private_key(x, pkey)) {
3666                 EVP_PKEY_free(pkey);
3667                 rv = KMF_ERR_KEY_MISMATCH;
3668                 goto err;
3669         }

3671         certlist = (KMF_DATA *)calloc(ncerts, sizeof (KMF_DATA));
3672         if (certlist == NULL) {
3673                 if (pkey != NULL)
3674                         EVP_PKEY_free(pkey);
3675                 rv = KMF_ERR_MEMORY;
3676                 goto err;
3677         }

3679         /*
3680          * Convert all of the certs to DER format.
3681          */
3682         matchcerts = 0;
3683         for (i = 0; rv == KMF_OK && certs != NULL && i < ncerts; i++) {
3684                 boolean_t match = FALSE;
3685                 info =  cert_infos[ncerts - 1 - i];

3687                 rv = check_cert(info->x509, issuer, subject, serial, &match);
3688                 if (rv != KMF_OK || match != TRUE) {
3689                         rv = KMF_OK;
3690                         continue;
3691                 }

3693                 rv = ssl_cert2KMFDATA(kmfh, info->x509,
3694                         &certlist[matchcerts++]);

3696                 if (rv != KMF_OK) {
3697                         int j;
3698                         for (j = 0; j < matchcerts; j++)
3699                                 kmf_free_data(&certlist[j]);
3700                         free(certlist);
3701                         certlist = NULL;
3702                         ncerts = matchcerts = 0;
3703                 }
3704         }

3706         if (numcerts != NULL)
3707                 *numcerts = matchcerts;

3709         if (certs != NULL)
3710                 *certs = certlist;
3711         else if (certlist != NULL) {
3712                 for (i = 0; i < ncerts; i++)
3713                         kmf_free_data(&certlist[i]);
3714                 free(certlist);
3715                 certlist = NULL;
3716         }

3718         if (priv_key == NULL && pkey != NULL)
3719                 EVP_PKEY_free(pkey);
3720         else if (priv_key != NULL && pkey != NULL)
3721                 *priv_key = pkey;

3723 err:
3724         /* Cleanup the stack of X509 info records */
3725         for (i = 0; i < sk_X509_INFO_num(x509_info_stack); i++) {
3726                 /* LINTED E_BAD_PTR_CAST_ALIGN */
3727 #endif /* ! codereview */
3728                 info = (X509_INFO *)sk_X509_INFO_value(x509_info_stack, i);
3729                 X509_INFO_free(info);
```



```
3730         }
3731         if (x509_info_stack)
3732                 sk_X509_INFO_free(x509_info_stack);

3734         if (cert_infos != NULL)
3735                 free(cert_infos);

3737         return (rv);
3738 }

3740 static KMF_RETURN
3741 openssl_parse_bags(STACK_OF(PKCS12_SAFEBAG) *bags, char *pin,
3742         STACK_OF(EVP_PKEY) *keys, STACK_OF(X509) *certs)
3743 {
3744         KMF_RETURN ret;
3745         int i;

3747         for (i = 0; i < sk_PKCS12_SAFEBAG_num(bags); i++) {
3748                 /* LINTED E_BAD_PTR_CAST_ALIGN */
3749 #endif /* ! codereview */
3750                 PKCS12_SAFEBAG *bag = sk_PKCS12_SAFEBAG_value(bags, i);
3751                 ret = openssl_parse_bag(bag, pin, (pin ? strlen(pin) : 0),
3752                     keys, certs);

3754                 if (ret != KMF_OK)
3755                         return (ret);
3756         }

3758         return (ret);
3759 }

3761 static KMF_RETURN
3762 set_pkey_attrib(EVP_PKEY *pkey, ASN1_TYPE *attrib, int nid)
3763 {
3764         X509_ATTRIBUTE *attr = NULL;

3766         if (pkey == NULL || attrib == NULL)
3767                 return (KMF_ERR_BAD_PARAMETER);

3769         if (pkey->attributes == NULL) {
3770                 pkey->attributes = sk_X509_ATTRIBUTE_new_null();
3771                 if (pkey->attributes == NULL)
3772                         return (KMF_ERR_MEMORY);
3773         }
3774         attr = X509_ATTRIBUTE_create(nid, attrib->type, attrib->value.ptr);
3775         if (attr != NULL) {
3776                 int i;
3777                 X509_ATTRIBUTE *a;
3778                 for (i = 0;
3779                     i < sk_X509_ATTRIBUTE_num(pkey->attributes); i++) {
3780                         /* LINTED E_BAD_PTR_CASE_ALIGN */
3781 #endif /* ! codereview */
3782                         a = sk_X509_ATTRIBUTE_value(pkey->attributes, i);
3783                         if (OBJ_obj2nid(a->object) == nid) {
3784                                 X509_ATTRIBUTE_free(a);
3785                                 /* LINTED E_BAD_PTR_CAST_ALIGN */
3786 #endif /* ! codereview */
3787                                 (void) sk_X509_ATTRIBUTE_set(pkey->attributes,
3788                                     i, attr);
3789                                 return (KMF_OK);
3790                         }
3791                 }
3792                 if (sk_X509_ATTRIBUTE_push(pkey->attributes, attr) == NULL) {
3793                         X509_ATTRIBUTE_free(attr);
3794                         return (KMF_ERR_MEMORY);
3795                 }
```

```
3796         } else {
3797                 return (KMF_ERR_MEMORY);
3798         }

3800         return (KMF_OK);
3801 }

3803 static KMF_RETURN
3804 openssl_parse_bag(PKCS12_SAFEBAG *bag, char *pass, int passlen,
3805         STACK_OF(EVP_PKEY) *keylist, STACK_OF(X509) *certlist)
3806 {
3807         KMF_RETURN ret = KMF_OK;
3808         PKCS8_PRIV_KEY_INFO *p8 = NULL;
3809         EVP_PKEY *pkey = NULL;
3810         X509 *xcert = NULL;
3811         ASN1_TYPE *keyid = NULL;
3812         ASN1_TYPE *fname = NULL;
3813         uchar_t *data = NULL;

3815         keyid = PKCS12_get_attr(bag, NID_localKeyID);
3816         fname = PKCS12_get_attr(bag, NID_friendlyName);

3818         switch (M_PKCS12_bag_type(bag)) {
3819                 case NID_keyBag:
3820                         if (keylist == NULL)
3821                                 goto end;
3822                         pkey = EVP_PKCS82PKEY(bag->value.keybag);
3823                         if (pkey == NULL)
3824                                 ret = KMF_ERR_PKCS12_FORMAT;

3826                         break;
3827                 case NID_pkcs8ShroudedKeyBag:
3828                         if (keylist == NULL)
3829                                 goto end;
3830                         p8 = M_PKCS12_decrypt_skey(bag, pass, passlen);
3831                         if (p8 == NULL)
3832                                 return (KMF_ERR_AUTH_FAILED);
3833                         pkey = EVP_PKCS82PKEY(p8);
3834                         PKCS8_PRIV_KEY_INFO_free(p8);
3835                         if (pkey == NULL)
3836                                 ret = KMF_ERR_PKCS12_FORMAT;
3837                         break;
3838                 case NID_certBag:
3839                         if (certlist == NULL)
3840                                 goto end;
3841                         if (M_PKCS12_cert_bag_type(bag) != NID_x509Certificate)
3842                                 return (KMF_ERR_PKCS12_FORMAT);
3843                         xcert = M_PKCS12_certbag2x509(bag);
3844                         if (xcert == NULL) {
3845                                 ret = KMF_ERR_PKCS12_FORMAT;
3846                                 goto end;
3847                         }
3848                         if (keyid != NULL) {
3849                                 if (X509_keyid_set1(xcert,
3850                                     keyid->value.octet_string->data,
3851                                     keyid->value.octet_string->length) == 0) {
3852                                         ret = KMF_ERR_PKCS12_FORMAT;
3853                                         goto end;
3854                                 }
3855                         }
3856                         if (fname != NULL) {
3857                                 int len, r;
3858                                 len = ASN1_STRING_to_UTF8(&data,
3859                                     fname->value.asn1_string);
3860                                 if (len > 0 && data != NULL) {
3861                                         r = X509_alias_set1(xcert, data, len);
```



```
3862                                         if (r == NULL) {
3863                                                 ret = KMF_ERR_PKCS12_FORMAT;
3864                                                 goto end;
3865                                         }
3866                                 } else {
3867                                         ret = KMF_ERR_PKCS12_FORMAT;
3868                                         goto end;
3869                                 }
3870                         }
3871                         if (sk_X509_push(certlist, xcert) == 0)
3872                                 ret = KMF_ERR_MEMORY;
3873                         else
3874                                 xcert = NULL;
3875                         break;
3876                 case NID_safeContentsBag:
3877                         return (openssl_parse_bags(bag->value.safes, pass,
3878                             keylist, certlist));
3879                 default:
3880                         ret = KMF_ERR_PKCS12_FORMAT;
3881                         break;
3882         }

3884         /*
3885          * Set the ID and/or FriendlyName attributes on the key.
3886          * If converting to PKCS11 objects, these can translate to CKA_ID
3887          * and CKA_LABEL values.
3888          */
3889         if (pkey != NULL && ret == KMF_OK) {
3890                 ASN1_TYPE *attr = NULL;
3891                 if (keyid != NULL && keyid->type == V_ASN1_OCTET_STRING) {
3892                         if ((attr = ASN1_TYPE_new()) == NULL)
3893                                 return (KMF_ERR_MEMORY);
3894                         attr->value.octet_string =
3895                             ASN1_STRING_dup(keyid->value.octet_string);
3896                         attr->type = V_ASN1_OCTET_STRING;
3897                         attr->value.ptr = (char *)attr->value.octet_string;
3898                         ret = set_pkey_attrib(pkey, attr, NID_localKeyID);
3899                         OPENSSL_free(attr);
3900                 }

3902                 if (ret == KMF_OK && fname != NULL &&
3903                     fname->type == V_ASN1_BMPSTRING) {
3904                         if ((attr = ASN1_TYPE_new()) == NULL)
3905                                 return (KMF_ERR_MEMORY);
3906                         attr->value.bmpstring =
3907                             ASN1_STRING_dup(fname->value.bmpstring);
3908                         attr->type = V_ASN1_BMPSTRING;
3909                         attr->value.ptr = (char *)attr->value.bmpstring;
3910                         ret = set_pkey_attrib(pkey, attr, NID_friendlyName);
3911                         OPENSSL_free(attr);
3912                 }

3914                 if (ret == KMF_OK && keylist != NULL &&
3915                     sk_EVP_PKEY_push(keylist, pkey) == 0)
3916                         ret = KMF_ERR_MEMORY;
3917         }
3918         if (ret == KMF_OK && keylist != NULL)
3919                 pkey = NULL;
3920 end:
3921         if (pkey != NULL)
3922                 EVP_PKEY_free(pkey);
3923         if (xcert != NULL)
3924                 X509_free(xcert);
3925         if (data != NULL)
3926                 OPENSSL_free(data);
```

```
3928         return (ret);
3929 }

3931 static KMF_RETURN
3932 openssl_pkcs12_parse(PKCS12 *p12, char *pin,
3933         STACK_OF(EVP_PKEY) *keys,
3934         STACK_OF(X509) *certs,
3935         STACK_OF(X509) *ca)
3936 /* ARGSUSED3 */
3937 {
3938         KMF_RETURN ret = KMF_OK;
3939         STACK_OF(PKCS7) *asafes = NULL;
3940         STACK_OF(PKCS12_SAFEBAG) *bags = NULL;
3941         int i, bagnid;
3942         PKCS7 *p7;

3944         if (p12 == NULL || (keys == NULL && certs == NULL))
3945                 return (KMF_ERR_BAD_PARAMETER);

3947         if (pin == NULL || *pin == NULL) {
3948                 if (PKCS12_verify_mac(p12, NULL, 0)) {
3949                         pin = NULL;
3950                 } else if (PKCS12_verify_mac(p12, "", 0)) {
3951                         pin = "";
3952                 } else {
3953                         return (KMF_ERR_AUTH_FAILED);
3954                 }
3955         } else if (!PKCS12_verify_mac(p12, pin, -1)) {
3956                 return (KMF_ERR_AUTH_FAILED);
3957         }

3959         if ((asafes = PKCS12_unpack_authsafes(p12)) == NULL)
3960                 return (KMF_ERR_PKCS12_FORMAT);

3962         for (i = 0; ret == KMF_OK && i < sk_PKCS7_num(asafes); i++) {
3963                 bags = NULL;
3964                 /* LINTED E_BAD_PTR_CAST_ALIGN */
3965 #endif /* ! codereview */
3966                 p7 = sk_PKCS7_value(asafes, i);
3967                 bagnid = OBJ_obj2nid(p7->type);

3969                 if (bagnid == NID_pkcs7_data) {
3970                         bags = PKCS12_unpack_p7data(p7);
3971                 } else if (bagnid == NID_pkcs7_encrypted) {
3972                         bags = PKCS12_unpack_p7encdata(p7, pin,
3973                             (pin ? strlen(pin) : 0));
3974                 } else {
3975                         continue;
3976                 }
3977                 if (bags == NULL) {
3978                         ret = KMF_ERR_PKCS12_FORMAT;
3979                         goto out;
3980                 }

3982                 if (openssl_parse_bags(bags, pin, keys, certs) != KMF_OK)
3983                         ret = KMF_ERR_PKCS12_FORMAT;

3985                 sk_PKCS12_SAFEBAG_pop_free(bags, PKCS12_SAFEBAG_free);
3986         }
3987 out:
3988         if (asafes != NULL)
3989                 sk_PKCS7_pop_free(asafes, PKCS7_free);

3991         return (ret);
3992 }
```



```
3994 /*
3995  * Helper function to decrypt and parse PKCS#12 import file.
3996  */
3997 static KMF_RETURN
3998 extract_pkcs12(BIO *fbio, CK_UTF8CHAR *pin, CK_ULONG pinlen,
3999         STACK_OF(EVP_PKEY) **priv_key, STACK_OF(X509) **certs,
4000         STACK_OF(X509) **ca)
4001 /* ARGSUSED2 */
4002 {
4003         PKCS12                  *pk12, *pk12_tmp;
4004         STACK_OF(EVP_PKEY)      *pkeylist = NULL;
4005         STACK_OF(X509)          *xcertlist = NULL;
4006         STACK_OF(X509)          *cacertlist = NULL;

4008         if ((pk12 = PKCS12_new()) == NULL) {
4009                 return (KMF_ERR_MEMORY);
4010         }

4012         if ((pk12_tmp = d2i_PKCS12_bio(fbio, &pk12)) == NULL) {
4013                 /* This is ok; it seems to mean there is no more to read. */
4014                 if (ERR_GET_LIB(ERR_peek_error()) == ERR_LIB_ASN1 &&
4015                     ERR_GET_REASON(ERR_peek_error()) == ASN1_R_HEADER_TOO_LONG)
4016                         goto end_extract_pkcs12;

4018                 PKCS12_free(pk12);
4019                 return (KMF_ERR_PKCS12_FORMAT);
4020         }
4021         pk12 = pk12_tmp;

4023         xcertlist = sk_X509_new_null();
4024         if (xcertlist == NULL) {
4025                 PKCS12_free(pk12);
4026                 return (KMF_ERR_MEMORY);
4027         }
4028         pkeylist = sk_EVP_PKEY_new_null();
4029         if (pkeylist == NULL) {
4030                 sk_X509_pop_free(xcertlist, X509_free);
4031                 PKCS12_free(pk12);
4032                 return (KMF_ERR_MEMORY);
4033         }

4035         if (openssl_pkcs12_parse(pk12, (char *)pin, pkeylist, xcertlist,
4036             cacertlist) != KMF_OK) {
4037                 sk_X509_pop_free(xcertlist, X509_free);
4038                 sk_EVP_PKEY_pop_free(pkeylist, EVP_PKEY_free);
4039                 PKCS12_free(pk12);
4040                 return (KMF_ERR_PKCS12_FORMAT);
4041         }

4043         if (priv_key && pkeylist)
4044                 *priv_key = pkeylist;
4045         else if (pkeylist)
4046                 sk_EVP_PKEY_pop_free(pkeylist, EVP_PKEY_free);
4047         if (certs && xcertlist)
4048                 *certs = xcertlist;
4049         else if (xcertlist)
4050                 sk_X509_pop_free(xcertlist, X509_free);
4051         if (ca && cacertlist)
4052                 *ca = cacertlist;
4053         else if (cacertlist)
4054                 sk_X509_pop_free(cacertlist, X509_free);

4056 end_extract_pkcs12:

4058         PKCS12_free(pk12);
4059         return (KMF_OK);
```

```
4060 }

4062 static KMF_RETURN
4063 sslBN2KMFBN(BIGNUM *from, KMF_BIGINT *to)
4064 {
4065         KMF_RETURN rv = KMF_OK;
4066         uint32_t sz;

4068         sz = BN_num_bytes(from);
4069         to->val = (uchar_t *)malloc(sz);
4070         if (to->val == NULL)
4071                 return (KMF_ERR_MEMORY);

4073         if ((to->len = BN_bn2bin(from, to->val)) != sz) {
4074                 free(to->val);
4075                 to->val = NULL;
4076                 to->len = 0;
4077                 rv = KMF_ERR_MEMORY;
4078         }

4080         return (rv);
4081 }

4083 static KMF_RETURN
4084 exportRawRSAKey(RSA *rsa, KMF_RAW_KEY_DATA *key)
4085 {
4086         KMF_RETURN rv;
4087         KMF_RAW_RSA_KEY *kmfkey = &key->rawdata.rsa;

4089         (void) memset(kmfkey, 0, sizeof (KMF_RAW_RSA_KEY));
4090         if ((rv = sslBN2KMFBN(rsa->n, &kmfkey->mod)) != KMF_OK)
4091                 goto cleanup;

4093         if ((rv = sslBN2KMFBN(rsa->e, &kmfkey->pubexp)) != KMF_OK)
4094                 goto cleanup;

4096         if (rsa->d != NULL)
4097                 if ((rv = sslBN2KMFBN(rsa->d, &kmfkey->priexp)) != KMF_OK)
4098                         goto cleanup;

4100         if (rsa->p != NULL)
4101                 if ((rv = sslBN2KMFBN(rsa->p, &kmfkey->prime1)) != KMF_OK)
4102                         goto cleanup;

4104         if (rsa->q != NULL)
4105                 if ((rv = sslBN2KMFBN(rsa->q, &kmfkey->prime2)) != KMF_OK)
4106                         goto cleanup;

4108         if (rsa->dmp1 != NULL)
4109                 if ((rv = sslBN2KMFBN(rsa->dmp1, &kmfkey->exp1)) != KMF_OK)
4110                         goto cleanup;

4112         if (rsa->dmq1 != NULL)
4113                 if ((rv = sslBN2KMFBN(rsa->dmq1, &kmfkey->exp2)) != KMF_OK)
4114                         goto cleanup;

4116         if (rsa->iqmp != NULL)
4117                 if ((rv = sslBN2KMFBN(rsa->iqmp, &kmfkey->coef)) != KMF_OK)
4118                         goto cleanup;
4119 cleanup:
4120         if (rv != KMF_OK)
4121                 kmf_free_raw_key(key);
4122         else
4123                 key->keytype = KMF_RSA;

4125         /*
```



```
4126          * Free the reference to this key, SSL will not actually free
4127          * the memory until the refcount == 0, so this is safe.
4128          */
4129         RSA_free(rsa);

4131         return (rv);
4132 }

4134 static KMF_RETURN
4135 exportRawDSAKey(DSA *dsa, KMF_RAW_KEY_DATA *key)
4136 {
4137         KMF_RETURN rv;
4138         KMF_RAW_DSA_KEY *kmfkey = &key->rawdata.dsa;

4140         (void) memset(kmfkey, 0, sizeof (KMF_RAW_DSA_KEY));
4141         if ((rv = sslBN2KMFBN(dsa->p, &kmfkey->prime)) != KMF_OK)
4142                 goto cleanup;

4144         if ((rv = sslBN2KMFBN(dsa->q, &kmfkey->subprime)) != KMF_OK)
4145                 goto cleanup;

4147         if ((rv = sslBN2KMFBN(dsa->g, &kmfkey->base)) != KMF_OK)
4148                 goto cleanup;

4150         if ((rv = sslBN2KMFBN(dsa->priv_key, &kmfkey->value)) != KMF_OK)
4151                 goto cleanup;

4153 cleanup:
4154         if (rv != KMF_OK)
4155                 kmf_free_raw_key(key);
4156         else
4157                 key->keytype = KMF_DSA;

4159         /*
4160          * Free the reference to this key, SSL will not actually free
4161          * the memory until the refcount == 0, so this is safe.
4162          */
4163         DSA_free(dsa);

4165         return (rv);
4166 }

4168 static KMF_RETURN
4169 add_cert_to_list(KMF_HANDLE *kmfh, X509 *sslcert,
4170         KMF_X509_DER_CERT **certlist, int *ncerts)
4171 {
4172         KMF_RETURN rv = KMF_OK;
4173         KMF_X509_DER_CERT *list = (*certlist);
4174         KMF_X509_DER_CERT cert;
4175         int n = (*ncerts);

4177         if (list == NULL) {
4178                 list = (KMF_X509_DER_CERT *)malloc(sizeof (KMF_X509_DER_CERT));
4179         } else {
4180                 list = (KMF_X509_DER_CERT *)realloc(list,
4181                     sizeof (KMF_X509_DER_CERT) * (n + 1));
4182         }

4184         if (list == NULL)
4185                 return (KMF_ERR_MEMORY);

4187         (void) memset(&cert, 0, sizeof (cert));
4188         rv = ssl_cert2KMFDATA(kmfh, sslcert, &cert.certificate);
4189         if (rv == KMF_OK) {
4190                 int len = 0;
4191                 /* Get the alias name for the cert if there is one */
```

```
4192                 char *a = (char *)X509_alias_get0(sslcert, &len);
4193                 if (a != NULL)
4194                         cert.kmf_private.label = strdup(a);
4195                 cert.kmf_private.keystore_type = KMF_KEYSTORE_OPENSSL;

4197                 list[n] = cert;
4198                 (*ncerts) = n + 1;

4200                 *certlist = list;
4201         } else {
4202                 free(list);
4203         }

4205         return (rv);
4206 }

4208 static KMF_RETURN
4209 add_key_to_list(KMF_RAW_KEY_DATA **keylist,
4210         KMF_RAW_KEY_DATA *newkey, int *nkeys)
4211 {
4212         KMF_RAW_KEY_DATA *list = (*keylist);
4213         int n = (*nkeys);

4215         if (list == NULL) {
4216                 list = (KMF_RAW_KEY_DATA *)malloc(sizeof (KMF_RAW_KEY_DATA));
4217         } else {
4218                 list = (KMF_RAW_KEY_DATA *)realloc(list,
4219                     sizeof (KMF_RAW_KEY_DATA) * (n + 1));
4220         }

4222         if (list == NULL)
4223                 return (KMF_ERR_MEMORY);

4225         list[n] = *newkey;
4226         (*nkeys) = n + 1;

4228         *keylist = list;

4230         return (KMF_OK);
4231 }

4233 static X509_ATTRIBUTE *
4234 find_attr(STACK_OF(X509_ATTRIBUTE) *attrs, int nid)
4235 {
4236         X509_ATTRIBUTE *a;
4237         int i;

4239         if (attrs == NULL)
4240                 return (NULL);

4242         for (i = 0; i < sk_X509_ATTRIBUTE_num(attrs); i++) {
4243                 /* LINTED E_BAD_PTR_CAST_ALIGN */
4244 #endif /* ! codereview */
4245                 a = sk_X509_ATTRIBUTE_value(attrs, i);
4246                 if (OBJ_obj2nid(a->object) == nid)
4247                         return (a);
4248         }
4249         return (NULL);
4250 }

4252 static KMF_RETURN
4253 convertToRawKey(EVP_PKEY *pkey, KMF_RAW_KEY_DATA *key)
4254 {
4255         KMF_RETURN rv = KMF_OK;
4256         X509_ATTRIBUTE *attr;
```



```
4258         if (pkey == NULL || key == NULL)
4259                 return (KMF_ERR_BAD_PARAMETER);
4260         /* Convert SSL key to raw key */
4261         switch (pkey->type) {
4262                 case EVP_PKEY_RSA:
4263                         rv = exportRawRSAKey(EVP_PKEY_get1_RSA(pkey),
4264                             key);
4265                         if (rv != KMF_OK)
4266                                 return (rv);
4267                         break;
4268                 case EVP_PKEY_DSA:
4269                         rv = exportRawDSAKey(EVP_PKEY_get1_DSA(pkey),
4270                             key);
4271                         if (rv != KMF_OK)
4272                                 return (rv);
4273                         break;
4274                 default:
4275                         return (KMF_ERR_BAD_PARAMETER);
4276         }
4277         /*
4278          * If friendlyName, add it to record.
4279          */
4280         attr = find_attr(pkey->attributes, NID_friendlyName);
4281         if (attr != NULL) {
4282                 ASN1_TYPE *ty = NULL;
4283                 int numattr = sk_ASN1_TYPE_num(attr->value.set);
4284                 if (attr->single == 0 && numattr > 0) {
4285                         /* LINTED E_BAD_PTR_CAST_ALIGN */
4286 #endif /* ! codereview */
4287                         ty = sk_ASN1_TYPE_value(attr->value.set, 0);
4288                 }
4289                 if (ty != NULL) {
4290 #if OPENSSL_VERSION_NUMBER < 0x10000000L
4291                         key->label = uni2asc(ty->value.bmpstring->data,
4292                             ty->value.bmpstring->length);
4293 #else
4294                         key->label = OPENSSL_uni2asc(ty->value.bmpstring->data,
4295                             ty->value.bmpstring->length);
4296 #endif
4297                 }
4298         } else {
4299                 key->label = NULL;
4300         }

4302         /*
4303          * If KeyID, add it to record as a KMF_DATA object.
4304          */
4305         attr = find_attr(pkey->attributes, NID_localKeyID);
4306         if (attr != NULL) {
4307                 ASN1_TYPE *ty = NULL;
4308                 int numattr = sk_ASN1_TYPE_num(attr->value.set);
4309                 if (attr->single == 0 && numattr > 0) {
4310                         /* LINTED E_BAD_PTR_CAST_ALIGN */
4311 #endif /* ! codereview */
4312                         ty = sk_ASN1_TYPE_value(attr->value.set, 0);
4313                 }
4314                 key->id.Data = (uchar_t *)malloc(
4315                     ty->value.octet_string->length);
4316                 if (key->id.Data == NULL)
4317                         return (KMF_ERR_MEMORY);
4318                 (void) memcpy(key->id.Data, ty->value.octet_string->data,
4319                     ty->value.octet_string->length);
4320                 key->id.Length = ty->value.octet_string->length;
4321         } else {
4322                 (void) memset(&key->id, 0, sizeof (KMF_DATA));
4323         }
```

```
4325         return (rv);
4326 }

4328 static KMF_RETURN
4329 convertPK12Objects(
4330         KMF_HANDLE *kmfh,
4331         STACK_OF(EVP_PKEY) *sslkeys,
4332         STACK_OF(X509) *sslcert,
4333         STACK_OF(X509) *sslcacerts,
4334         KMF_RAW_KEY_DATA **keylist, int *nkeys,
4335         KMF_X509_DER_CERT **certlist, int *ncerts)
4336 {
4337         KMF_RETURN rv = KMF_OK;
4338         KMF_RAW_KEY_DATA key;
4339         int i;

4341         for (i = 0; sslkeys != NULL && i < sk_EVP_PKEY_num(sslkeys); i++) {
4342                 /* LINTED E_BAD_PTR_CAST_ALIGN */
4343 #endif /* ! codereview */
4344                 EVP_PKEY *pkey = sk_EVP_PKEY_value(sslkeys, i);
4345                 rv = convertToRawKey(pkey, &key);
4346                 if (rv == KMF_OK)
4347                         rv = add_key_to_list(keylist, &key, nkeys);

4349                 if (rv != KMF_OK)
4350                         return (rv);
4351         }

4353         /* Now add the certificate to the certlist */
4354         for (i = 0; sslcert != NULL && i < sk_X509_num(sslcert); i++) {
4355                 /* LINTED E_BAD_PTR_CAST_ALIGN */
4356 #endif /* ! codereview */
4357                 X509 *cert = sk_X509_value(sslcert, i);
4358                 rv = add_cert_to_list(kmfh, cert, certlist, ncerts);
4359                 if (rv != KMF_OK)
4360                         return (rv);
4361         }

4363         /* Also add any included CA certs to the list */
4364         for (i = 0; sslcacerts != NULL && i < sk_X509_num(sslcacerts); i++) {
4365                 X509 *c;
4366                 /*
4367                  * sk_X509_value() is macro that embeds a cast to (X509 *).
4368                  * Here it translates into ((X509 *)sk_value((ca), (i))).
4369                  * Lint is complaining about the embedded casting, and
4370                  * to fix it, you need to fix openssl header files.
4371                  */
4372                 /* LINTED E_BAD_PTR_CAST_ALIGN */
4373 #endif /* ! codereview */
4374                 c = sk_X509_value(sslcacerts, i);

4376                 /* Now add the ca cert to the certlist */
4377                 rv = add_cert_to_list(kmfh, c, certlist, ncerts);
4378                 if (rv != KMF_OK)
4379                         return (rv);
4380         }
4381         return (rv);
4382 }

4384 KMF_RETURN
4385 openssl_import_objects(KMF_HANDLE *kmfh,
4386         char *filename, KMF_CREDENTIAL *cred,
4387         KMF_X509_DER_CERT **certlist, int *ncerts,
4388         KMF_RAW_KEY_DATA **keylist, int *nkeys)
4389 {
```



```
4390         KMF_RETURN      rv = KMF_OK;
4391         KMF_ENCODE_FORMAT format;
4392         BIO             *bio = NULL;
4393         STACK_OF(EVP_PKEY)      *privkeys = NULL;
4394         STACK_OF(X509)          *certs = NULL;
4395         STACK_OF(X509)          *cacerts = NULL;

4397         /*
4398          * auto-detect the file format, regardless of what
4399          * the 'format' parameters in the params say.
4400          */
4401         rv = kmf_get_file_format(filename, &format);
4402         if (rv != KMF_OK) {
4403                 return (rv);
4404         }

4406         /* This function only works for PEM or PKCS#12 files */
4407         if (format != KMF_FORMAT_PEM &&
4408             format != KMF_FORMAT_PEM_KEYPAIR &&
4409             format != KMF_FORMAT_PKCS12)
4410                 return (KMF_ERR_ENCODING);

4412         *certlist = NULL;
4413         *keylist = NULL;
4414         *ncerts = 0;
4415         *nkeys = 0;

4417         if (format == KMF_FORMAT_PKCS12) {
4418                 bio = BIO_new_file(filename, "rb");
4419                 if (bio == NULL) {
4420                         SET_ERROR(kmfh, ERR_get_error());
4421                         rv = KMF_ERR_OPEN_FILE;
4422                         goto end;
4423                 }

4425                 rv = extract_pkcs12(bio, (uchar_t *)cred->cred,
4426                     (uint32_t)cred->credlen, &privkeys, &certs, &cacerts);

4428                 if (rv  == KMF_OK)
4429                         /* Convert keys and certs to exportable format */
4430                         rv = convertPK12Objects(kmfh, privkeys, certs, cacerts,
4431                             keylist, nkeys, certlist, ncerts);
4432         } else {
4433                 EVP_PKEY *pkey;
4434                 KMF_DATA *certdata = NULL;
4435                 KMF_X509_DER_CERT *kmfcerts = NULL;
4436                 int i;
4437                 rv = extract_pem(kmfh, NULL, NULL, NULL, filename,
4438                     (uchar_t *)cred->cred, (uint32_t)cred->credlen,
4439                     &pkey, &certdata, ncerts);

4441                 /* Reached end of import file? */
4442                 if (rv == KMF_OK && pkey != NULL) {
4443                         privkeys = sk_EVP_PKEY_new_null();
4444                         if (privkeys == NULL) {
4445                                 rv = KMF_ERR_MEMORY;
4446                                 goto end;
4447                         }
4448                         (void) sk_EVP_PKEY_push(privkeys, pkey);
4449                         /* convert the certificate list here */
4450                         if (*ncerts > 0 && certlist != NULL) {
4451                                 kmfcerts = (KMF_X509_DER_CERT *)calloc(*ncerts,
4452                                     sizeof (KMF_X509_DER_CERT));
4453                                 if (kmfcerts == NULL) {
4454                                         rv = KMF_ERR_MEMORY;
4455                                         goto end;
```

```
4456                                }
4457                                for (i = 0; i < *ncerts; i++) {
4458                                        kmfcerts[i].certificate = certdata[i];
4459                                        kmfcerts[i].kmf_private.keystore_type =
4460                                            KMF_KEYSTORE_OPENSSL;
4461                                }
4462                                *certlist = kmfcerts;
4463                        }
4464                        /*
4465                         * Convert keys to exportable format, the certs
4466                         * are already OK.
4467                         */
4468                        rv = convertPK12Objects(kmfh, privkeys, NULL, NULL,
4469                            keylist, nkeys, NULL, NULL);
4470                }
4471        }
4472 end:
4473        if (bio != NULL)
4474                (void) BIO_free(bio);

4476        if (privkeys)
4477                sk_EVP_PKEY_pop_free(privkeys, EVP_PKEY_free);
4478        if (certs)
4479                sk_X509_pop_free(certs, X509_free);
4480        if (cacerts)
4481                sk_X509_pop_free(cacerts, X509_free);

4483        return (rv);
4484 }

4486 static KMF_RETURN
4487 create_deskey(DES_cblock **deskey)
4488 {
4489        DES_cblock *key;

4491        key = (DES_cblock *) malloc(sizeof (DES_cblock));
4492        if (key == NULL) {
4493                return (KMF_ERR_MEMORY);
4494        }

4496        if (DES_random_key(key) == 0) {
4497                free(key);
4498                return (KMF_ERR_KEYGEN_FAILED);
4499        }

4501        *deskey = key;
4502        return (KMF_OK);
4503 }

4505 #define KEYGEN_RETRY 3
4506 #define DES3_KEY_SIZE 24

4508 static KMF_RETURN
4509 create_des3key(unsigned char **des3key)
4510 {
4511        KMF_RETURN ret = KMF_OK;
4512        DES_cblock *deskey1 = NULL;
4513        DES_cblock *deskey2 = NULL;
4514        DES_cblock *deskey3 = NULL;
4515        unsigned char *newkey = NULL;
4516        int retry;

4518        if ((newkey = malloc(DES3_KEY_SIZE)) == NULL) {
4519                return (KMF_ERR_MEMORY);
4520        }
```



```
4522        /* create the 1st DES key */
4523        if ((ret = create_deskey(&deskey1)) != KMF_OK) {
4524                goto out;
4525        }

4527        /*
4528         * Create the 2nd DES key and make sure its value is different
4529         * from the 1st DES key.
4530         */
4531        retry = 0;
4532        do {
4533                if (deskey2 != NULL) {
4534                        free(deskey2);
4535                        deskey2 = NULL;
4536                }

4538                if ((ret = create_deskey(&deskey2)) != KMF_OK) {
4539                        goto out;
4540                }

4542                if (memcmp((const void *) deskey1, (const void *) deskey2, 8)
4543                    == 0) {
4544                        ret = KMF_ERR_KEYGEN_FAILED;
4545                        retry++;
4546                }
4547        } while (ret == KMF_ERR_KEYGEN_FAILED && retry < KEYGEN_RETRY);

4549        if (ret != KMF_OK) {
4550                goto out;
4551        }

4553        /*
4554         * Create the 3rd DES key and make sure its value is different
4555         * from the 2nd DES key.
4556         */
4557        retry = 0;
4558        do {
4559                if (deskey3 != NULL) {
4560                        free(deskey3);
4561                        deskey3 = NULL;
4562                }

4564                if ((ret = create_deskey(&deskey3)) != KMF_OK) {
4565                        goto out;
4566                }

4568                if (memcmp((const void *)deskey2, (const void *)deskey3, 8)
4569                    == 0) {
4570                        ret = KMF_ERR_KEYGEN_FAILED;
4571                        retry++;
4572                }
4573        } while (ret == KMF_ERR_KEYGEN_FAILED && retry < KEYGEN_RETRY);

4575        if (ret != KMF_OK) {
4576                goto out;
4577        }

4579        /* Concatenate 3 DES keys into a DES3 key */
4580        (void) memcpy((void *)newkey, (const void *)deskey1, 8);
4581        (void) memcpy((void *)(newkey + 8), (const void *)deskey2, 8);
4582        (void) memcpy((void *)(newkey + 16), (const void *)deskey3, 8);
4583        *des3key = newkey;

4585 out:
4586        if (deskey1 != NULL)
4587                free(deskey1);
```

```
4589         if (deskey2 != NULL)
4590                 free(deskey2);

4592         if (deskey3 != NULL)
4593                 free(deskey3);

4595         if (ret != KMF_OK && newkey != NULL)
4596                 free(newkey);

4598         return (ret);
4599 }

4601 KMF_RETURN
4602 OpenSSL_CreateSymKey(KMF_HANDLE_T handle,
4603         int numattr, KMF_ATTRIBUTE *attrlist)
4604 {
4605         KMF_RETURN ret = KMF_OK;
4606         KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
4607         char *fullpath = NULL;
4608         KMF_RAW_SYM_KEY *rkey = NULL;
4609         DES_cblock *deskey = NULL;
4610         unsigned char *des3key = NULL;
4611         unsigned char *random = NULL;
4612         int fd = -1;
4613         KMF_KEY_HANDLE *symkey;
4614         KMF_KEY_ALG keytype;
4615         uint32_t keylen;
4616         uint32_t keylen_size = sizeof (keylen);
4617         char *dirpath;
4618         char *keyfile;

4620         if (kmfh == NULL)
4621                 return (KMF_ERR_UNINITIALIZED);

4623         symkey = kmf_get_attr_ptr(KMF_KEY_HANDLE_ATTR, attrlist, numattr);
4624         if (symkey == NULL)
4625                 return (KMF_ERR_BAD_PARAMETER);

4627         dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);

4629         keyfile = kmf_get_attr_ptr(KMF_KEY_FILENAME_ATTR, attrlist, numattr);
4630         if (keyfile == NULL)
4631                 return (KMF_ERR_BAD_PARAMETER);

4633         ret = kmf_get_attr(KMF_KEYALG_ATTR, attrlist, numattr,
4634             (void *)&keytype, NULL);
4635         if (ret != KMF_OK)
4636                 return (KMF_ERR_BAD_PARAMETER);

4638         ret = kmf_get_attr(KMF_KEYLENGTH_ATTR, attrlist, numattr,
4639             &keylen, &keylen_size);
4640         if (ret == KMF_ERR_ATTR_NOT_FOUND &&
4641             (keytype == KMF_DES || keytype == KMF_DES3))
4642                 /* keylength is not required for DES and 3DES */
4643                 ret = KMF_OK;
4644         if (ret != KMF_OK)
4645                 return (KMF_ERR_BAD_PARAMETER);

4647         fullpath = get_fullpath(dirpath, keyfile);
4648         if (fullpath == NULL)
4649                 return (KMF_ERR_BAD_PARAMETER);

4651         /* If the requested file exists, return an error */
4652         if (test_for_file(fullpath, 0400) == 1) {
4653                 free(fullpath);
```



```
4654                 return (KMF_ERR_DUPLICATE_KEYFILE);
4655         }

4657         fd = open(fullpath, O_CREAT|O_TRUNC|O_RDWR, 0400);
4658         if (fd == -1) {
4659                 ret = KMF_ERR_OPEN_FILE;
4660                 goto out;
4661         }

4663         rkey = malloc(sizeof (KMF_RAW_SYM_KEY));
4664         if (rkey == NULL) {
4665                 ret = KMF_ERR_MEMORY;
4666                 goto out;
4667         }
4668         (void) memset(rkey, 0, sizeof (KMF_RAW_SYM_KEY));

4670         if (keytype == KMF_DES) {
4671                 if ((ret = create_deskey(&deskey)) != KMF_OK) {
4672                         goto out;
4673                 }
4674                 rkey->keydata.val = (uchar_t *)deskey;
4675                 rkey->keydata.len = 8;

4677                 symkey->keyalg = KMF_DES;

4679         } else if (keytype == KMF_DES3) {
4680                 if ((ret = create_des3key(&des3key)) != KMF_OK) {
4681                         goto out;
4682                 }
4683                 rkey->keydata.val = (uchar_t *)des3key;
4684                 rkey->keydata.len = DES3_KEY_SIZE;
4685                 symkey->keyalg = KMF_DES3;

4687         } else if (keytype == KMF_AES || keytype == KMF_RC4 ||
4688             keytype == KMF_GENERIC_SECRET) {
4689                 int bytes;

4691                 if (keylen % 8 != 0) {
4692                         ret = KMF_ERR_BAD_KEY_SIZE;
4693                         goto out;
4694                 }

4696                 if (keytype == KMF_AES) {
4697                         if (keylen != 128 &&
4698                             keylen != 192 &&
4699                             keylen != 256) {
4700                                 ret = KMF_ERR_BAD_KEY_SIZE;
4701                                 goto out;
4702                         }
4703                 }

4705                 bytes = keylen/8;
4706                 random = malloc(bytes);
4707                 if (random == NULL) {
4708                         ret = KMF_ERR_MEMORY;
4709                         goto out;
4710                 }
4711                 if (RAND_bytes(random, bytes) != 1) {
4712                         ret = KMF_ERR_KEYGEN_FAILED;
4713                         goto out;
4714                 }

4716                 rkey->keydata.val = (uchar_t *)random;
4717                 rkey->keydata.len = bytes;
4718                 symkey->keyalg = keytype;
```

```
4720         } else {
4721                 ret = KMF_ERR_BAD_KEY_TYPE;
4722                 goto out;
4723         }

4725         (void) write(fd, (const void *) rkey->keydata.val, rkey->keydata.len);

4727         symkey->kstype = KMF_KEYSTORE_OPENSSL;
4728         symkey->keyclass = KMF_SYMMETRIC;
4729         symkey->keylabel = (char *)fullpath;
4730         symkey->israw = TRUE;
4731         symkey->keyp = rkey;

4733 out:
4734         if (fd != -1)
4735                 (void) close(fd);

4737         if (ret != KMF_OK && fullpath != NULL) {
4738                 free(fullpath);
4739         }
4740         if (ret != KMF_OK) {
4741                 kmf_free_raw_sym_key(rkey);
4742                 symkey->keyp = NULL;
4743                 symkey->keyalg = KMF_KEYALG_NONE;
4744         }

4746         return (ret);
4747 }

4749 /*
4750  * Check a file to see if it is a CRL file with PEM or DER format.
4751  * If success, return its format in the "pformat" argument.
4752  */
4753 KMF_RETURN
4754 OpenSSL_IsCRLFile(KMF_HANDLE_T handle, char *filename, int *pformat)
4755 {
4756         KMF_RETURN      ret = KMF_OK;
4757         KMF_HANDLE      *kmfh = (KMF_HANDLE *)handle;
4758         BIO             *bio = NULL;
4759         X509_CRL        *xcrl = NULL;

4761         if (filename == NULL) {
4762                 return (KMF_ERR_BAD_PARAMETER);
4763         }

4765         bio = BIO_new_file(filename, "rb");
4766         if (bio == NULL)        {
4767                 SET_ERROR(kmfh, ERR_get_error());
4768                 ret = KMF_ERR_OPEN_FILE;
4769                 goto out;
4770         }

4772         if ((xcrl = PEM_read_bio_X509_CRL(bio, NULL, NULL, NULL)) != NULL) {
4773                 *pformat = KMF_FORMAT_PEM;
4774                 goto out;
4775         }
4776         (void) BIO_free(bio);

4778         /*
4779          * Now try to read it as raw DER data.
4780          */
4781         bio = BIO_new_file(filename, "rb");
4782         if (bio == NULL)        {
4783                 SET_ERROR(kmfh, ERR_get_error());
4784                 ret = KMF_ERR_OPEN_FILE;
4785                 goto out;
```



```
4786         }

4788         if ((xcrl = d2i_X509_CRL_bio(bio, NULL)) != NULL) {
4789                 *pformat = KMF_FORMAT_ASN1;
4790         } else {
4791                 ret = KMF_ERR_BAD_CRLFILE;
4792         }

4794 out:
4795         if (bio != NULL)
4796                 (void) BIO_free(bio);

4798         if (xcrl != NULL)
4799                 X509_CRL_free(xcrl);

4801         return (ret);
4802 }

4804 KMF_RETURN
4805 OpenSSL_GetSymKeyValue(KMF_HANDLE_T handle, KMF_KEY_HANDLE *symkey,
4806     KMF_RAW_SYM_KEY *rkey)
4807 {
4808         KMF_RETURN      rv = KMF_OK;
4809         KMF_HANDLE      *kmfh = (KMF_HANDLE *)handle;
4810         KMF_DATA        keyvalue;

4812         if (kmfh == NULL)
4813                 return (KMF_ERR_UNINITIALIZED);

4815         if (symkey == NULL || rkey == NULL)
4816                 return (KMF_ERR_BAD_PARAMETER);
4817         else if (symkey->keyclass != KMF_SYMMETRIC)
4818                 return (KMF_ERR_BAD_KEY_CLASS);

4820         if (symkey->israw) {
4821                 KMF_RAW_SYM_KEY *rawkey = (KMF_RAW_SYM_KEY *)symkey->keyp;

4823                 if (rawkey == NULL ||
4824                     rawkey->keydata.val == NULL ||
4825                     rawkey->keydata.len == 0)
4826                         return (KMF_ERR_BAD_KEYHANDLE);

4828                 rkey->keydata.len = rawkey->keydata.len;
4829                 if ((rkey->keydata.val = malloc(rkey->keydata.len)) == NULL)
4830                         return (KMF_ERR_MEMORY);
4831                 (void) memcpy(rkey->keydata.val, rawkey->keydata.val,
4832                     rkey->keydata.len);
4833         } else {
4834                 rv = kmf_read_input_file(handle, symkey->keylabel, &keyvalue);
4835                 if (rv != KMF_OK)
4836                         return (rv);
4837                 rkey->keydata.len = keyvalue.Length;
4838                 rkey->keydata.val = keyvalue.Data;
4839         }

4841         return (rv);
4842 }

4844 /*
4845  * substitute for the unsafe access(2) function.
4846  * If the file in question already exists, return 1.
4847  * else 0.  If an error occurs during testing (other
4848  * than EEXIST), return -1.
4849  */
4850 static int
4851 test_for_file(char *filename, mode_t mode)
```

```
{
        int fd;

        /*
         * Try to create the file with the EXCL flag.
         * The call should fail if the file exists.
         */
        fd = open(filename, O_WRONLY|O_CREAT|O_EXCL, mode);
        if (fd == -1 && errno == EEXIST)
                return (1);
        else if (fd == -1) /* some other error */
                return (-1);

        /* The file did NOT exist.  Delete the testcase. */
        (void) close(fd);
        (void) unlink(filename);
        return (0);
}

KMF_RETURN
OpenSSL_StoreKey(KMF_HANDLE_T handle, int numattr,
        KMF_ATTRIBUTE *attrlist)
{
        KMF_RETURN rv = KMF_OK;
        KMF_HANDLE      *kmfh = (KMF_HANDLE *)handle;
        KMF_KEY_HANDLE *pubkey = NULL, *prikey = NULL;
        KMF_RAW_KEY_DATA *rawkey;
        EVP_PKEY *pkey = NULL;
        KMF_ENCODE_FORMAT format = KMF_FORMAT_PEM;
        KMF_CREDENTIAL cred = {NULL, 0};
        BIO *out = NULL;
        int keys = 0;
        char *fullpath = NULL;
        char *keyfile = NULL;
        char *dirpath = NULL;

        pubkey = kmf_get_attr_ptr(KMF_PUBKEY_HANDLE_ATTR, attrlist, numattr);
        if (pubkey != NULL)
                keys++;

        prikey = kmf_get_attr_ptr(KMF_PRIVKEY_HANDLE_ATTR, attrlist, numattr);
        if (prikey != NULL)
                keys++;

        rawkey = kmf_get_attr_ptr(KMF_RAW_KEY_ATTR, attrlist, numattr);
        if (rawkey != NULL)
                keys++;

        /*
         * Exactly 1 type of key must be passed to this function.
         */
        if (keys != 1)
                return (KMF_ERR_BAD_PARAMETER);

        keyfile = (char *)kmf_get_attr_ptr(KMF_KEY_FILENAME_ATTR, attrlist,
            numattr);
        if (keyfile == NULL)
                return (KMF_ERR_BAD_PARAMETER);

        dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);

        fullpath = get_fullpath(dirpath, keyfile);

        /* Once we have the full path, we don't need the pieces */
        if (fullpath == NULL)
                return (KMF_ERR_BAD_PARAMETER);
```



```
        /* If the requested file exists, return an error */
        if (test_for_file(fullpath, 0400) == 1) {
                free(fullpath);
                return (KMF_ERR_DUPLICATE_KEYFILE);
        }

        rv = kmf_get_attr(KMF_ENCODE_FORMAT_ATTR, attrlist, numattr,
            &format, NULL);
        if (rv != KMF_OK)
                /* format is optional. */
                rv = KMF_OK;

        /* CRED is not required for OpenSSL files */
        (void) kmf_get_attr(KMF_CREDENTIAL_ATTR, attrlist, numattr,
            &cred, NULL);

        /* Store the private key to the keyfile */
        out = BIO_new_file(fullpath, "wb");
        if (out == NULL) {
                SET_ERROR(kmfh, ERR_get_error());
                rv = KMF_ERR_OPEN_FILE;
                goto end;
        }

        if (prikey != NULL && prikey->keyp != NULL) {
                if (prikey->keyalg == KMF_RSA ||
                    prikey->keyalg == KMF_DSA) {
                        pkey = (EVP_PKEY *)prikey->keyp;

                        rv = ssl_write_key(kmfh, format,
                            out, &cred, pkey, TRUE);

                        if (rv == KMF_OK && prikey->keylabel == NULL) {
                                prikey->keylabel = strdup(fullpath);
                                if (prikey->keylabel == NULL)
                                        rv = KMF_ERR_MEMORY;
                        }
                }
        } else if (pubkey != NULL && pubkey->keyp != NULL) {
                if (pubkey->keyalg == KMF_RSA ||
                    pubkey->keyalg == KMF_DSA) {
                        pkey = (EVP_PKEY *)pubkey->keyp;

                        rv = ssl_write_key(kmfh, format,
                            out, &cred, pkey, FALSE);

                        if (rv == KMF_OK && pubkey->keylabel == NULL) {
                                pubkey->keylabel = strdup(fullpath);
                                if (pubkey->keylabel == NULL)
                                        rv = KMF_ERR_MEMORY;
                        }
                }
        } else if (rawkey != NULL) {
                if (rawkey->keytype == KMF_RSA) {
                        pkey = ImportRawRSAKey(&rawkey->rawdata.rsa);
                } else if (rawkey->keytype == KMF_DSA) {
                        pkey = ImportRawDSAKey(&rawkey->rawdata.dsa);
                } else {
                        rv = KMF_ERR_BAD_PARAMETER;
                }
                if (pkey != NULL) {
                        KMF_KEY_CLASS kclass = KMF_ASYM_PRI;

                        rv = kmf_get_attr(KMF_KEYCLASS_ATTR, attrlist, numattr,
                            (void *)&kclass, NULL);
```

```
4984                        if (rv != KMF_OK)
4985                                rv = KMF_OK;
4986                        rv = ssl_write_key(kmfh, format, out,
4987                            &cred, pkey, (kclass == KMF_ASYM_PRI));
4988                        EVP_PKEY_free(pkey);
4989                }
4990        }

4992 end:

4994        if (out)
4995                (void) BIO_free(out);


4998        if (rv == KMF_OK)
4999                (void) chmod(fullpath, 0400);

5001        free(fullpath);
5002        return (rv);
5003 }

5005 KMF_RETURN
5006 OpenSSL_ImportCRL(KMF_HANDLE_T handle, int numattr, KMF_ATTRIBUTE *attrlist)
5007 {
5008        KMF_RETURN ret = KMF_OK;
5009        KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
5010        X509_CRL *xcrl = NULL;
5011        X509 *xcert = NULL;
5012        EVP_PKEY *pkey;
5013        KMF_ENCODE_FORMAT format;
5014        BIO *in = NULL, *out = NULL;
5015        int openssl_ret = 0;
5016        KMF_ENCODE_FORMAT outformat;
5017        boolean_t crlcheck = FALSE;
5018        char *certfile, *dirpath, *crlfile, *incrl, *outcrl, *outcrlfile;

5020        if (numattr == 0 || attrlist == NULL) {
5021                return (KMF_ERR_BAD_PARAMETER);
5022        }

5024        /* CRL check is optional */
5025        (void) kmf_get_attr(KMF_CRL_CHECK_ATTR, attrlist, numattr,
5026            &crlcheck, NULL);

5028        certfile = kmf_get_attr_ptr(KMF_CERT_FILENAME_ATTR, attrlist, numattr);
5029        if (crlcheck == B_TRUE && certfile == NULL) {
5030                return (KMF_ERR_BAD_CERTFILE);
5031        }

5033        dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);
5034        incrl = kmf_get_attr_ptr(KMF_CRL_FILENAME_ATTR, attrlist, numattr);
5035        outcrl = kmf_get_attr_ptr(KMF_CRL_OUTFILE_ATTR, attrlist, numattr);

5037        crlfile = get_fullpath(dirpath, incrl);

5039        if (crlfile == NULL)
5040                return (KMF_ERR_BAD_CRLFILE);

5042        outcrlfile = get_fullpath(dirpath, outcrl);
5043        if (outcrlfile == NULL)
5044                return (KMF_ERR_BAD_CRLFILE);

5046        if (isdir(outcrlfile)) {
5047                free(outcrlfile);
5048                return (KMF_ERR_BAD_CRLFILE);
5049        }
```



```
5051        ret = kmf_is_crl_file(handle, crlfile, &format);
5052        if (ret != KMF_OK) {
5053                free(outcrlfile);
5054                return (ret);
5055        }

5057        in = BIO_new_file(crlfile, "rb");
5058        if (in == NULL) {
5059                SET_ERROR(kmfh, ERR_get_error());
5060                ret = KMF_ERR_OPEN_FILE;
5061                goto end;
5062        }

5064        if (format == KMF_FORMAT_ASN1) {
5065                xcrl = d2i_X509_CRL_bio(in, NULL);
5066        } else if (format == KMF_FORMAT_PEM) {
5067                xcrl = PEM_read_bio_X509_CRL(in, NULL, NULL, NULL);
5068        }

5070        if (xcrl == NULL) {
5071                SET_ERROR(kmfh, ERR_get_error());
5072                ret = KMF_ERR_BAD_CRLFILE;
5073                goto end;
5074        }

5076        /* If bypasscheck is specified, no need to verify. */
5077        if (crlcheck == B_FALSE)
5078                goto output;

5080        ret = kmf_is_cert_file(handle, certfile, &format);
5081        if (ret != KMF_OK)
5082                goto end;

5084        /* Read in the CA cert file and convert to X509 */
5085        if (BIO_read_filename(in, certfile) <= 0) {
5086                SET_ERROR(kmfh, ERR_get_error());
5087                ret = KMF_ERR_OPEN_FILE;
5088                goto end;
5089        }

5091        if (format == KMF_FORMAT_ASN1) {
5092                xcert = d2i_X509_bio(in, NULL);
5093        } else if (format == KMF_FORMAT_PEM) {
5094                xcert = PEM_read_bio_X509(in, NULL, NULL, NULL);
5095        } else {
5096                ret = KMF_ERR_BAD_CERT_FORMAT;
5097                goto end;
5098        }

5100        if (xcert == NULL) {
5101                SET_ERROR(kmfh, ERR_get_error());
5102                ret = KMF_ERR_BAD_CERT_FORMAT;
5103                goto end;
5104        }
5105        /* Now get the public key from the CA cert */
5106        pkey = X509_get_pubkey(xcert);
5107        if (pkey == NULL) {
5108                SET_ERROR(kmfh, ERR_get_error());
5109                ret = KMF_ERR_BAD_CERTFILE;
5110                goto end;
5111        }

5113        /* Verify the CRL with the CA's public key */
5114        openssl_ret = X509_CRL_verify(xcrl, pkey);
5115        EVP_PKEY_free(pkey);
```

```
5116         if (openssl_ret > 0) {
5117                 ret = KMF_OK;  /* verify succeed */
5118         } else {
5119                 SET_ERROR(kmfh, openssl_ret);
5120                 ret = KMF_ERR_BAD_CRLFILE;
5121         }

5123 output:
5124         ret = kmf_get_attr(KMF_ENCODE_FORMAT_ATTR, attrlist, numattr,
5125             &outformat, NULL);
5126         if (ret != KMF_OK) {
5127                 ret = KMF_OK;
5128                 outformat = KMF_FORMAT_PEM;
5129         }

5131         out = BIO_new_file(outcrlfile, "wb");
5132         if (out == NULL) {
5133                 SET_ERROR(kmfh, ERR_get_error());
5134                 ret = KMF_ERR_OPEN_FILE;
5135                 goto end;
5136         }

5138         if (outformat == KMF_FORMAT_ASN1) {
5139                 openssl_ret = (int)i2d_X509_CRL_bio(out, xcrl);
5140         } else if (outformat == KMF_FORMAT_PEM) {
5141                 openssl_ret = PEM_write_bio_X509_CRL(out, xcrl);
5142         } else {
5143                 ret = KMF_ERR_BAD_PARAMETER;
5144                 goto end;
5145         }

5147         if (openssl_ret <= 0) {
5148                 SET_ERROR(kmfh, ERR_get_error());
5149                 ret = KMF_ERR_WRITE_FILE;
5150         } else {
5151                 ret = KMF_OK;
5152         }

5154 end:
5155         if (xcrl != NULL)
5156                 X509_CRL_free(xcrl);

5158         if (xcert != NULL)
5159                 X509_free(xcert);

5161         if (in != NULL)
5162                 (void) BIO_free(in);

5164         if (out != NULL)
5165                 (void) BIO_free(out);

5167         if (outcrlfile != NULL)
5168                 free(outcrlfile);

5170         return (ret);
5171 }

5173 KMF_RETURN
5174 OpenSSL_ListCRL(KMF_HANDLE_T handle, int numattr, KMF_ATTRIBUTE *attrlist)
5175 {
5176         KMF_RETURN ret = KMF_OK;
5177         KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
5178         X509_CRL   *x = NULL;
5179         KMF_ENCODE_FORMAT format;
5180         char *crlfile = NULL;
5181         BIO *in = NULL;
```



```
5182         BIO *mem = NULL;
5183         long len;
5184         char *memptr;
5185         char *data = NULL;
5186         char **crldata;
5187         char *crlfilename, *dirpath;

5189         if (numattr == 0 || attrlist == NULL) {
5190                 return (KMF_ERR_BAD_PARAMETER);
5191         }
5192         crlfilename = kmf_get_attr_ptr(KMF_CRL_FILENAME_ATTR,
5193             attrlist, numattr);
5194         if (crlfilename == NULL)
5195                 return (KMF_ERR_BAD_CRLFILE);

5197         crldata = (char **)kmf_get_attr_ptr(KMF_CRL_DATA_ATTR,
5198             attrlist, numattr);

5200         if (crldata == NULL)
5201                 return (KMF_ERR_BAD_PARAMETER);

5203         dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);

5205         crlfile = get_fullpath(dirpath, crlfilename);

5207         if (crlfile == NULL)
5208                 return (KMF_ERR_BAD_CRLFILE);

5210         if (isdir(crlfile)) {
5211                 free(crlfile);
5212                 return (KMF_ERR_BAD_CRLFILE);
5213         }

5215         ret = kmf_is_crl_file(handle, crlfile, &format);
5216         if (ret != KMF_OK) {
5217                 free(crlfile);
5218                 return (ret);
5219         }

5221         if (bio_err == NULL)
5222                 bio_err = BIO_new_fp(stderr, BIO_NOCLOSE);

5224         in = BIO_new_file(crlfile, "rb");
5225         if (in == NULL) {
5226                 SET_ERROR(kmfh, ERR_get_error());
5227                 ret = KMF_ERR_OPEN_FILE;
5228                 goto end;
5229         }

5231         if (format == KMF_FORMAT_ASN1) {
5232                 x = d2i_X509_CRL_bio(in, NULL);
5233         } else if (format == KMF_FORMAT_PEM) {
5234                 x = PEM_read_bio_X509_CRL(in, NULL, NULL, NULL);
5235         }

5237         if (x == NULL) { /* should not happen */
5238                 SET_ERROR(kmfh, ERR_get_error());
5239                 ret = KMF_ERR_OPEN_FILE;
5240                 goto end;
5241         }

5243         mem = BIO_new(BIO_s_mem());
5244         if (mem == NULL) {
5245                 SET_ERROR(kmfh, ERR_get_error());
5246                 ret = KMF_ERR_MEMORY;
5247                 goto end;
```

```
5248         }

5250         (void) X509_CRL_print(mem, x);
5251         len = BIO_get_mem_data(mem, &memptr);
5252         if (len <= 0) {
5253                 SET_ERROR(kmfh, ERR_get_error());
5254                 ret = KMF_ERR_MEMORY;
5255                 goto end;
5256         }

5258         data = malloc(len + 1);
5259         if (data == NULL) {
5260                 ret = KMF_ERR_MEMORY;
5261                 goto end;
5262         }

5264         (void) memcpy(data, memptr, len);
5265         data[len] = '\0';
5266         *crldata = data;

5268 end:
5269         if (x != NULL)
5270                 X509_CRL_free(x);

5272         if (crlfile != NULL)
5273                 free(crlfile);

5275         if (in != NULL)
5276                 (void) BIO_free(in);

5278         if (mem != NULL)
5279                 (void) BIO_free(mem);

5281         return (ret);
5282 }

5284 KMF_RETURN
5285 OpenSSL_DeleteCRL(KMF_HANDLE_T handle, int numattr, KMF_ATTRIBUTE *attrlist)
5286 {
5287         KMF_RETURN ret = KMF_OK;
5288         KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
5289         KMF_ENCODE_FORMAT format;
5290         char *crlfile = NULL;
5291         BIO *in = NULL;
5292         char *crlfilename, *dirpath;

5294         if (numattr == 0 || attrlist == NULL) {
5295                 return (KMF_ERR_BAD_PARAMETER);
5296         }

5298         crlfilename = kmf_get_attr_ptr(KMF_CRL_FILENAME_ATTR,
5299             attrlist, numattr);

5301         if (crlfilename == NULL)
5302                 return (KMF_ERR_BAD_CRLFILE);

5304         dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);

5306         crlfile = get_fullpath(dirpath, crlfilename);

5308         if (crlfile == NULL)
5309                 return (KMF_ERR_BAD_CRLFILE);

5311         if (isdir(crlfile)) {
5312                 ret = KMF_ERR_BAD_CRLFILE;
5313                 goto end;
```



```
5314         }

5316         ret = kmf_is_crl_file(handle, crlfile, &format);
5317         if (ret != KMF_OK)
5318                 goto end;

5320         if (unlink(crlfile) != 0) {
5321                 SET_SYS_ERROR(kmfh, errno);
5322                 ret = KMF_ERR_INTERNAL;
5323                 goto end;
5324         }

5326 end:
5327         if (in != NULL)
5328                 (void) BIO_free(in);
5329         if (crlfile != NULL)
5330                 free(crlfile);

5332         return (ret);
5333 }

5335 KMF_RETURN
5336 OpenSSL_FindCertInCRL(KMF_HANDLE_T handle, int numattr, KMF_ATTRIBUTE *attrlist)
5337 {
5338         KMF_RETURN ret = KMF_OK;
5339         KMF_HANDLE *kmfh = (KMF_HANDLE *)handle;
5340         KMF_ENCODE_FORMAT format;
5341         BIO *in = NULL;
5342         X509   *xcert = NULL;
5343         X509_CRL   *xcrl = NULL;
5344         STACK_OF(X509_REVOKED) *revoke_stack = NULL;
5345         X509_REVOKED *revoke;
5346         int i;
5347         char *crlfilename, *crlfile, *dirpath, *certfile;

5349         if (numattr == 0 || attrlist == NULL) {
5350                 return (KMF_ERR_BAD_PARAMETER);
5351         }

5353         crlfilename = kmf_get_attr_ptr(KMF_CRL_FILENAME_ATTR,
5354             attrlist, numattr);

5356         if (crlfilename == NULL)
5357                 return (KMF_ERR_BAD_CRLFILE);

5359         certfile = kmf_get_attr_ptr(KMF_CERT_FILENAME_ATTR, attrlist, numattr);
5360         if (certfile == NULL)
5361                 return (KMF_ERR_BAD_CRLFILE);

5363         dirpath = kmf_get_attr_ptr(KMF_DIRPATH_ATTR, attrlist, numattr);

5365         crlfile = get_fullpath(dirpath, crlfilename);

5367         if (crlfile == NULL)
5368                 return (KMF_ERR_BAD_CRLFILE);

5370         if (isdir(crlfile)) {
5371                 ret = KMF_ERR_BAD_CRLFILE;
5372                 goto end;
5373         }

5375         ret = kmf_is_crl_file(handle, crlfile, &format);
5376         if (ret != KMF_OK)
5377                 goto end;

5379         /* Read the CRL file and load it into a X509_CRL structure */
```

```
5380         in = BIO_new_file(crlfilename, "rb");
5381         if (in == NULL) {
5382                 SET_ERROR(kmfh, ERR_get_error());
5383                 ret = KMF_ERR_OPEN_FILE;
5384                 goto end;
5385         }

5387         if (format == KMF_FORMAT_ASN1) {
5388                 xcrl = d2i_X509_CRL_bio(in, NULL);
5389         } else if (format == KMF_FORMAT_PEM) {
5390                 xcrl = PEM_read_bio_X509_CRL(in, NULL, NULL, NULL);
5391         }

5393         if (xcrl == NULL) {
5394                 SET_ERROR(kmfh, ERR_get_error());
5395                 ret = KMF_ERR_BAD_CRLFILE;
5396                 goto end;
5397         }
5398         (void) BIO_free(in);

5400         /* Read the Certificate file and load it into a X509 structure */
5401         ret = kmf_is_cert_file(handle, certfile, &format);
5402         if (ret != KMF_OK)
5403                 goto end;

5405         in = BIO_new_file(certfile, "rb");
5406         if (in == NULL) {
5407                 SET_ERROR(kmfh, ERR_get_error());
5408                 ret = KMF_ERR_OPEN_FILE;
5409                 goto end;
5410         }

5412         if (format == KMF_FORMAT_ASN1) {
5413                 xcert = d2i_X509_bio(in, NULL);
5414         } else if (format == KMF_FORMAT_PEM) {
5415                 xcert = PEM_read_bio_X509(in, NULL, NULL, NULL);
5416         }

5418         if (xcert == NULL) {
5419                 SET_ERROR(kmfh, ERR_get_error());
5420                 ret = KMF_ERR_BAD_CERTFILE;
5421                 goto end;
5422         }

5424         /* Check if the certificate and the CRL have same issuer */
5425         if (X509_NAME_cmp(xcert->cert_info->issuer, xcrl->crl->issuer) != 0) {
5426                 ret = KMF_ERR_ISSUER;
5427                 goto end;
5428         }

5430         /* Check to see if the certificate serial number is revoked */
5431         revoke_stack = X509_CRL_get_REVOKED(xcrl);
5432         if (sk_X509_REVOKED_num(revoke_stack) <= 0) {
5433                 /* No revoked certificates in the CRL file */
5434                 SET_ERROR(kmfh, ERR_get_error());
5435                 ret = KMF_ERR_EMPTY_CRL;
5436                 goto end;
5437         }

5439         for (i = 0; i < sk_X509_REVOKED_num(revoke_stack); i++) {
5440                 /* LINTED E_BAD_PTR_CAST_ALIGN */
5441 #endif /* ! codereview */
5442                 revoke = sk_X509_REVOKED_value(revoke_stack, i);
5443                 if (ASN1_INTEGER_cmp(xcert->cert_info->serialNumber,
5444                     revoke->serialNumber) == 0) {
5445                         break;
```



```
5446                 }
5447         }

5449         if (i < sk_X509_REVOKED_num(revoke_stack)) {
5450                 ret = KMF_OK;
5451         } else {
5452                 ret = KMF_ERR_NOT_REVOKED;
5453         }

5455 end:
5456         if (in != NULL)
5457                 (void) BIO_free(in);
5458         if (xcrl != NULL)
5459                 X509_CRL_free(xcrl);
5460         if (xcert != NULL)
5461                 X509_free(xcert);

5463         return (ret);
5464 }

5466 KMF_RETURN
5467 OpenSSL_VerifyCRLFile(KMF_HANDLE_T handle, char *crlname, KMF_DATA *tacert)
5468 {
5469         KMF_RETURN      ret = KMF_OK;
5470         KMF_HANDLE      *kmfh = (KMF_HANDLE *)handle;
5471         BIO             *bcrl = NULL;
5472         X509_CRL        *xcrl = NULL;
5473         X509            *xcert = NULL;
5474         EVP_PKEY        *pkey;
5475         int             sslret;
5476         KMF_ENCODE_FORMAT crl_format;
5477         unsigned char   *p;
5478         long            len;

5480         if (handle == NULL || crlname == NULL || tacert == NULL) {
5481                 return (KMF_ERR_BAD_PARAMETER);
5482         }

5484         ret = kmf_get_file_format(crlname, &crl_format);
5485         if (ret != KMF_OK)
5486                 return (ret);

5488         bcrl = BIO_new_file(crlname, "rb");
5489         if (bcrl == NULL)       {
5490                 SET_ERROR(kmfh, ERR_get_error());
5491                 ret = KMF_ERR_OPEN_FILE;
5492                 goto cleanup;
5493         }

5495         if (crl_format == KMF_FORMAT_ASN1) {
5496                 xcrl = d2i_X509_CRL_bio(bcrl, NULL);
5497         } else if (crl_format == KMF_FORMAT_PEM) {
5498                 xcrl = PEM_read_bio_X509_CRL(bcrl, NULL, NULL, NULL);
5499         } else {
5500                 ret = KMF_ERR_BAD_PARAMETER;
5501                 goto cleanup;
5502         }

5504         if (xcrl == NULL) {
5505                 SET_ERROR(kmfh, ERR_get_error());
5506                 ret = KMF_ERR_BAD_CRLFILE;
5507                 goto cleanup;
5508         }

5510         p = tacert->Data;
5511         len = tacert->Length;
```

```
5512         xcert = d2i_X509(NULL, (const uchar_t **)&p, len);

5514         if (xcert == NULL) {
5515                 SET_ERROR(kmfh, ERR_get_error());
5516                 ret = KMF_ERR_BAD_CERTFILE;
5517                 goto cleanup;
5518         }

5520         /* Get issuer certificate public key */
5521         pkey = X509_get_pubkey(xcert);
5522         if (pkey == NULL) {
5523                 SET_ERROR(kmfh, ERR_get_error());
5524                 ret = KMF_ERR_BAD_CERT_FORMAT;
5525                 goto cleanup;
5526         }

5528         /* Verify CRL signature */
5529         sslret = X509_CRL_verify(xcrl, pkey);
5530         EVP_PKEY_free(pkey);
5531         if (sslret > 0) {
5532                 ret = KMF_OK;
5533         } else {
5534                 SET_ERROR(kmfh, sslret);
5535                 ret = KMF_ERR_BAD_CRLFILE;
5536         }

5538 cleanup:
5539         if (bcrl != NULL)
5540                 (void) BIO_free(bcrl);

5542         if (xcrl != NULL)
5543                 X509_CRL_free(xcrl);

5545         if (xcert != NULL)
5546                 X509_free(xcert);

5548         return (ret);

5550 }

5552 KMF_RETURN
5553 OpenSSL_CheckCRLDate(KMF_HANDLE_T handle, char *crlname)
5554 {
5555         KMF_RETURN      ret = KMF_OK;
5556         KMF_HANDLE      *kmfh = (KMF_HANDLE *)handle;
5557         KMF_ENCODE_FORMAT crl_format;
5558         BIO             *bcrl = NULL;
5559         X509_CRL        *xcrl = NULL;
5560         int             i;

5562         if (handle == NULL || crlname == NULL) {
5563                 return (KMF_ERR_BAD_PARAMETER);
5564         }

5566         ret = kmf_is_crl_file(handle, crlname, &crl_format);
5567         if (ret != KMF_OK)
5568                 return (ret);

5570         bcrl = BIO_new_file(crlname, "rb");
5571         if (bcrl == NULL) {
5572                 SET_ERROR(kmfh, ERR_get_error());
5573                 ret = KMF_ERR_OPEN_FILE;
5574                 goto cleanup;
5575         }

5577         if (crl_format == KMF_FORMAT_ASN1)
```



```
5578                 xcrl = d2i_X509_CRL_bio(bcrl, NULL);
5579         else if (crl_format == KMF_FORMAT_PEM)
5580                 xcrl = PEM_read_bio_X509_CRL(bcrl, NULL, NULL, NULL);

5582         if (xcrl == NULL) {
5583                 SET_ERROR(kmfh, ERR_get_error());
5584                 ret = KMF_ERR_BAD_CRLFILE;
5585                 goto cleanup;
5586         }
5587         i = X509_cmp_time(X509_CRL_get_lastUpdate(xcrl), NULL);
5588         if (i >= 0) {
5589                 ret = KMF_ERR_VALIDITY_PERIOD;
5590                 goto cleanup;
5591         }
5592         if (X509_CRL_get_nextUpdate(xcrl)) {
5593                 i = X509_cmp_time(X509_CRL_get_nextUpdate(xcrl), NULL);

5595                 if (i <= 0) {
5596                         ret = KMF_ERR_VALIDITY_PERIOD;
5597                         goto cleanup;
5598                 }
5599         }

5601         ret = KMF_OK;

5603 cleanup:
5604         if (bcrl != NULL)
5605                 (void) BIO_free(bcrl);

5607         if (xcrl != NULL)
5608                 X509_CRL_free(xcrl);

5610         return (ret);
5611 }
```

```
**********************************************************
    2572 Tue May 20 20:20:11 2014
new/usr/src/lib/libpkg/Makefile.com
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

LIBRARY= libpkg.a
VERS=   .1

# include library definitions
OBJECTS=        \
                canonize.o   ckparam.o    ckvolseq.o \
                devtype.o    dstream.o    gpkglist.o \
                gpkgmap.o    isdir.o      logerr.o \
                mappath.o    ncgrpw.o     nhash.o \
                pkgexecl.o   pkgexecv.o   pkgmount.o \
                pkgtrans.o   ppkgmap.o \
                progerr.o    putcfile.o   rrmdir.o \
                runcmd.o     srchcfile.o  tputcfent.o \
                verify.o     security.o   pkgweb.o \
                pkgerr.o     keystore.o   p12lib.o \
                vfpops.o     fmkdir.o     pkgstr.o \
                handlelocalfs.o pkgserv.o


# include library definitions
include $(SRC)/lib/Makefile.lib

SRCDIR=         ../common

POFILE =        libpkg.po
MSGFILES =      $(OBJECTS:%.o=../common/%.i)
CLEANFILES +=   $(MSGFILES)

# This library is NOT lint clean

# openssl forces us to ignore dubious pointer casts, thanks to its clever
# use of macros for stack management.
LINTFLAGS=      -umx -errtags \
                -erroff=E_BAD_PTR_CAST_ALIGN,E_BAD_PTR_CAST,E_SUPPRESSION_DIRECT
                -erroff=E_BAD_PTR_CAST_ALIGN,E_BAD_PTR_CAST
```



```
$(LINTLIB):=    SRCS = $(SRCDIR)/$(LINTSRC)


LIBS = $(DYNLIB) $(LINTLIB)


LDLIBS +=       -lc -lwanboot -lscf -ladm

# libcrypto and libssl have no lint library, and so can only be used when
# building
$(DYNLIB) := LDLIBS += -lcrypto -lssl

CFLAGS +=       $(CCVERBOSE)
CERRWARN +=     -_gcc=-Wno-unused-label
CERRWARN +=     -_gcc=-Wno-parentheses
CERRWARN +=     -_gcc=-Wno-uninitialized
CERRWARN +=     -_gcc=-Wno-clobbered
CERRWARN +=     -_gcc=-Wno-switch
CERRWARN +=     -_gcc=-Wno-unused-value
CPPFLAGS +=     -I$(SRCDIR) -D_FILE_OFFSET_BITS=64

.KEEP_STATE:

all:    $(LIBS)

$(POFILE): $(MSGFILES)
        $(BUILDPO.msgfiles)

_msg: $(MSGDOMAINPOFILE)

lint: lintcheck

# include library targets
include $(SRC)/lib/Makefile.targ
include $(SRC)/Makefile.msg.targ
```

```
**********************************************************
    2932 Tue May 20 20:20:12 2014
new/usr/src/lib/libwanboot/Makefile.com
4853 illumos-gate is not lint-clean when built with openssl 1.0 (fix openssl 0.9
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# Copyright (c) 2012 by Delphix. All rights reserved.
#

LIBRARY =       libwanboot.a
VERS =          .1

# List of locally located modules.
LOC_DIR =       ../common
LOC_OBJS =      socket_inet.o bootinfo_aux.o
LOC_SRCS =      $(LOC_OBJS:%.o=$(LOC_DIR)/%.c)

# List of common wanboot objects.
COM_DIR =       ../../../common/net/wanboot
COM_OBJS =      auxutil.o \
                boot_http.o \
                bootconf.o \
                bootconf_errmsg.o \
                bootinfo.o \
                bootlog.o \
                http_errorstr.o \
                p12access.o \
                p12auxpars.o \
                p12auxutl.o \
                p12err.o \
                p12misc.o \
                parseURL.o
COM_SRCS =      $(COM_OBJS:%.o=$(COM_DIR)/%.c)

# List of common DHCP modules.
DHCP_DIR =      $(SRC)/common/net/dhcp
DHCP_OBJS =     dhcpinfo.o
DHCP_SRCS =     $(DHCP_OBJS:%.o=$(DHCP_DIR)/%.c)

OBJECTS =       $(LOC_OBJS) $(COM_OBJS) $(DHCP_OBJS)

include ../../Makefile.lib

LIBS +=         $(LINTLIB)
```



```
LDLIBS +=       -lnvpair -lresolv -lnsl -lsocket -ldevinfo -ldhcputil \
                -linetutil -lc

# libcrypto and libssl have no lint library, so we can only use it when
# building
$(DYNLIB) := LDLIBS += -lcrypto -lssl

CPPFLAGS =      -I$(SRC)/common/net/wanboot/crypt $(CPPFLAGS.master)
CERRWARN +=     -_gcc=-Wno-switch
CERRWARN +=     -_gcc=-Wno-parentheses
CERRWARN +=     -_gcc=-Wno-unused-value
CERRWARN +=     -_gcc=-Wno-uninitialized

# Must override SRCS from Makefile.lib since sources have
# multiple source directories.
SRCS =          $(LOC_SRCS) $(COM_SRCS) $(DHCP_SRCS)

# Must define location of lint library source.
SRCDIR =        $(LOC_DIR)
$(LINTLIB) :=   SRCS = $(SRCDIR)/$(LINTSRC)

# OpenSSL requires us to turn this off
LINTFLAGS   +=    -erroff=E_BAD_PTR_CAST_ALIGN
LINTFLAGS64 +=    -erroff=E_BAD_PTR_CAST_ALIGN

# OpenSSL 1.0 and 0.9.8 produce different lint warnings
LINTFLAGS   +=    -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED
LINTFLAGS64 +=    -erroff=E_SUPPRESSION_DIRECTIVE_UNUSED

#endif /* ! codereview */
CFLAGS +=       $(CCVERBOSE)
CPPFLAGS +=     -I$(LOC_DIR) -I$(COM_DIR) -I$(DHCP_DIR)

.KEEP_STATE:

all: $(LIBS)

lint: lintcheck

pics/%.o: $(COM_DIR)/%.c
        $(COMPILE.c) -o $@ $<
        $(POST_PROCESS_O)

pics/%.o: $(DHCP_DIR)/%.c
        $(COMPILE.c) -o $@ $<
        $(POST_PROCESS_O)

include ../../Makefile.targ
```