```
**********************************************************
   34920 Tue Oct 28 16:45:20 2008
new/usr/src/cmd/cmd-crypto/cryptoadm/adm_kef.c
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <locale.h>
#include <libgen.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/crypto/ioctladmin.h>
#include <signal.h>
#include <sys/crypto/elfsign.h>
#include "cryptoadm.h"

static int err; /* to store the value of errno in case being overwritten */
static int check_hardware_provider(char *, char *, int *, int *);

/*
 * Display the mechanism list for a kernel software provider.
 * This implements part of the "cryptoadm list -m" command.
 *
 * Parameters phardlist and psoftlist are supplied by get_kcfconf_info().
 * If NULL, this function obtains it by calling get_kcfconf_info() internally.
 */
int
list_mechlist_for_soft(char *provname,
    entrylist_t *phardlist, entrylist_t *psoftlist)
list_mechlist_for_soft(char *provname)
{
        mechlist_t      *pmechlist = NULL;
        mechlist_t *pmechlist;
        int             rc;

        if (provname == NULL) {
                return (FAILURE);
        }
```

```
        rc = get_soft_info(provname, &pmechlist, phardlist, psoftlist);
        rc = get_soft_info(provname, &pmechlist);
        if (rc == SUCCESS) {
                (void) filter_mechlist(&pmechlist, RANDOM);
                print_mechlist(provname, pmechlist);
                free_mechlist(pmechlist);
        } else {
                cryptoerror(LOG_STDERR, gettext(
                    "failed to retrieve the mechanism list for %s."),
                    provname);
        }

        return (rc);

}

/*
 * Display the mechanism list for a kernel hardware provider.
 * This implements part of the "cryptoadm list -m" command.
 */
int
list_mechlist_for_hard(char *provname)
{
        mechlist_t      *pmechlist = NULL;
        mechlist_t *pmechlist;
        char            devname[MAXNAMELEN];
        int             inst_num;
        int             count;
        int             rc = SUCCESS;

        if (provname == NULL) {
                return (FAILURE);
        }

        /*
         * Check if the provider is valid. If it is valid, get the number of
         * mechanisms also.
         */
        if (check_hardware_provider(provname, devname, &inst_num, &count) ==
            FAILURE) {
                return (FAILURE);
        }

        /* Get the mechanism list for the kernel hardware provider */
        if ((rc = get_dev_info(devname, inst_num, count, &pmechlist)) ==
            SUCCESS) {
                (void) filter_mechlist(&pmechlist, RANDOM);
                print_mechlist(provname, pmechlist);
                free_mechlist(pmechlist);
        }

        return (rc);
}


/*
 * Display the policy information for a kernel software provider.
 * This implements part of the "cryptoadm list -p" command.
 *
 * Parameters phardlist and psoftlist are supplied by get_kcfconf_info().
 * If NULL, this function obtains it by calling get_kcfconf_info() internally.
 */
int
list_policy_for_soft(char *provname,
    entrylist_t *phardlist, entrylist_t *psoftlist)
```

```
list_policy_for_soft(char *provname)
{
        int             rc;
        entry_t         *pent = NULL;
        mechlist_t      *pmechlist = NULL;
        mechlist_t *pmechlist;
        boolean_t       has_random = B_FALSE;
        boolean_t       has_mechs = B_FALSE;
        boolean_t       in_kernel = B_FALSE;

        if (provname == NULL) {
                return (FAILURE);
        }

        if (check_kernel_for_soft(provname, NULL, &in_kernel) == FAILURE) {
                return (FAILURE);
        } else if (in_kernel == B_FALSE) {
        if ((pent = getent_kef(provname)) == NULL) {
                cryptoerror(LOG_STDERR, gettext("%s does not exist."),
                    provname);
                return (FAILURE);
        }
        pent = getent_kef(provname, phardlist, psoftlist);

        rc = get_soft_info(provname, &pmechlist, phardlist, psoftlist);
        rc = get_soft_info(provname, &pmechlist);
        if (rc == SUCCESS) {
                has_random = filter_mechlist(&pmechlist, RANDOM);
                if (pmechlist != NULL) {
                        has_mechs = B_TRUE;
                        free_mechlist(pmechlist);
                }
        } else {
                cryptoerror(LOG_STDERR, gettext(
                    "failed to retrieve the mechanism list for %s."),
                    provname);
                return (rc);
        }

        print_kef_policy(provname, pent, has_random, has_mechs);
        print_kef_policy(pent, has_random, has_mechs);
        free_entry(pent);
        return (SUCCESS);
}



/*
 * Display the policy information for a kernel hardware provider.
 * This implements part of the "cryptoadm list -p" command.
 *
 * Parameters phardlist and psoftlist are supplied by get_kcfconf_info().
 * If NULL, this function obtains it by calling get_kcfconf_info() internally.
 * Parameter pdevlist is supplied by get_dev_list().
 * If NULL, this function obtains it by calling get_dev_list() internally.
 */
int
list_policy_for_hard(char *provname,
        entrylist_t *phardlist, entrylist_t *psoftlist,
        crypto_get_dev_list_t *pdevlist)
list_policy_for_hard(char *provname)
{
        entry_t         *pent = NULL;
        boolean_t       in_kernel;
        mechlist_t      *pmechlist = NULL;
        entry_t *pent;
```

```
        boolean_t is_active;
        mechlist_t *pmechlist;
        char            devname[MAXNAMELEN];
        int             inst_num;
        int             count;
        int             rc = SUCCESS;
        boolean_t       has_random = B_FALSE;
        boolean_t       has_mechs = B_FALSE;

        if (provname == NULL) {
                return (FAILURE);
        }

        /*
         * Check if the provider is valid. If it is valid, get the number of
         * mechanisms also.
         */
        if (check_hardware_provider(provname, devname, &inst_num, &count) ==
            FAILURE) {
                return (FAILURE);
        }

        /* Get the mechanism list for the kernel hardware provider */
        if ((rc = get_dev_info(devname, inst_num, count, &pmechlist)) ==
            SUCCESS) {
                has_random = filter_mechlist(&pmechlist, RANDOM);

                if (pmechlist != NULL) {
                        has_mechs = B_TRUE;
                        free_mechlist(pmechlist);
                }
        } else {
                cryptoerror(LOG_STDERR, gettext(
                    "failed to retrieve the mechanism list for %s."),
                    devname);
                return (rc);
        }

        /*
         * If the hardware provider has an entry in the kcf.conf file,
         * some of its mechanisms must have been disabled.  Print out
         * the disabled list from the config file entry.  Otherwise,
         * if it is active, then all the mechanisms for it are enabled.
         */
        if ((pent = getent_kef(provname, phardlist, psoftlist)) != NULL) {
                print_kef_policy(provname, pent, has_random, has_mechs);
        if ((pent = getent_kef(provname)) != NULL) {
                print_kef_policy(pent, has_random, has_mechs);
                free_entry(pent);
                return (SUCCESS);
        } else {
                if (check_kernel_for_hard(provname, pdevlist,
                    &in_kernel) == FAILURE) {
                if (check_active_for_hard(provname, &is_active) ==
                    FAILURE) {
                        return (FAILURE);
                } else if (in_kernel == B_TRUE) {
                } else if (is_active == B_TRUE) {
                        (void) printf(gettext(
                            "%s: all mechanisms are enabled."), provname);
                        if (has_random)
                                /*
                                 * TRANSLATION_NOTE
                                 * "random" is a keyword and not to be
                                 * translated.
                                 */
                                (void) printf(gettext(" %s is enabled.\n"),
                                    "random");
                        else
                                (void) printf("\n");
                        return (SUCCESS);
                } else {
                        cryptoerror(LOG_STDERR,
                            gettext("%s does not exist."), provname);
                        return (FAILURE);
                }
        }
}


/*
 * Disable a kernel hardware provider.
 * This implements the "cryptoadm disable" command for
 * kernel hardware providers.
 */

int
disable_kef_hardware(char *provname, boolean_t rndflag, boolean_t allflag,
    mechlist_t *dislist)
{
        crypto_load_dev_disabled_t      *pload_dev_dis = NULL;
        mechlist_t                      *infolist = NULL;
        entry_t                         *pent = NULL;
        crypto_load_dev_disabled_t      *pload_dev_dis;
        mechlist_t      *infolist;
        entry_t         *pent;
        boolean_t                       new_dev_entry = B_FALSE;
        char                            devname[MAXNAMELEN];
        int                             inst_num;
        int                             count;
        int                             fd = -1;
        int     fd;
        int                             rc = SUCCESS;

        if (provname == NULL) {
                return (FAILURE);
        }

        /*
         * Check if the provider is valid. If it is valid, get the number of
         * mechanisms also.
         */
        if (check_hardware_provider(provname, devname, &inst_num, &count)
            == FAILURE) {
                return (FAILURE);
        }

        /* Get the mechanism list for the kernel hardware provider */
        if (get_dev_info(devname, inst_num, count, &infolist) == FAILURE) {
                return (FAILURE);
        }

        /*
         * Get the entry of this hardware provider from the config file.
         * If there is no entry yet, create one for it.
         */
        if ((pent = getent_kef(provname, NULL, NULL)) == NULL) {
                if ((pent = create_entry(provname)) == NULL) {
        if ((pent = getent_kef(provname)) == NULL) {
                if ((pent = malloc(sizeof (entry_t))) == NULL) {
                        cryptoerror(LOG_STDERR, gettext("out of memory."));
                        free_mechlist(infolist);
```

```
                        return (FAILURE);
                }
                new_dev_entry = B_TRUE;
                (void) strlcpy(pent->name, provname, MAXNAMELEN);
                pent->suplist = NULL;
                pent->sup_count = 0;
                pent->dislist = NULL;
                pent->dis_count = 0;
        }

        /*
         * kCF treats random as an internal mechanism. So, we need to
         * filter it from the mechanism list here, if we are NOT disabling
         * or enabling the random feature. Note that we map random feature at
         * cryptoadm(1M) level to the "random" mechanism in kCF.
         */
        if (!rndflag) {
                (void) filter_mechlist(&dislist, RANDOM);
        }

        /* Calculate the new disabled list */
        if (disable_mechs(&pent, infolist, allflag, dislist) == FAILURE) {
                free_mechlist(infolist);
                free_entry(pent);
                return (FAILURE);
        }
        free_mechlist(infolist);

        /* If no mechanisms are to be disabled, return */
        if (pent->dis_count == 0) {
                free_entry(pent);
                return (SUCCESS);
        }

        /* Update the config file with the new entry or the updated entry */
        if (new_dev_entry) {
                rc = update_kcfconf(pent, ADD_MODE);
        } else {
                rc = update_kcfconf(pent, MODIFY_MODE);
        }

        if (rc == FAILURE) {
                free_entry(pent);
                return (FAILURE);
        }

        /* Inform kernel about the new disabled mechanism list */
        if ((pload_dev_dis = setup_dev_dis(pent)) == NULL) {
                free_entry(pent);
                return (FAILURE);
        }
        free_entry(pent);

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                free(pload_dev_dis);
                return (FAILURE);
        }

        if (ioctl(fd, CRYPTO_LOAD_DEV_DISABLED, pload_dev_dis) == -1) {
                cryptodebug("CRYPTO_LOAD_DEV_DISABLED ioctl failed: %s",
                    strerror(errno));
                free(pload_dev_dis);
                (void) close(fd);
                return (FAILURE);
        }

        if (pload_dev_dis->dd_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_LOAD_DEV_DISABLED ioctl return_value = "
                    "%d", pload_dev_dis->dd_return_value);
                free(pload_dev_dis);
                (void) close(fd);
                return (FAILURE);
        }

        free(pload_dev_dis);
        (void) close(fd);
        return (SUCCESS);
}


/*
 * Disable a kernel software provider.
 * This implements the "cryptoadm disable" command for
 * kernel software providers.
 */

int
disable_kef_software(char *provname, boolean_t rndflag, boolean_t allflag,
    mechlist_t *dislist)
{
        crypto_load_soft_disabled_t     *pload_soft_dis = NULL;
        mechlist_t                      *infolist = NULL;
        entry_t                         *pent = NULL;
        entrylist_t                     *phardlist = NULL;
        entrylist_t                     *psoftlist = NULL;
        boolean_t                       in_kernel = B_FALSE;
        int                             fd = -1;
        int                             rc = SUCCESS;
        mechlist_t      *infolist;
        entry_t         *pent;
        boolean_t       is_active;
        int     fd;

        if (provname == NULL) {
                return (FAILURE);
        }

        /* Get the entry of this provider from the config file. */
        if ((pent = getent_kef(provname)) == NULL) {
                cryptoerror(LOG_STDERR,
                    gettext("%s does not exist."), provname);
                return (FAILURE);
        }

        /*
         * Check if the kernel software provider is currently unloaded.
         * If it is unloaded, return FAILURE, because the disable subcommand
         * can not perform on inactive (unloaded) providers.
         */
        if (check_kernel_for_soft(provname, NULL, &in_kernel) == FAILURE) {
        if (check_active_for_soft(provname, &is_active) == FAILURE) {
                free_entry(pent);
                return (FAILURE);
        } else if (in_kernel == B_FALSE) {
        } else if (is_active == B_FALSE) {
                /*
                 * TRANSLATION_NOTE
                 * "disable" is a keyword and not to be translated.
                 */
                cryptoerror(LOG_STDERR,
```

```
                    gettext("%s is not loaded or does not exist."),
                    provname);
                    gettext("can not do %1$s on an unloaded "
                    "kernel software provider -- %2$s."), "disable", provname);
                free_entry(pent);
                return (FAILURE);
        }

        if (get_kcfconf_info(&phardlist, &psoftlist) == FAILURE) {
                cryptoerror(LOG_ERR,
                    "failed to retrieve the providers' "
                    "information from the configuration file - %s.",
                    _PATH_KCF_CONF);
        /* Get the mechanism list for the software provider */
        if (get_soft_info(provname, &infolist) == FAILURE) {
                free(pent);
                return (FAILURE);
        }

        /*
         * Get the entry of this provider from the kcf.conf file, if any.
         * Otherwise, create a new kcf.conf entry for writing back to the file.
         */
        pent = getent_kef(provname, phardlist, psoftlist);
        if (pent == NULL) { /* create a new entry */
                pent = create_entry(provname);
                if (pent == NULL) {
                        cryptodebug("out of memory.");
                        rc = FAILURE;
                        goto out;
                }
        }

        /* Get the mechanism list for the software provider from the kernel */
        if (get_soft_info(provname, &infolist, phardlist, psoftlist) ==
            FAILURE) {
                rc = FAILURE;
                goto out;
        }

        if ((infolist != NULL) && (infolist->name[0] != '\0')) {
                /*
                 * Replace the supportedlist from kcf.conf with possibly
                 * more-up-to-date list from the kernel.  This is the case
                 * for default software providers that had more mechanisms
                 * added in the current version of the kernel.
                 */
                free_mechlist(pent->suplist);
                pent->suplist = infolist;
        }

        /*
         * kCF treats random as an internal mechanism. So, we need to
         * filter it from the mechanism list here, if we are NOT disabling
         * or enabling the random feature. Note that we map random feature at
         * cryptoadm(1M) level to the "random" mechanism in kCF.
         */
        /* See comments in disable_kef_hardware() */
        if (!rndflag) {
                (void) filter_mechlist(&infolist, RANDOM);
        }

        /* Calculate the new disabled list */
        if (disable_mechs(&pent, infolist, allflag, dislist) == FAILURE) {
                rc = FAILURE;
                goto out;
                free_entry(pent);
                free_mechlist(infolist);
                return (FAILURE);
        }

        /* infolist is no longer needed; free it */
        free_mechlist(infolist);

        /* Update the kcf.conf file with the updated entry */
        if (update_kcfconf(pent, MODIFY_MODE) == FAILURE) {
                rc = FAILURE;
                goto out;
                free_entry(pent);
                return (FAILURE);
        }

        /* Setup argument to inform kernel about the new disabled list. */
        /* Inform kernel about the new disabled list. */
        if ((pload_soft_dis = setup_soft_dis(pent)) == NULL) {
                rc = FAILURE;
                goto out;
                free_entry(pent);
                return (FAILURE);
        }

        /* pent is no longer needed; free it. */
        free_entry(pent);

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                cryptoerror(LOG_STDERR,
                    gettext("failed to open %s for RW: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                rc = FAILURE;
                goto out;
                free(pload_soft_dis);
                return (FAILURE);
        }

        /* Inform kernel about the new disabled list. */
        if (ioctl(fd, CRYPTO_LOAD_SOFT_DISABLED, pload_soft_dis) == -1) {
                cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl failed: %s",
                    strerror(errno));
                rc = FAILURE;
                goto out;
                free(pload_soft_dis);
                (void) close(fd);
                return (FAILURE);
        }

        if (pload_soft_dis->sd_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl return_value = "
                    "%d", pload_soft_dis->sd_return_value);
                rc = FAILURE;
                goto out;
                free(pload_soft_dis);
                (void) close(fd);
                return (FAILURE);
        }

out:
        free_entrylist(phardlist);
        free_entrylist(psoftlist);
        free_mechlist(infolist);
        free_entry(pent);
        free(pload_soft_dis);
        if (fd != -1)
```

```
                (void) close(fd);
        return (rc);
        return (SUCCESS);
}


/*
 * Enable a kernel software or hardware provider.
 * This implements the "cryptoadm enable" command for kernel providers.
 */
int
enable_kef(char *provname, boolean_t rndflag, boolean_t allflag,
    mechlist_t *mlist)
{
        crypto_load_soft_disabled_t     *pload_soft_dis = NULL;
        crypto_load_dev_disabled_t      *pload_dev_dis = NULL;
        entry_t                         *pent = NULL;
        entry_t         *pent;
        boolean_t                       redo_flag = B_FALSE;
        boolean_t                       in_kernel = B_FALSE;
        int                             fd = -1;
        int     fd;
        int                             rc = SUCCESS;


        /* Get the entry of this provider from the kcf.conf file, if any. */
        pent = getent_kef(provname, NULL, NULL);
        /* Get the entry with the provider name from the kcf.conf file */
        pent = getent_kef(provname);

        if (is_device(provname)) {
                if (pent == NULL) {
                        /*
                         * This device doesn't have an entry in the config
                         * file, therefore nothing is disabled.
                         */
                        cryptoerror(LOG_STDERR, gettext(
                            "all mechanisms are enabled already for %s."),
                            provname);
                        free_entry(pent);
                        return (SUCCESS);
                }
        } else { /* a software module */
                if (check_kernel_for_soft(provname, NULL, &in_kernel) ==
                    FAILURE) {
                        free_entry(pent);
                if (pent == NULL) {
                        cryptoerror(LOG_STDERR,
                            gettext("%s does not exist."), provname);
                        return (FAILURE);
                } else if (in_kernel == B_FALSE) {
                        cryptoerror(LOG_STDERR, gettext("%s does not exist."),
                            provname);
                        free_entry(pent);
                        return (FAILURE);
                } else if ((pent == NULL) || (pent->dis_count == 0)) {
                } else if (pent->dis_count == 0) {
                        /* nothing to be enabled. */
                        cryptoerror(LOG_STDERR, gettext(
                            "all mechanisms are enabled already for %s."),
                            provname);
                        free_entry(pent);
                        return (SUCCESS);
                }
        }
        /*
         * kCF treats random as an internal mechanism. So, we need to
         * filter it from the mechanism list here, if we are NOT disabling
         * or enabling the random feature. Note that we map random feature at
         * cryptoadm(1M) level to the "random" mechanism in kCF.
         */
        if (!rndflag) {
                /* See comments in disable_kef_hardware() */
                redo_flag = filter_mechlist(&pent->dislist, RANDOM);
                if (redo_flag)
                        pent->dis_count--;
        }

        /* Update the entry by enabling mechanisms for this provider */
        if ((rc = enable_mechs(&pent, allflag, mlist)) != SUCCESS) {
                free_entry(pent);
                return (rc);
        }

        if (redo_flag) {
                mechlist_t *tmp;

                if ((tmp = create_mech(RANDOM)) == NULL) {
                        free_entry(pent);
                        return (FAILURE);
                }
                tmp->next = pent->dislist;
                pent->dislist = tmp;
                pent->dis_count++;
        }

        /*
         * Update the kcf.conf file with the updated entry.
         * For a hardware provider, if there is no more disabled mechanism,
         * remove the entire kcf.conf entry.
         * the entire entry in the config file should be removed.
         */
        if (is_device(pent->name) && (pent->dis_count == 0)) {
                rc = update_kcfconf(pent, DELETE_MODE);
        } else {
                rc = update_kcfconf(pent, MODIFY_MODE);
        }

        if (rc == FAILURE) {
                free_entry(pent);
                return (FAILURE);
        }


        /* Inform Kernel about the policy change */

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                free_entry(pent);
                return (FAILURE);
        }

        if (is_device(provname)) {
                /*  LOAD_DEV_DISABLED */
                if ((pload_dev_dis = setup_dev_dis(pent)) == NULL) {
                        free_entry(pent);
                        return (FAILURE);
                }

                if (ioctl(fd, CRYPTO_LOAD_DEV_DISABLED, pload_dev_dis) == -1) {
```

```
                        cryptodebug("CRYPTO_LOAD_DEV_DISABLED ioctl failed: "
                            "%s", strerror(errno));
                        free_entry(pent);
                        free(pload_dev_dis);
                        (void) close(fd);
                        return (FAILURE);
                }

                if (pload_dev_dis->dd_return_value != CRYPTO_SUCCESS) {
                        cryptodebug("CRYPTO_LOAD_DEV_DISABLED ioctl "
                            "return_value = %d",
                            pload_dev_dis->dd_return_value);
                        free_entry(pent);
                        free(pload_dev_dis);
                        (void) close(fd);
                        return (FAILURE);
                }

        } else { /* a software module */
        } else {
                /* LOAD_SOFT_DISABLED */
                if ((pload_soft_dis = setup_soft_dis(pent)) == NULL) {
                        free_entry(pent);
                        return (FAILURE);
                }

                if (ioctl(fd, CRYPTO_LOAD_SOFT_DISABLED, pload_soft_dis)
                    == -1) {
                        cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl failed: "
                            "%s", strerror(errno));
                        free_entry(pent);
                        free(pload_soft_dis);
                        (void) close(fd);
                        return (FAILURE);
                }

                if (pload_soft_dis->sd_return_value != CRYPTO_SUCCESS) {
                        cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl "
                            "return_value = %d",
                            pload_soft_dis->sd_return_value);
                        free_entry(pent);
                        free(pload_soft_dis);
                        (void) close(fd);
                        return (FAILURE);
                }
        }

        free_entry(pent);
        free(pload_soft_dis);
        (void) close(fd);
        return (SUCCESS);
}


/*
 * Install a software module with the specified mechanism list into the system.
 * This routine adds an entry into the config file for this software module
 * first, then makes a CRYPTO_LOAD_SOFT_CONFIG ioctl call to inform kernel
 * about the new addition.
 */
int
install_kef(char *provname, mechlist_t *mlist)
{
        crypto_load_soft_config_t       *pload_soft_conf = NULL;
        boolean_t                       found;
        entry_t                         *pent = NULL;
        FILE                            *pfile = NULL;
        FILE                            *pfile_tmp = NULL;
        entry_t *pent;
        FILE    *pfile;
        FILE    *pfile_tmp;
        char                            tmpfile_name[MAXPATHLEN];
        char                            *ptr;
        char                            *str;
        char                            *name;
        char                            buffer[BUFSIZ];
        char                            buffer2[BUFSIZ];
        int                             found_count;
        int                             fd = -1;
        int     fd;
        int                             rc = SUCCESS;
        int                             err;

        if ((provname == NULL) || (mlist == NULL)) {
                return (FAILURE);
        }

        /* Check if the provider already exists */
        if ((pent = getent_kef(provname, NULL, NULL)) != NULL) {
        if ((pent = getent_kef(provname)) != NULL) {
                cryptoerror(LOG_STDERR, gettext("%s exists already."),
                    provname);
                free_entry(pent);
                return (FAILURE);
        }

        /* Create an entry with provname and mlist. */
        if ((pent = create_entry(provname)) == NULL) {
        if ((pent = malloc(sizeof (entry_t))) == NULL) {
                cryptoerror(LOG_STDERR, gettext("out of memory."));
                return (FAILURE);
        }

        (void) strlcpy(pent->name, provname, MAXNAMELEN);
        pent->sup_count = get_mech_count(mlist);
        pent->suplist = mlist;
        pent->dis_count = 0;
        pent->dislist = NULL;

        /* Append an entry for this software module to the kcf.conf file. */
        if ((str = ent2str(pent)) == NULL) {
                free_entry(pent);
                return (FAILURE);
        }

        if ((pfile = fopen(_PATH_KCF_CONF, "r+")) == NULL) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to open %s for write.", _PATH_KCF_CONF);
                free_entry(pent);
                return (FAILURE);
        }

        if (lockf(fileno(pfile), F_TLOCK, 0) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to lock the configuration - %s"),
                    strerror(err));
                free_entry(pent);
                (void) fclose(pfile);
```

```
                return (FAILURE);
        }

        /*
         * Create a temporary file in the /etc/crypto directory.
         */
        (void) strlcpy(tmpfile_name, TMPFILE_TEMPLATE, sizeof (tmpfile_name));
        if (mkstemp(tmpfile_name) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to create a temporary file - %s"),
                    strerror(err));
                free_entry(pent);
                (void) fclose(pfile);
                return (FAILURE);
        }

        if ((pfile_tmp = fopen(tmpfile_name, "w")) == NULL) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext("failed to open %s - %s"),
                    tmpfile_name, strerror(err));
                free_entry(pent);
                (void) fclose(pfile);
                return (FAILURE);
        }


        /*
         * Loop thru the config file. If the provider was reserved within a
         * package bracket, just uncomment it.  Otherwise, append it at
         * the end.  The resulting file will be saved in the temp file first.
         */
        found_count = 0;
        rc = SUCCESS;
        while (fgets(buffer, BUFSIZ, pfile) != NULL) {
                found = B_FALSE;
                if (buffer[0] == '#') {
                        (void) strlcpy(buffer2, buffer, BUFSIZ);
                        ptr = buffer2;
                        ptr++;
                        if ((name = strtok(ptr, SEP_COLON)) == NULL) {
                                rc = FAILURE;
                                break;
                        } else if (strcmp(provname, name) == 0) {
                                found = B_TRUE;
                                found_count++;
                        }
                }

                if (found == B_FALSE) {
                        if (fputs(buffer, pfile_tmp) == EOF) {
                                rc = FAILURE;
                        }
                } else {
                        if (found_count == 1) {
                                if (fputs(str, pfile_tmp) == EOF) {
                                        rc = FAILURE;
                                }
                        } else {
                                /*
                                 * Found a second entry with #libname.
                                 * Should not happen. The kcf.conf file
                                 * Should not happen. The kcf.conf ffile
                                 * is corrupted. Give a warning and skip
                                 * this entry.
                                 */
```

```c
                                cryptoerror(LOG_STDERR, gettext(
                                    "(Warning) Found an additional reserved "
                                    "entry for %s."), provname);
                        }
                }

                if (rc == FAILURE) {
                        break;
                }
        }
        (void) fclose(pfile);

        if (rc == FAILURE) {
                cryptoerror(LOG_STDERR, gettext("write error."));
                (void) fclose(pfile_tmp);
                if (unlink(tmpfile_name) != 0) {
                        err = errno;
                        cryptoerror(LOG_STDERR, gettext(
                            "(Warning) failed to remove %s: %s"), tmpfile_name,
                            strerror(err));
                }
                free_entry(pent);
                return (FAILURE);
        }

        if (found_count == 0) {
                /*
                 * This libname was not in package before, append it to the
                 * end of the temp file.
                 */
                if (fputs(str, pfile_tmp) == EOF) {
                        cryptoerror(LOG_STDERR, gettext(
                            "failed to write to %s: %s"), tmpfile_name,
                            strerror(errno));
                        (void) fclose(pfile_tmp);
                        if (unlink(tmpfile_name) != 0) {
                                err = errno;
                                cryptoerror(LOG_STDERR, gettext(
                                    "(Warning) failed to remove %s: %s"),
                                    tmpfile_name, strerror(err));
                        }
                        free_entry(pent);
                        return (FAILURE);
                }
        }

        if (fclose(pfile_tmp) != 0) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to close %s: %s"), tmpfile_name,
                    strerror(err));
                free_entry(pent);
                return (FAILURE);
        }

        if (rename(tmpfile_name, _PATH_KCF_CONF) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to rename %s to %s: %s", tmpfile_name,
                    _PATH_KCF_CONF, strerror(err));
                rc = FAILURE;
        } else if (chmod(_PATH_KCF_CONF,
            S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1) {
                err = errno;
```

```c
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to chmod to %s: %s", _PATH_KCF_CONF,
                    strerror(err));
                rc = FAILURE;
        } else {
                rc = SUCCESS;
        }

        if (rc == FAILURE) {
                if (unlink(tmpfile_name) != 0) {
                        err = errno;
                        cryptoerror(LOG_STDERR, gettext(
                            "(Warning) failed to remove %s: %s"),
                            tmpfile_name, strerror(err));
                }
                free_entry(pent);
                return (FAILURE);
        }


        /* Inform kernel of this new software module. */

        if ((pload_soft_conf = setup_soft_conf(pent)) == NULL) {
                free_entry(pent);
                return (FAILURE);
        }

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                free_entry(pent);
                free(pload_soft_conf);
                return (FAILURE);
        }

        if (ioctl(fd, CRYPTO_LOAD_SOFT_CONFIG, pload_soft_conf) == -1) {
                cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl failed: %s",
                    strerror(errno));
                free_entry(pent);
                free(pload_soft_conf);
                (void) close(fd);
                return (FAILURE);
        }

        if (pload_soft_conf->sc_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl failed, "
                    "return_value = %d", pload_soft_conf->sc_return_value);
                free_entry(pent);
                free(pload_soft_conf);
                (void) close(fd);
                return (FAILURE);
        }

        free_entry(pent);
        free(pload_soft_conf);
        (void) close(fd);
        return (SUCCESS);
}

/*
 * Uninstall the software module. This routine first unloads the software
 * module with 3 ioctl calls, then deletes its entry from the config file.
 * Removing an entry from the config file needs to be done last to ensure
 * that there is still an entry if the earlier unload failed for any reason.
 */
int
uninstall_kef(char *provname)
{
        entry_t         *pent = NULL;
        entry_t         *pent;
        boolean_t       is_active;
        boolean_t       in_package;
        boolean_t       found;
        FILE    *pfile;
        FILE    *pfile_tmp;
        char    tmpfile_name[MAXPATHLEN];
        char    *name;
        char    strbuf[BUFSIZ];
        char    buffer[BUFSIZ];
        char    buffer2[BUFSIZ];
        char    *str;
        int     len;
        int             rc = SUCCESS;
        boolean_t       in_kernel = B_FALSE;
        boolean_t       in_kcfconf = B_FALSE;
        int             fd = -1;
        crypto_load_soft_config_t *pload_soft_conf = NULL;

        /* Check to see if the provider exists first. */
        if (check_kernel_for_soft(provname, NULL, &in_kernel) == FAILURE) {

        /* Check if it is in the kcf.conf file first. */
        if ((pent = getent_kef(provname)) == NULL) {
                cryptoerror(LOG_STDERR,
                    gettext("%s does not exist."), provname);
                return (FAILURE);
        } else if (in_kernel == B_FALSE) {
                cryptoerror(LOG_STDERR, gettext("%s does not exist."),
                    provname);
                return (FAILURE);
        }


        /*
         * If it is loaded, unload it first.  This does 2 ioctl calls:
         * CRYPTO_UNLOAD_SOFT_MODULE and CRYPTO_LOAD_SOFT_DISABLED.
         * Get rid of the disabled list for the provider and get the converted
         * string for the entry.  This is to prepare the string for a provider
         * that is in a package.
         */
        if (unload_kef_soft(provname) == FAILURE) {
        free_mechlist(pent->dislist);
        pent->dis_count = 0;
        pent->dislist = NULL;
        str = ent2str(pent);
        free_entry(pent);
        if (str == NULL) {
                cryptoerror(LOG_STDERR, gettext("internal error."));
                return (FAILURE);
        }
        (void) snprintf(strbuf, sizeof (strbuf), "%s%s", "#", str);
        free(str);

        /* If it is not loaded, unload it first  */
        if (check_active_for_soft(provname, &is_active) == FAILURE) {
                return (FAILURE);
        } else if ((is_active == B_TRUE) &&
            (unload_kef_soft(provname, B_TRUE) == FAILURE)) {
                cryptoerror(LOG_STDERR,
                    gettext("failed to unload %s during uninstall.\n"),
```

```
                    provname);
                    gettext("failed to uninstall %s.\n"), provname);
                return (FAILURE);
        }

        /*
         * Inform kernel to remove the configuration of this software module.
         * Remove the entry from the config file.  If the provider to be
         * uninstalled is in a package, just comment it off.
         */
        if ((pfile = fopen(_PATH_KCF_CONF, "r+")) == NULL) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to open %s for write.", _PATH_KCF_CONF);
                return (FAILURE);
        }

        /* Setup ioctl() parameter */
        pent = getent_kef(provname, NULL, NULL);
        if (pent != NULL) { /* in kcf.conf */
                in_kcfconf = B_TRUE;
                free_mechlist(pent->suplist);
                pent->suplist = NULL;
                pent->sup_count = 0;
        } else if ((pent = create_entry(provname)) == NULL) {
                cryptoerror(LOG_STDERR, gettext("out of memory."));
        if (lockf(fileno(pfile), F_TLOCK, 0) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to lock the configuration - %s"),
                    strerror(err));
                (void) fclose(pfile);
                return (FAILURE);
        }
        if ((pload_soft_conf = setup_soft_conf(pent)) == NULL) {
                free_entry(pent);

        /*
         * Create a temporary file in the /etc/crypto directory to save
         * the new configuration file first.
         */
        (void) strlcpy(tmpfile_name, TMPFILE_TEMPLATE, sizeof (tmpfile_name));
        if (mkstemp(tmpfile_name) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to create a temporary file - %s"),
                    strerror(err));
                (void) fclose(pfile);
                return (FAILURE);
        }

        /* Open the /dev/cryptoadm device */
        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                int     err = errno;
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(err));
                free_entry(pent);
                free(pload_soft_conf);
        if ((pfile_tmp = fopen(tmpfile_name, "w")) == NULL) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext("failed to open %s - %s"),
                    tmpfile_name, strerror(err));
                if (unlink(tmpfile_name) != 0) {
                        err = errno;
                        cryptoerror(LOG_STDERR, gettext(
                            "(Warning) failed to remove %s: %s"), tmpfile_name,
                            strerror(err));
                }
                (void) fclose(pfile);
                return (FAILURE);
        }

        if (ioctl(fd, CRYPTO_LOAD_SOFT_CONFIG,
            pload_soft_conf) == -1) {
                cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl failed: %s",
                    strerror(errno));
                free_entry(pent);
                free(pload_soft_conf);
                (void) close(fd);
        /*
         * Loop thru the config file.  If the kernel software provider
         * to be uninstalled is in a package, just comment it off.
         */
        in_package = B_FALSE;
        while (fgets(buffer, BUFSIZ, pfile) != NULL) {
                found = B_FALSE;
                if (!(buffer[0] == ' ' || buffer[0] == '\n' ||
                    buffer[0] == '\t')) {
                        if (strstr(buffer, " Start ") != NULL) {
                                in_package = B_TRUE;
                        } else if (strstr(buffer, " End ") != NULL) {
                                in_package = B_FALSE;
                        } else if (buffer[0] != '#') {
                                (void) strlcpy(buffer2, buffer, BUFSIZ);

                                /* get rid of trailing '\n' */
                                len = strlen(buffer2);
                                if (buffer2[len-1] == '\n') {
                                        len--;
                                }
                                buffer2[len] = '\0';

                                if ((name = strtok(buffer2, SEP_COLON))
                                    == NULL) {
                                        rc = FAILURE;
                                        break;
                                } else if (strcmp(provname, name) == 0) {
                                        found = B_TRUE;
                                }
                        }
                }

                if (found) {
                        if (in_package) {
                                if (fputs(strbuf, pfile_tmp) == EOF) {
                                        rc = FAILURE;
                                }
                        }
                } else {
                        if (fputs(buffer, pfile_tmp) == EOF) {
                                rc = FAILURE;
                        }
                }

                if (rc == FAILURE) {
                        break;
                }
        }

        if (rc == FAILURE) {
```

```
                cryptoerror(LOG_STDERR, gettext("write error."));
                (void) fclose(pfile);
                (void) fclose(pfile_tmp);
                if (unlink(tmpfile_name) != 0) {
                        err = errno;
                        cryptoerror(LOG_STDERR, gettext(
                            "(Warning) failed to remove %s: %s"), tmpfile_name,
                            strerror(err));
                }
                return (FAILURE);
        }

        if (pload_soft_conf->sc_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl = return_value = %d",
                    pload_soft_conf->sc_return_value);
                free_entry(pent);
                free(pload_soft_conf);
                (void) close(fd);
        (void) fclose(pfile);
        if (fclose(pfile_tmp) != 0) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to close %s: %s"), tmpfile_name,
                    strerror(err));
                return (FAILURE);
        }

        /* ioctl cleanup */
        free(pload_soft_conf);
        (void) close(fd);
        /* Now update the real config file */
        if (rename(tmpfile_name, _PATH_KCF_CONF) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to rename %1$s to %2$s: %3$s", tmpfile,
                    _PATH_KCF_CONF, strerror(err));
                rc = FAILURE;
        } else if (chmod(_PATH_KCF_CONF,
            S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to chmod to %s: %s", _PATH_KCF_CONF,
                    strerror(err));
                rc = FAILURE;
        } else {
                rc = SUCCESS;
        }


        /* Finally, remove entry from kcf.conf, if present */
        if (in_kcfconf && (pent != NULL)) {
                rc = update_kcfconf(pent, DELETE_MODE);
        if ((rc == FAILURE) && (unlink(tmpfile_name) != 0)) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext(
                    "(Warning) failed to remove %s: %s"), tmpfile_name,
                    strerror(err));
        }

        free_entry(pent);
        return (rc);
```

```
}


/*
 * Implement the "cryptoadm refresh" command for global zones.
 * That is, send the current contents of kcf.conf to the kernel via ioctl().
 */
int
refresh(void)
{
        crypto_get_soft_list_t          *psoftlist_kernel = NULL;
        crypto_load_soft_config_t       *pload_soft_conf = NULL;
        crypto_load_soft_disabled_t     *pload_soft_dis = NULL;
        crypto_load_dev_disabled_t      *pload_dev_dis = NULL;
        entrylist_t                     *pdevlist = NULL;
        entrylist_t                     *psoftlist = NULL;
        entrylist_t                     *ptr;
        int                             fd = -1;
        boolean_t       found;
        char    *psoftname;
        int     fd;
        int                             rc = SUCCESS;
        int                             err;
        int     i;

        if (get_soft_list(&psoftlist_kernel) == FAILURE) {
                cryptoerror(LOG_ERR, gettext("Failed to retrieve the "
                    "software provider list from kernel."));
                return (FAILURE);
        }

        if (get_kcfconf_info(&pdevlist, &psoftlist) == FAILURE) {
                cryptoerror(LOG_ERR, "failed to retrieve the providers' "
                    "information from the configuration file - %s.",
                    _PATH_KCF_CONF);
                return (FAILURE);
        }

        /*
         * If a kernel software provider is in kernel, but it is not in the
         * kcf.conf file, it must have been pkgrm'ed and needs to be unloaded
         * now.
         */
        if (psoftlist_kernel->sl_soft_count > 0) {
                psoftname = psoftlist_kernel->sl_soft_names;
                for (i = 0; i < psoftlist_kernel->sl_soft_count; i++) {
                        ptr = psoftlist;
                        found = B_FALSE;
                        while (ptr != NULL) {
                                if (strcmp(psoftname, ptr->pent->name) == 0) {
                                        found = B_TRUE;
                                        break;
                                }
                                ptr = ptr->next;
                        }

                        if (!found) {
                                rc = unload_kef_soft(psoftname, B_FALSE);
                                if (rc == FAILURE) {
                                        cryptoerror(LOG_ERR, gettext(
                                            "WARNING - the provider %s is "
                                            "still in kernel."), psoftname);
                                }
                        }
                        psoftname = psoftname + strlen(psoftname) + 1;
                }
```

```
        }
        free(psoftlist_kernel);

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(err));
                free(psoftlist);
                free(pdevlist);
                return (FAILURE);
        }

        /*
         * For each software provider module, pass two sets of information to
         * the kernel: the supported list and the disabled list.
         * For each software module, pass two sets of information to kernel
         * - the supported list and the disabled list
         */
        for (ptr = psoftlist; ptr != NULL; ptr = ptr->next) {
                entry_t         *pent = ptr->pent;

        ptr = psoftlist;
        while (ptr != NULL) {
                /* load the supported list */
                if ((pload_soft_conf = setup_soft_conf(pent)) == NULL) {
                        cryptodebug("setup_soft_conf() failed");
                if ((pload_soft_conf = setup_soft_conf(ptr->pent)) == NULL) {
                        rc = FAILURE;
                        break;
                }

                if (!pent->load) { /* unloaded--mark as loaded */
                        pent->load = B_TRUE;
                        rc = update_kcfconf(pent, MODIFY_MODE);
                        if (rc != SUCCESS) {
                                free(pload_soft_conf);
                                break;
                        }
                }

                if (ioctl(fd, CRYPTO_LOAD_SOFT_CONFIG, pload_soft_conf)
                    == -1) {
                        cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl failed: %s",
                            strerror(errno));
                        free(pload_soft_conf);
                        rc = FAILURE;
                        break;
                }

                if (pload_soft_conf->sc_return_value != CRYPTO_SUCCESS) {
                        cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl "
                            "return_value = %d",
                            pload_soft_conf->sc_return_value);
                        free(pload_soft_conf);
                        rc = FAILURE;
                        break;
                }

                free(pload_soft_conf);

                /* load the disabled list */
                if (ptr->pent->dis_count != 0) {
                        pload_soft_dis = setup_soft_dis(ptr->pent);
                        if (pload_soft_dis == NULL) {
                                cryptodebug("setup_soft_dis() failed");
                                free(pload_soft_dis);
                                rc = FAILURE;
                                break;
                        }

                        if (ioctl(fd, CRYPTO_LOAD_SOFT_DISABLED,
                            pload_soft_dis) == -1) {
                                cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl "
                                    "failed: %s", strerror(errno));
                                free(pload_soft_dis);
                                rc = FAILURE;
                                break;
                        }

                        if (pload_soft_dis->sd_return_value !=
                            CRYPTO_SUCCESS) {
                                cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl "
                                    "return_value = %d",
                                    pload_soft_dis->sd_return_value);
                                free(pload_soft_dis);
                                rc = FAILURE;
                                break;
                        }
                        free(pload_soft_dis);
                }

                free(pload_soft_conf);
                ptr = ptr->next;
        }

        if (rc != SUCCESS) {
                (void) close(fd);
                return (rc);
        }


        /*
         * For each hardware provider module, pass the disabled list
         * information to the kernel.
         */
        for (ptr = pdevlist; ptr != NULL; ptr = ptr->next) {
        /* Pass the disabledlist information for Device to kernel */
        ptr = pdevlist;
        while (ptr != NULL) {
                /* load the disabled list */
                if (ptr->pent->dis_count != 0) {
                        pload_dev_dis = setup_dev_dis(ptr->pent);
                        if (pload_dev_dis == NULL) {
                                rc = FAILURE;
                                break;
                        }

                        if (ioctl(fd, CRYPTO_LOAD_DEV_DISABLED, pload_dev_dis)
                            == -1) {
                                cryptodebug("CRYPTO_LOAD_DEV_DISABLED ioctl "
                                    "failed: %s", strerror(errno));
                                free(pload_dev_dis);
                                rc = FAILURE;
                                break;
                        }

                        if (pload_dev_dis->dd_return_value != CRYPTO_SUCCESS) {
                                cryptodebug("CRYPTO_LOAD_DEV_DISABLED ioctl "
                                    "return_value = %d",
                                    pload_dev_dis->dd_return_value);
                                free(pload_dev_dis);
                                rc = FAILURE;
```

```
                                break;
                        }
                        free(pload_dev_dis);
                }

                ptr = ptr->next;
        }

        (void) close(fd);
        return (rc);
}

/*
 * Unload the kernel software provider. Before calling this function, the
 * caller should check to see if the provider is in the kernel.
 *
 * This routine makes 2 ioctl calls to remove it completely from the kernel:
 *      CRYPTO_UNLOAD_SOFT_MODULE - does a modunload of the KCF module
 *      CRYPTO_LOAD_SOFT_DISABLED - updates kernel disabled mechanism list
 *
 * This implements part of "cryptoadm unload" and "cryptoadm uninstall".
 * caller should check if the provider is in the config file and if it
 * is kernel. This routine makes 3 ioctl calls to remove it from kernel
 * completely. The argument do_check set to B_FALSE means that the
 * caller knows the provider is not the config file and hence the check
 * is skipped.
 */
int
unload_kef_soft(char *provname)
unload_kef_soft(char *provname, boolean_t do_check)
{
        crypto_unload_soft_module_t     *punload_soft = NULL;
        crypto_load_soft_config_t       *pload_soft_conf = NULL;
        crypto_load_soft_disabled_t     *pload_soft_dis = NULL;
        entry_t                         *pent = NULL;
        int                             fd = -1;
        int                             err;
        int     fd;

        if (provname == NULL) {
                cryptoerror(LOG_STDERR, gettext("internal error."));
                return (FAILURE);
        }

        pent = getent_kef(provname, NULL, NULL);
        if (pent == NULL) { /* not in kcf.conf */
        if (!do_check) {
                /* Construct an entry using the provname */
                pent = create_entry(provname);
                pent = calloc(1, sizeof (entry_t));
                if (pent == NULL) {
                        cryptoerror(LOG_STDERR, gettext("out of memory."));
                        return (FAILURE);
                }
                (void) strlcpy(pent->name, provname, MAXNAMELEN);
        } else if ((pent = getent_kef(provname)) == NULL) {
                cryptoerror(LOG_STDERR, gettext("%s does not exist."),
                    provname);
                return (FAILURE);
        }

        /* Open the admin_ioctl_device */
        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDWR)) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(err));
                free_entry(pent);
                return (FAILURE);
        }

        /* Inform kernel to unload this software module */
        if ((punload_soft = setup_unload_soft(pent)) == NULL) {
                free_entry(pent);
                (void) close(fd);
                return (FAILURE);
        }

        if (ioctl(fd, CRYPTO_UNLOAD_SOFT_MODULE, punload_soft) == -1) {
                cryptodebug("CRYPTO_UNLOAD_SOFT_MODULE ioctl failed: %s",
                    strerror(errno));
                free_entry(pent);
                free(punload_soft);
                (void) close(fd);
                return (FAILURE);
        }

        if (punload_soft->sm_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_UNLOAD_SOFT_MODULE ioctl return_value = "
                    "%d", punload_soft->sm_return_value);
                /*
                 * If the return value is CRYPTO_UNKNOWN_PROVIDER, it means
                 * that the provider is not registered yet.  Should just
                 * continue.
                 */
                if (punload_soft->sm_return_value != CRYPTO_UNKNOWN_PROVIDER) {
                        free_entry(pent);
                        free(punload_soft);
                        (void) close(fd);
                        return (FAILURE);
                }
        }

        free(punload_soft);

        /*
         * Inform kernel to remove the configuration of this software
         * module.
         */
        free_mechlist(pent->suplist);
        pent->suplist = NULL;
        pent->sup_count = 0;
        if ((pload_soft_conf = setup_soft_conf(pent)) == NULL) {
                free_entry(pent);
                (void) close(fd);
                return (FAILURE);
        }

        if (ioctl(fd, CRYPTO_LOAD_SOFT_CONFIG, pload_soft_conf) == -1) {
                cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl failed: %s",
                    strerror(errno));
                free_entry(pent);
                free(pload_soft_conf);
                (void) close(fd);
                return (FAILURE);
        }

        if (pload_soft_conf->sc_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_LOAD_SOFT_CONFIG ioctl return_value = "
                    "%d", pload_soft_conf->sc_return_value);
                free_entry(pent);
                free(pload_soft_conf);
                (void) close(fd);
```

```
                return (FAILURE);
        }

        free(pload_soft_conf);

        /* Inform kernel to remove the disabled entries if any */
        if (pent->dis_count == 0) {
                free_entry(pent);
                (void) close(fd);
                return (SUCCESS);
        } else {
                free_mechlist(pent->dislist);
                pent->dislist = NULL;
                pent->dis_count = 0;
        }

        if ((pload_soft_dis = setup_soft_dis(pent)) == NULL) {
                free_entry(pent);
                (void) close(fd);
                return (FAILURE);
        }

        /* pent is no longer needed; free it */
        free_entry(pent);

        if (ioctl(fd, CRYPTO_LOAD_SOFT_DISABLED, pload_soft_dis) == -1) {
                cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl failed: %s",
                    strerror(errno));
                free(pload_soft_dis);
                (void) close(fd);
                return (FAILURE);
        }

        if (pload_soft_dis->sd_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_LOAD_SOFT_DISABLED ioctl return_value = "
                    "%d", pload_soft_dis->sd_return_value);
                free(pload_soft_dis);
                (void) close(fd);
                return (FAILURE);
        }

        free(pload_soft_dis);
        (void) close(fd);
        return (SUCCESS);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   14657 Tue Oct 28 16:45:23 2008
new/usr/src/cmd/cmd-crypto/cryptoadm/adm_kef_ioctl.c
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
/*
 * 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.
 * Common Development and Distribution License, Version 1.0 only
 * (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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <locale.h>
#include <libgen.h>
#include <sys/types.h>
#include <zone.h>
#include <sys/crypto/ioctladmin.h>
#include "cryptoadm.h"

#define DEFAULT_DEV_NUM 5
#define DEFAULT_SOFT_NUM 10

static crypto_get_soft_info_t *setup_get_soft_info(char *, int);

/*
 * Prepare the argument for the LOAD_SOFT_CONFIG ioctl call for the
 * provider pointed by pent.  Return NULL if out of memory.
 */
crypto_load_soft_config_t *
setup_soft_conf(entry_t *pent)
{
        crypto_load_soft_config_t       *pload_soft_conf;
        mechlist_t      *plist;
        uint_t          sup_count;
        size_t          extra_mech_size = 0;
        int             i;
```

```
        if (pent == NULL) {
                return (NULL);
        }

        sup_count = pent->sup_count;
        if (sup_count > 1) {
                extra_mech_size = sizeof (crypto_mech_name_t) *
                    (sup_count - 1);
        }

        pload_soft_conf = malloc(sizeof (crypto_load_soft_config_t) +
            extra_mech_size);
        if (pload_soft_conf == NULL) {
                cryptodebug("out of memory.");
                return (NULL);
        }

        (void) strlcpy(pload_soft_conf->sc_name, pent->name, MAXNAMELEN);
        pload_soft_conf->sc_count = sup_count;

        i = 0;
        plist =  pent->suplist;
        while (i < sup_count) {
                (void) strlcpy(pload_soft_conf->sc_list[i++],
                    plist->name, CRYPTO_MAX_MECH_NAME);
                plist = plist->next;
        }

        return (pload_soft_conf);
}


/*
 * Prepare the argument for the LOAD_SOFT_DISABLED ioctl call for the
 * provider pointed by pent.  Return NULL if out of memory.
 */
crypto_load_soft_disabled_t *
setup_soft_dis(entry_t *pent)
{
        crypto_load_soft_disabled_t     *pload_soft_dis = NULL;
        mechlist_t      *plist = NULL;
        crypto_load_soft_disabled_t     *pload_soft_dis;
        mechlist_t      *plist;
        size_t          extra_mech_size = 0;
        uint_t          dis_count;
        int             i;

        if (pent == NULL) {
                return (NULL);
        }

        dis_count = pent->dis_count;
        if (dis_count > 1) {
                extra_mech_size = sizeof (crypto_mech_name_t) *
                    (dis_count - 1);
        }

        pload_soft_dis = malloc(sizeof (crypto_load_soft_disabled_t) +
            extra_mech_size);
        if (pload_soft_dis == NULL) {
                cryptodebug("out of memory.");
                return (NULL);
        }

        (void) strlcpy(pload_soft_dis->sd_name, pent->name, MAXNAMELEN);
        pload_soft_dis->sd_count = dis_count;
        i = 0;
        plist =  pent->dislist;
        while (i < dis_count) {
                (void) strlcpy(pload_soft_dis->sd_list[i++],
                    plist->name, CRYPTO_MAX_MECH_NAME);
                plist = plist->next;
        }

        return (pload_soft_dis);
}


/*
 * Prepare the argument for the LOAD_DEV_DISABLED ioctl call for the
 * provider pointed by pent.  Return NULL if out of memory.
 */
crypto_load_dev_disabled_t *
setup_dev_dis(entry_t *pent)
{
        crypto_load_dev_disabled_t      *pload_dev_dis = NULL;
        mechlist_t      *plist = NULL;
        crypto_load_dev_disabled_t      *pload_dev_dis;
        mechlist_t      *plist;
        size_t          extra_mech_size = 0;
        uint_t          dis_count;
        int             i;
        char            pname[MAXNAMELEN];
        int             inst_num;

        if (pent == NULL) {
                return (NULL);
        }

        /* get the device name and the instance number */
        if (split_hw_provname(pent->name, pname, &inst_num) == FAILURE) {
                return (NULL);
        }

        /* allocate space for pload_dev_des */
        dis_count = pent->dis_count;
        if (dis_count > 1) {
                extra_mech_size = sizeof (crypto_mech_name_t) *
                    (dis_count - 1);
        }

        pload_dev_dis = malloc(sizeof (crypto_load_dev_disabled_t) +
            extra_mech_size);
        if (pload_dev_dis == NULL) {
                cryptodebug("out of memory.");
                return (NULL);
        }

        /* set the values for pload_dev_dis */
        (void) strlcpy(pload_dev_dis->dd_dev_name, pname, MAXNAMELEN);
        pload_dev_dis->dd_dev_instance = inst_num;
        pload_dev_dis->dd_count = dis_count;

        i = 0;
        plist =  pent->dislist;
        while (i < dis_count) {
                (void) strlcpy(pload_dev_dis->dd_list[i++],
                    plist->name, CRYPTO_MAX_MECH_NAME);
                plist = plist->next;
        }
```

```
        return (pload_dev_dis);
}
```
______***unchanged_portion_omitted_***

```
/*
 * Prepare the calling argument for the GET_SOFT_INFO call for the provider
 * with the number of mechanisms specified in the second argument.
 *
 * Called by get_soft_info().
 */
static crypto_get_soft_info_t *
setup_get_soft_info(char *provname, int count)
{
        crypto_get_soft_info_t  *psoft_info;
        size_t                  extra_mech_size = 0;

        if (provname == NULL) {
                return (NULL);
        }

        if (count > 1) {
                extra_mech_size = sizeof (crypto_mech_name_t) * (count - 1);
        }

        psoft_info = malloc(sizeof (crypto_get_soft_info_t) + extra_mech_size);
        if (psoft_info == NULL) {
                cryptodebug("out of memory.");
                return (NULL);
        }

        (void) strlcpy(psoft_info->si_name, provname, MAXNAMELEN);
        psoft_info->si_count = count;

        return (psoft_info);
}


/*
 * Get the device list from kernel.
 */
int
get_dev_list(crypto_get_dev_list_t **ppdevlist)
{
        crypto_get_dev_list_t   *pdevlist;
        int                     fd = -1;
        int fd;
        int                     count = DEFAULT_DEV_NUM;

        pdevlist = malloc(sizeof (crypto_get_dev_list_t) +
            sizeof (crypto_dev_list_entry_t) * (count - 1));
        if (pdevlist == NULL) {
                cryptodebug("out of memory.");
                return (FAILURE);
        }

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDONLY)) == -1) {
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                return (FAILURE);
        }

        pdevlist->dl_dev_count = count;
        if (ioctl(fd, CRYPTO_GET_DEV_LIST, pdevlist) == -1) {
                cryptodebug("CRYPTO_GET_DEV_LIST ioctl failed: %s",
                    strerror(errno));
                free(pdevlist);
                (void) close(fd);
                return (FAILURE);
        }

        /* BUFFER is too small, get the number of devices and retry it. */
        if (pdevlist->dl_return_value == CRYPTO_BUFFER_TOO_SMALL) {
                count = pdevlist->dl_dev_count;
                free(pdevlist);
                pdevlist = malloc(sizeof (crypto_get_dev_list_t) +
                    sizeof (crypto_dev_list_entry_t) * (count - 1));
                if (pdevlist == NULL) {
                        cryptodebug("out of memory.");
                        (void) close(fd);
                        return (FAILURE);
                }

                if (ioctl(fd, CRYPTO_GET_DEV_LIST, pdevlist) == -1) {
                        cryptodebug("CRYPTO_GET_DEV_LIST ioctl failed: %s",
                            strerror(errno));
                        free(pdevlist);
                        (void) close(fd);
                        return (FAILURE);
                }
        }

        if (pdevlist->dl_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_GET_DEV_LIST ioctl failed, "
                    "return_value = %d", pdevlist->dl_return_value);
                free(pdevlist);
                (void) close(fd);
                return (FAILURE);
        }

        *ppdevlist = pdevlist;
        (void) close(fd);
        return (SUCCESS);
}


/*
 * Get all the mechanisms supported by the hardware provider.
 * The result will be stored in the second argument.
 */
int
get_dev_info(char *devname, int inst_num, int count, mechlist_t **ppmechlist)
{
        crypto_get_dev_info_t   *dev_info;
        mechlist_t      *phead;
        mechlist_t      *pcur;
        mechlist_t      *pmech;
        int             fd = -1;
        int fd;
        int             i;
        int             rc;

        if (devname == NULL || count < 1) {
                cryptodebug("get_dev_info(): devname is NULL or bogus count");
                return (FAILURE);
        }

        /* Set up the argument for the CRYPTO_GET_DEV_INFO ioctl call */
        dev_info = malloc(sizeof (crypto_get_dev_info_t) +
            sizeof (crypto_mech_name_t) * (count - 1));
        if (dev_info == NULL) {
                cryptodebug("out of memory.");
```

```
                return (FAILURE);
        }
        (void) strlcpy(dev_info->di_dev_name, devname, MAXNAMELEN);
        dev_info->di_dev_instance = inst_num;
        dev_info->di_count = count;

        /* Open the ioctl device */
        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDONLY)) == -1) {
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                free(dev_info);
                return (FAILURE);
        }

        if (ioctl(fd, CRYPTO_GET_DEV_INFO, dev_info) == -1) {
                cryptodebug("CRYPTO_GET_DEV_INFO ioctl failed: %s",
                    strerror(errno));
                free(dev_info);
                (void) close(fd);
                return (FAILURE);
        }

        if (dev_info->di_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_GET_DEV_INFO ioctl failed, "
                    "return_value = %d", dev_info->di_return_value);
                free(dev_info);
                (void) close(fd);
                return (FAILURE);
        }

        phead = pcur = NULL;
        rc = SUCCESS;
        for (i = 0; i < dev_info->di_count; i++) {
                pmech = create_mech(&dev_info->di_list[i][0]);
                if (pmech == NULL) {
                        rc = FAILURE;
                        break;
                } else {
                        if (phead == NULL) {
                                phead = pcur = pmech;
                        } else {
                                pcur->next = pmech;
                                pcur = pmech;
                        }
                }
        }

        if (rc == SUCCESS) {
                *ppmechlist = phead;
        } else {
                free_mechlist(phead);
        }

        free(dev_info);
        (void) close(fd);
        return (rc);
}


/*
 * Get the supported mechanism list of the software provider from kernel.
 *
 * Parameters phardlist and psoftlist are supplied by get_kcfconf_info().
 * If NULL, this function calls get_kcfconf_info() internally.
 */
```

```
int
get_soft_info(char *provname, mechlist_t **ppmechlist,
        entrylist_t *phardlist, entrylist_t *psoftlist)
get_soft_info(char *provname, mechlist_t **ppmechlist)
{
        boolean_t               in_kernel = B_FALSE;
        crypto_get_soft_info_t  *psoft_info;
        mechlist_t              *phead;
        mechlist_t              *pmech;
        mechlist_t              *pcur;
        entry_t                 *pent = NULL;
        entry_t *pent;
        int                     count;
        int                     fd = -1;
        int     fd;
        int                     rc;
        int                     i;

        if (provname == NULL) {
                return (FAILURE);
        }

        if (getzoneid() == GLOBAL_ZONEID) {
                /* use kcf.conf for kernel software providers in global zone */
                if ((pent = getent_kef(provname, phardlist, psoftlist)) ==
                    NULL) {

                        /* No kcf.conf entry for this provider */
                        if (check_kernel_for_soft(provname, NULL, &in_kernel)
                            == FAILURE) {
                if ((pent = getent_kef(provname)) == NULL) {
                        cryptoerror(LOG_STDERR, gettext("%s does not exist."),
                            provname);
                                return (FAILURE);
                        } else if (in_kernel == B_FALSE) {
                                cryptoerror(LOG_STDERR,
                                    gettext("%s does not exist."), provname);
                                return (FAILURE);
                        }

                        /*
                         * Set mech count to 1.  It will be reset to the
                         * correct value later if the setup buffer is too small.
                         */
                        count = 1;
                } else {
                        count = pent->sup_count;
                        free_entry(pent);
                }
        } else {
                /*
                 * kcf.conf not there in non-global zone: set mech count to 1.
                 * It will be reset to the correct value later if the setup
                 * buffer is too small.
                 * kcf.conf not there in non-global zone, set mech count to 1;
                 * it will be reset to the correct value later if the setup
                 * buffer is too small
                 */
                count = 1;
        }

        if ((psoft_info = setup_get_soft_info(provname, count)) == NULL) {
                return (FAILURE);
        }

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDONLY)) == -1) {
```

```
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                free(psoft_info);
                return (FAILURE);
        }

        /* make GET_SOFT_INFO ioctl call */
        if ((rc = ioctl(fd, CRYPTO_GET_SOFT_INFO, psoft_info)) == -1) {
                cryptodebug("CRYPTO_GET_SOFT_INFO ioctl failed: %s",
                    strerror(errno));
                (void) close(fd);
                free(psoft_info);
                return (FAILURE);
        }

        /* BUFFER is too small, get the number of mechanisms and retry it. */
        if (psoft_info->si_return_value == CRYPTO_BUFFER_TOO_SMALL) {
                count = psoft_info->si_count;
                free(psoft_info);
                if ((psoft_info = setup_get_soft_info(provname, count))
                    == NULL) {
                        (void) close(fd);
                        return (FAILURE);
                } else {
                        rc = ioctl(fd, CRYPTO_GET_SOFT_INFO, psoft_info);
                        if (rc == -1) {
                                cryptodebug("CRYPTO_GET_SOFT_INFO ioctl "
                                    "failed: %s", strerror(errno));
                                (void) close(fd);
                                free(psoft_info);
                                return (FAILURE);
                        }
                }
        }

        (void) close(fd);
        if (psoft_info->si_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_GET_SOFT_INFO ioctl failed, "
                    "return_value = %d", psoft_info->si_return_value);
                free(psoft_info);
                return (FAILURE);
        }


        /* Build the mechanism linked list and return it */
        /* Get the mechanism list and return it */
        rc = SUCCESS;
        phead = pcur = NULL;
        for (i = 0; i < psoft_info->si_count; i++) {
                pmech = create_mech(&psoft_info->si_list[i][0]);
                if (pmech == NULL) {
                        rc = FAILURE;
                        break;
                } else {
                        if (phead == NULL) {
                                phead = pcur = pmech;
                        } else {
                                pcur->next = pmech;
                                pcur = pmech;
                        }
                }
        }

        if (rc == FAILURE) {
                free_mechlist(phead);
        } else {
                *ppmechlist = phead;
        }

        free(psoft_info);
        return (rc);
}


/*
 * Get the kernel software provider list from kernel.
 */
int
get_soft_list(crypto_get_soft_list_t **ppsoftlist)
{
        crypto_get_soft_list_t *psoftlist = NULL;
        int     count = DEFAULT_SOFT_NUM;
        int     len;
        int     fd = -1;
        int fd;

        if ((fd = open(ADMIN_IOCTL_DEVICE, O_RDONLY)) == -1) {
                cryptoerror(LOG_STDERR, gettext("failed to open %s: %s"),
                    ADMIN_IOCTL_DEVICE, strerror(errno));
                return (FAILURE);
        }

        len = MAXNAMELEN * count;
        psoftlist = malloc(sizeof (crypto_get_soft_list_t) + len);
        if (psoftlist == NULL) {
                cryptodebug("out of memory.");
                (void) close(fd);
                return (FAILURE);
        }
        psoftlist->sl_soft_names = (caddr_t)(psoftlist + 1);
        psoftlist->sl_soft_count = count;
        psoftlist->sl_soft_len = len;

        if (ioctl(fd, CRYPTO_GET_SOFT_LIST, psoftlist) == -1) {
                cryptodebug("CRYPTO_GET_SOFT_LIST ioctl failed: %s",
                    strerror(errno));
                free(psoftlist);
                (void) close(fd);
                return (FAILURE);
        }

        /*
         * if BUFFER is too small, get the number of software providers and
         * the minimum length needed for names and length and retry it.
         */
        if (psoftlist->sl_return_value == CRYPTO_BUFFER_TOO_SMALL) {
                count = psoftlist->sl_soft_count;
                len = psoftlist->sl_soft_len;
                free(psoftlist);
                psoftlist = malloc(sizeof (crypto_get_soft_list_t) + len);
                if (psoftlist == NULL) {
                        cryptodebug("out of memory.");
                        (void) close(fd);
                        return (FAILURE);
                }
                psoftlist->sl_soft_names = (caddr_t)(psoftlist + 1);
                psoftlist->sl_soft_count = count;
                psoftlist->sl_soft_len = len;

                if (ioctl(fd, CRYPTO_GET_SOFT_LIST, psoftlist) == -1) {
                        cryptodebug("CRYPTO_GET_SOFT_LIST ioctl failed:"
                            "%s", strerror(errno));
```

```
                        free(psoftlist);
                        (void) close(fd);
                        return (FAILURE);
                }
        }

        if (psoftlist->sl_return_value != CRYPTO_SUCCESS) {
                cryptodebug("CRYPTO_GET_SOFT_LIST ioctl failed, "
                    "return_value = %d", psoftlist->sl_return_value);
                free(psoftlist);
                (void) close(fd);
                return (FAILURE);
        }

        *ppsoftlist = psoftlist;
        (void) close(fd);
        return (SUCCESS);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   31046 Tue Oct 28 16:45:29 2008
new/usr/src/cmd/cmd-crypto/cryptoadm/adm_kef_util.c
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <time.h>
#include <unistd.h>
#include <locale.h>
#include <sys/types.h>
#include <zone.h>
#include <sys/stat.h>
#include "cryptoadm.h"

static int err; /* To store errno which may be overwritten by gettext() */
static int build_entrylist(entry_t *, entrylist_t **);
static entry_t *dup_entry(entry_t *);
static mechlist_t *dup_mechlist(mechlist_t *);
static entry_t *getent(char *, entrylist_t *);
static int interpret(char *, entry_t **);
static int parse_sup_dis_list(char *, entry_t *);
static int parse_dislist(char *, entry_t *);


/*
 * Duplicate the mechanism list.  A null pointer is returned if the storage
 * space available is insufficient or the input argument is NULL.
 */
static mechlist_t *
dup_mechlist(mechlist_t *plist)
{
        mechlist_t      *pres = NULL;
        mechlist_t      *pcur;
        mechlist_t      *ptmp;
        int             rc = SUCCESS;

        while (plist != NULL) {
```

```
                if (!(ptmp = create_mech(plist->name))) {
                        rc = FAILURE;
                        break;
                }

                if (pres == NULL) {
                        pres = pcur = ptmp;
                } else {
                        pcur->next = ptmp;
                        pcur = pcur->next;
                }
                plist = plist->next;
        }

        if (rc != SUCCESS) {
                free_mechlist(pres);
                return (NULL);
        }

        return (pres);
}
______unchanged_portion_omitted_

/*
 * Create one item of type entry_t with the provider name.
 * Return NULL if there's not enough memory or provname is NULL.
 */
entry_t *
create_entry(char *provname)
{
        entry_t         *pent = NULL;

        if (provname == NULL) {
                return (NULL);
        }

        pent = calloc(1, sizeof (entry_t));
        if (pent == NULL) {
                cryptodebug("out of memory.");
                return (NULL);
        }

        (void) strlcpy(pent->name, provname, MAXNAMELEN);
        pent->suplist = NULL;
        pent->sup_count = 0;
        pent->dislist = NULL;
        pent->dis_count = 0;
        pent->load = B_TRUE;

        return (pent);
}

/*
 * Duplicate an entry for a provider from kcf.conf.
 * Return NULL if memory is insufficient or the input argument is NULL.
 * Called by getent().
 * Duplicate an entry.  A null pointer is returned if the storage space
 * available is insufficient or the input argument is NULL.
 */
static entry_t *
dup_entry(entry_t *pent1)
{
        entry_t *pent2 = NULL;

        if (pent1 == NULL) {
                return (NULL);
        }

        if ((pent2 = create_entry(pent1->name)) == NULL) {
        if ((pent2 = malloc(sizeof (entry_t))) == NULL) {
                cryptodebug("out of memory.");
                return (NULL);
        }

        (void) strlcpy(pent2->name, pent1->name, sizeof (pent2->name));
        pent2->sup_count = pent1->sup_count;
        pent2->dis_count = pent1->dis_count;
        pent2->load = pent1->load;
        pent2->suplist = NULL;
        pent2->dislist = NULL;
        if (pent1->suplist != NULL) {
                pent2->suplist = dup_mechlist(pent1->suplist);
                if (pent2->suplist == NULL) {
                        free_entry(pent2);
                        return (NULL);
                }
        }
        if (pent1->dislist != NULL) {
                pent2->dislist = dup_mechlist(pent1->dislist);
                if (pent2->dislist == NULL) {
                        free_entry(pent2);
                        return (NULL);
                }
        }

        return (pent2);
}


/*
 * This routine parses the disabledlist or the supportedlist of an entry
 * in the kcf.conf configuration file.
 *
 * Arguments:
 *      buf: an input argument which is a char string with the format of
 *           "disabledlist=m1,m2,..." or "supportedlist=m1,m2,..."
 *      pent: the entry for the disabledlist.  This is an IN/OUT argument.
 *
 * Return value: SUCCESS or FAILURE.
 */
static int
parse_sup_dis_list(char *buf, entry_t *pent)
parse_dislist(char *buf, entry_t *pent)
{
        mechlist_t      *pmech = NULL;
        mechlist_t      *phead = NULL;
        mechlist_t *pmech;
        mechlist_t *phead;
        char            *next_token;
        char            *value;
        int             count;
        int             supflag = B_FALSE;
        int             disflag = B_FALSE;
        int             rc = SUCCESS;

        if (strncmp(buf, EF_SUPPORTED, strlen(EF_SUPPORTED)) == 0) {
                supflag = B_TRUE;
        } else if (strncmp(buf, EF_DISABLED, strlen(EF_DISABLED)) == 0) {
                disflag = B_TRUE;
        } else {
                /* should not come here */
                return (FAILURE);
```

```
        }

        if (value = strpbrk(buf, SEP_EQUAL)) {
                value++; /* get rid of = */
        } else {
                cryptodebug("failed to parse the kcf.conf file.");
                return (FAILURE);
        }

        if ((next_token = strtok(value, SEP_COMMA)) == NULL) {
                cryptodebug("failed to parse the kcf.conf file.");
                return (FAILURE);
        }

        if ((pmech = create_mech(next_token)) == NULL) {
                return (FAILURE);
        }

        if (supflag) {
                pent->suplist = phead = pmech;
        } else if (disflag) {
                pent->dislist = phead = pmech;
        }

        count = 1;
        while (next_token) {
                if (next_token = strtok(NULL, SEP_COMMA)) {
                        if ((pmech = create_mech(next_token)) == NULL) {
                                rc = FAILURE;
                                break;
                        }
                        count++;
                        phead->next = pmech;
                        phead = phead->next;
                }
        }

        if (rc == SUCCESS) {
                if (supflag) {
                        pent->sup_count = count;
                } else if (disflag) {
                        pent->dis_count = count;
                }
        } else {
                free_mechlist(phead);
        }

        return (rc);
}



/*
 * Convert a char string containing a line about a provider
 * from kcf.conf into an entry_t structure.
 *
 * See ent2str(), the reverse of this function, for the format of
 * kcf.conf lines.
 * This routine converts a char string into an entry_t structure
 */
static int
interpret(char *buf, entry_t **ppent)
{
        entry_t *pent = NULL;
        entry_t *pent;
        char    *token1;
        char    *token2;
        char    *token3;
        int     rc;

        /* Get provider name */
        if ((token1 = strtok(buf, SEP_COLON)) == NULL) { /* buf is NULL */
                return (FAILURE);
        };

        pent = create_entry(token1);
        pent = malloc(sizeof (entry_t));
        if (pent == NULL) {
                cryptodebug("out of memory.");
                return (FAILURE);
        }
        (void) strlcpy(pent->name, token1, sizeof (pent->name));
        pent->suplist = NULL;
        pent->dislist = NULL;
        pent->sup_count = 0;
        pent->dis_count = 0;

        if ((token2 = strtok(NULL, SEP_SEMICOLON)) == NULL) {
                /* The entry contains a provider name only */
                free_entry(pent);
                return (FAILURE);
        }

        if (strncmp(token2, EF_UNLOAD, strlen(EF_UNLOAD)) == 0) {
                pent->load = B_FALSE; /* cryptoadm unload */
                if ((token2 = strtok(NULL, SEP_SEMICOLON)) == NULL) {
                        /* The entry contains a provider name:unload only */
                        free_entry(pent);
                        return (FAILURE);
                }
        }

        /* need to get token3 first to satisfy nested strtok invocations */
        token3 = strtok(NULL, SEP_SEMICOLON); /* optional */
        token3 = strtok(NULL, SEP_SEMICOLON);

        /* parse supportedlist (or disabledlist if no supportedlist) */
        if ((token2 != NULL) && ((rc = parse_sup_dis_list(token2, pent)) !=
            SUCCESS)) {
        if (token2 && ((rc = parse_dislist(token2, pent)) != SUCCESS)) {
                free_entry(pent);
                return (rc);
        }

        /* parse disabledlist (if there’s a supportedlist) */
        if ((token3 != NULL) && ((rc = parse_sup_dis_list(token3, pent)) !=
            SUCCESS)) {
        if (token3 && ((rc = parse_dislist(token3, pent)) != SUCCESS)) {
                free_entry(pent);
                return (rc);
        }

        *ppent = pent;
        return (SUCCESS);
}


/*
 * Add an entry about a provider from kcf.conf to the end of an entry list.
 * If the entry list pplist is NULL, create the linked list with pent as the
 * first element.
 * Add an entry to the end of an entry list. If the entry list is NULL, will
```

```
 * create an entry list with the pent.
 */
static int
build_entrylist(entry_t *pent, entrylist_t **pplist)
{
        entrylist_t     *pentlist;
        entrylist_t     *pcur = NULL;
        entrylist_t *pcur;

        pentlist = malloc(sizeof (entrylist_t));
        if (pentlist == NULL) {
                cryptodebug("out of memory.");
                return (FAILURE);
        }
        pentlist->pent = pent;
        pentlist->next = NULL;

        if (*pplist) {
                pcur = *pplist;
                while (pcur->next != NULL)
                        pcur = pcur->next;
                pcur->next = pentlist;
        } else { /* empty list */
                *pplist = pentlist;
        }

        return (SUCCESS);
}



/*
 * Find the entry with the "provname" name from the entry list and duplicate
 * it.  Called by getent_kef().
 * it.
 */
static entry_t *
getent(char *provname, entrylist_t *entrylist)
{
        boolean_t       found = B_FALSE;
        entry_t         *pent1 = NULL;

        if ((provname == NULL) || (entrylist == NULL)) {
                return (NULL);
        }

        while (!found && entrylist) {
                if (strcmp(entrylist->pent->name, provname) == 0) {
                        found = B_TRUE;
                        pent1 = entrylist->pent;
                } else {
                        entrylist = entrylist->next;
                }
        }

        if (!found) {
                return (NULL);
        }

        /* duplicate the entry to be returned */
        return (dup_entry(pent1));
}


/*
 * Free memory in entry_t.
 * That is, the supported and disabled lists for a provider
 * from kcf.conf.
 */
void
free_entry(entry_t  *pent)
{
        if (pent == NULL) {
                return;
        } else {
                free_mechlist(pent->suplist);
                free_mechlist(pent->dislist);
                free(pent);
        }
}


/*
 * Free elements in a entrylist_t linked list,
 * which lists providers in kcf.conf.
 */
void
free_entrylist(entrylist_t *entrylist)
{
        entrylist_t *pnext;

        while (entrylist != NULL) {
                pnext = entrylist->next;
                free_entry(entrylist->pent);
                entrylist = pnext;
        }
}


/*
 * Convert an entry to a string.  This routine builds a string for the entry
 * to be inserted in the kcf.conf file.  Based on the content of each entry,
 * the result string can be one of these 6 forms:
 * to be inserted in the config file.  Based on the content of each entry,
 * the result string can be one of the 4 forms:
 *  - name
 *  - name:supportedlist=m1,m2,...,mj
 *  - name:disabledlist=m1,m2,...,mj
 *  - name:supportedlist=m1,...,mj;disabledlist=m1,m2,...,mk
 *
 *  - name:unload;supportedlist=m1,m2,...,mj
 *  - name:unload;disabledlist=m1,m2,...,mj
 *  - name:unload;supportedlist=m1,...,mj;disabledlist=m1,m2,...,mk
 *
 * Note that the caller is responsible for freeing the returned string
 * (with free_entry()).
 * See interpret() for the reverse of this function: converting a string
 * to an entry_t.
 * Note that the caller is responsible for freeing the returned string.
 */
char *
ent2str(entry_t *pent)
{
        char            *buf;
        mechlist_t      *pcur = NULL;
        boolean_t       semicolon_separator = B_FALSE;
        mechlist_t  *phead;
        boolean_t supflag = B_FALSE;


        if (pent == NULL) {
```

```
                return (NULL);
        }

        if ((buf = malloc(BUFSIZ)) == NULL) {
                return (NULL);
        }

        /* convert the provider name */
        if (strlcpy(buf, pent->name, BUFSIZ) >= BUFSIZ) {
                free(buf);
                return (NULL);
        }

        if (!pent->load) { /* add "unload" keyword */
        /* convert the supported list if any */
        phead = pent->suplist;
        if (phead != NULL) {
                supflag = B_TRUE;

                if (strlcat(buf, SEP_COLON, BUFSIZ) >= BUFSIZ) {
                        free(buf);
                        return (NULL);
                }

                if (strlcat(buf, EF_UNLOAD, BUFSIZ) >= BUFSIZ) {
                        free(buf);
                        return (NULL);
                }

                semicolon_separator = B_TRUE;
        }

        /* convert the supported list if any */
        pcur = pent->suplist;
        if (pcur != NULL) {
                if (strlcat(buf,
                    semicolon_separator ? SEP_SEMICOLON : SEP_COLON,
                    BUFSIZ) >= BUFSIZ) {
                        free(buf);
                        return (NULL);
                }

                if (strlcat(buf, EF_SUPPORTED, BUFSIZ) >= BUFSIZ) {
                        free(buf);
                        return (NULL);
                }

                while (pcur != NULL) {
                        if (strlcat(buf, pcur->name, BUFSIZ) >= BUFSIZ) {
                while (phead != NULL) {
                        if (strlcat(buf, phead->name, BUFSIZ) >= BUFSIZ) {
                                free(buf);
                                return (NULL);
                        }

                        pcur = pcur->next;
                        if (pcur != NULL) {
                        phead = phead->next;
                        if (phead != NULL) {
                                if (strlcat(buf, SEP_COMMA, BUFSIZ)
                                    >= BUFSIZ) {
                                        free(buf);
                                        return (NULL);
                                }
                        }
                }
                semicolon_separator = B_TRUE;
        }

        /* convert the disabled list if any */
        pcur = pent->dislist;
        if (pcur != NULL) {
                if (strlcat(buf,
                    semicolon_separator ? SEP_SEMICOLON : SEP_COLON,
                    BUFSIZ) >= BUFSIZ) {
        phead = pent->dislist;
        if (phead != NULL) {
                if (supflag) {
                        if (strlcat(buf, ";disabledlist=", BUFSIZ) >= BUFSIZ) {
                        free(buf);
                        return (NULL);
                }

                if (strlcat(buf, EF_DISABLED, BUFSIZ) >= BUFSIZ) {
                } else {
                        if (strlcat(buf, ":disabledlist=", BUFSIZ) >= BUFSIZ) {
                        free(buf);
                        return (NULL);
                }
                }

                while (pcur != NULL) {
                        if (strlcat(buf, pcur->name, BUFSIZ) >= BUFSIZ) {
                while (phead != NULL) {
                        if (strlcat(buf, phead->name, BUFSIZ) >= BUFSIZ) {
                                free(buf);
                                return (NULL);
                        }

                        pcur = pcur->next;
                        if (pcur != NULL) {
                        phead = phead->next;
                        if (phead != NULL) {
                                if (strlcat(buf, SEP_COMMA, BUFSIZ)
                                    >= BUFSIZ) {
                                        free(buf);
                                        return (NULL);
                                }
                        }
                }
                semicolon_separator = B_TRUE;
        }

        if (strlcat(buf, "\n", BUFSIZ) >= BUFSIZ) {
                free(buf);
                return (NULL);
        }

        return (buf);
}


/*
 * Enable the mechanisms for the provider pointed by *ppent.  If allflag is
 * TRUE, enable all.  Otherwise, enable the mechanisms specified in the 3rd
 * argument "mlist".  The result will be stored in ppent also.
 */
int
enable_mechs(entry_t **ppent, boolean_t allflag, mechlist_t *mlist)
{
        entry_t         *pent;
        mechlist_t      *phead; /* the current and resulting disabled list */
```

```
        mechlist_t      *ptr = NULL;
        mechlist_t      *pcur = NULL;
        mechlist_t *ptr;
        mechlist_t *pcur;
        boolean_t       found;

        pent = *ppent;
        if (pent == NULL) {
                return (FAILURE);
        }

        if (allflag) {
                free_mechlist(pent->dislist);
                pent->dis_count = 0;
                pent->dislist = NULL;
                return (SUCCESS);
        }

        /*
         * for each mechanism in the to-be-enabled mechanism list,
         * -    check if it is in the current disabled list
         * -    if found, delete it from the disabled list
         *      otherwise, give a warning.
         */
        ptr = mlist;
        while (ptr != NULL) {
                found = B_FALSE;
                phead = pcur =  pent->dislist;
                while (!found && pcur) {
                        if (strcmp(pcur->name, ptr->name) == 0) {
                                found = B_TRUE;
                        } else {
                                phead = pcur;
                                pcur = pcur->next;
                        }
                }

                if (found) {
                        if (phead == pcur) {
                                pent->dislist = pent->dislist->next;
                                free(pcur);
                        } else {
                                phead->next = pcur->next;
                                free(pcur);
                        }
                        pent->dis_count--;
                } else {
                        cryptoerror(LOG_STDERR, gettext(
                            "(Warning) %1$s is either enabled already or not "
                            "a valid mechanism for %2$s"), ptr->name,
                            pent->name);
                }
                ptr = ptr->next;
        }

        if (pent->dis_count == 0) {
                pent->dislist = NULL;
        }

        return (SUCCESS);

}


/*
 * Determine if the kernel provider name, path, is a device
 * (that is, it contains a slash character (e.g., "mca/0").
 * If so, it is a hardware provider; otherwise it is a software provider.
 */
boolean_t
is_device(char *path)
{
        if (strchr(path, SEP_SLASH) != NULL) {
                return (B_TRUE);
        } else {
                return (B_FALSE);
        }
}

/*
 * Split a hardware provider name with the "name/inst_num" format into
 * a name and a number (e.g., split "mca/0" into "mca" instance 0).
 * a name and a number.
 */
int
split_hw_provname(char *provname, char *pname, int *inst_num)
{
        char    name[MAXNAMELEN];
        char    *inst_str;

        if (provname == NULL) {
                return (FAILURE);
        }

        (void) strlcpy(name, provname, MAXNAMELEN);
        if (strtok(name, "/") == NULL) {
                return (FAILURE);
        }

        if ((inst_str = strtok(NULL, "/")) == NULL) {
                return (FAILURE);
        }

        (void) strlcpy(pname, name, MAXNAMELEN);
        *inst_num = atoi(inst_str);

        return (SUCCESS);
}


/*
 * Retrieve information from kcf.conf and build a hardware device entry list
 * and a software entry list of kernel crypto providers.
 *
 * This list is usually incomplete, as kernel crypto providers only have to
 * be listed in kcf.conf if a mechanism is disabled (by cryptoadm) or
 * if the kernel provider module is not one of the default kernel providers.
 *
 * The kcf.conf file is available only in the global zone.
 * Retrieve information from kcf.conf and build a device entry list and
 * a software entry list
 */
int
get_kcfconf_info(entrylist_t **ppdevlist, entrylist_t **ppsoftlist)
{
        FILE    *pfile = NULL;
        FILE *pfile;
        char    buffer[BUFSIZ];
        int     len;
        entry_t *pent = NULL;
        int     rc = SUCCESS;
```

```
        if ((pfile = fopen(_PATH_KCF_CONF, "r")) == NULL) {
                cryptodebug("failed to open the kcf.conf file for read only");
                return (FAILURE);
        }

        *ppdevlist = NULL;
        *ppsoftlist = NULL;
        while (fgets(buffer, BUFSIZ, pfile) != NULL) {
                if (buffer[0] == '#' || buffer[0] == ' ' ||
                    buffer[0] == '\n'|| buffer[0] == '\t') {
                        continue;   /* ignore comment lines */
                }

                len = strlen(buffer);
                if (buffer[len - 1] == '\n') { /* get rid of trailing '\n' */
                if (buffer[len-1] == '\n') { /* get rid of trailing '\n' */
                        len--;
                }
                buffer[len] = '\0';

                if ((rc = interpret(buffer,  &pent)) == SUCCESS) {
                        if (is_device(pent->name)) {
                                rc = build_entrylist(pent, ppdevlist);
                        } else {
                                rc = build_entrylist(pent, ppsoftlist);
                        }
                } else {
                        cryptoerror(LOG_STDERR, gettext(
                            "failed to parse configuration."));
                }

                if (rc != SUCCESS) {
                        free_entrylist(*ppdevlist);
                        free_entrylist(*ppsoftlist);
                        free_entry(pent);
                        break;
                }
        }

        (void) fclose(pfile);
        return (rc);
}

/*
 * Retrieve information from admin device and build a device entry list and
 * a software entry list.  This is used where there is no kcf.conf, e.g., the
 * a software entry list.  This is used where there is no kcf.conf, e.g.
 * non-global zone.
 */
int
get_admindev_info(entrylist_t **ppdevlist, entrylist_t **ppsoftlist)
{
        crypto_get_dev_list_t   *pdevlist_kernel = NULL;
        crypto_get_soft_list_t  *psoftlist_kernel = NULL;
        char                    *devname;
        int                     inst_num;
        int                     mcount;
        mechlist_t              *pmech = NULL;
        entry_t                 *pent_dev = NULL, *pent_soft = NULL;
        mechlist_t *pmech;
        entry_t *pent = NULL;
        int                     i;
        char                    *psoftname;
        entrylist_t             *tmp_pdev = NULL;
        entrylist_t             *tmp_psoft = NULL;
        entrylist_t             *phardlist = NULL, *psoftlist = NULL;
        /*
         * Get hardware providers
         */
        if (get_dev_list(&pdevlist_kernel) != SUCCESS) {
                cryptodebug("failed to get hardware provider list from kernel");
                return (FAILURE);
        }

        for (i = 0; i < pdevlist_kernel->dl_dev_count; i++) {
                devname = pdevlist_kernel->dl_devs[i].le_dev_name;
                inst_num = pdevlist_kernel->dl_devs[i].le_dev_instance;
                mcount = pdevlist_kernel->dl_devs[i].le_mechanism_count;

                pmech = NULL;
                if (get_dev_info(devname, inst_num, mcount, &pmech) !=
                    SUCCESS) {
                        cryptodebug(
                            "failed to retrieve the mechanism list for %s/%d.",
                            devname, inst_num);
                        goto fail_out;
                }

                if ((pent_dev = create_entry(devname)) == NULL) {
                if ((pent = malloc(sizeof (entry_t))) == NULL) {
                        cryptodebug("out of memory.");
                        free_mechlist(pmech);
                        goto fail_out;
                }
                pent_dev->suplist = pmech;
                pent_dev->sup_count = mcount;

                if (build_entrylist(pent_dev, &tmp_pdev) != SUCCESS) {
                (void) strlcpy(pent->name, devname, MAXNAMELEN);
                pent->suplist = pmech;
                pent->sup_count = mcount;
                pent->dislist = NULL;
                pent->dis_count = 0;

                if (build_entrylist(pent, &tmp_pdev) != SUCCESS) {
                        goto fail_out;
                }

                /* because incorporated in tmp_pdev */
                pent = NULL;
        }

        free(pdevlist_kernel);
        pdevlist_kernel = NULL;

        /*
         * Get software providers
         */
        if (getzoneid() == GLOBAL_ZONEID) {
                if (get_kcfconf_info(&phardlist, &psoftlist) != SUCCESS) {
                        goto fail_out;
                }
        }

        if (get_soft_list(&psoftlist_kernel) != SUCCESS) {
                cryptodebug("failed to get software provider list from kernel");
                goto fail_out;
        }

        for (i = 0, psoftname = psoftlist_kernel->sl_soft_names;
            i < psoftlist_kernel->sl_soft_count;
```

```
            i++, psoftname = psoftname + strlen(psoftname) + 1) {
                pmech = NULL;
                if (get_soft_info(psoftname, &pmech, phardlist, psoftlist) !=
                    SUCCESS) {
                if (get_soft_info(psoftname, &pmech) != SUCCESS) {
                        cryptodebug(
                            "failed to retrieve the mechanism list for %s.",
                            psoftname);
                        goto fail_out;
                }

                if ((pent_soft = create_entry(psoftname)) == NULL) {
                if ((pent = malloc(sizeof (entry_t))) == NULL) {
                        cryptodebug("out of memory.");
                        free_mechlist(pmech);
                        goto fail_out;
                }
                pent_soft->suplist = pmech;
                pent_soft->sup_count = get_mech_count(pmech);

                if (build_entrylist(pent_soft, &tmp_psoft) != SUCCESS) {
                (void) strlcpy(pent->name, psoftname, MAXNAMELEN);
                pent->suplist = pmech;
                pent->sup_count = get_mech_count(pmech);
                pent->dislist = NULL;
                pent->dis_count = 0;

                if (build_entrylist(pent, &tmp_psoft) != SUCCESS) {
                        goto fail_out;
                }
        }

        free(psoftlist_kernel);
        psoftlist_kernel = NULL;

        *ppdevlist = tmp_pdev;
        *ppsoftlist = tmp_psoft;

        return (SUCCESS);

fail_out:
        if (pent_dev != NULL)
                free_entry(pent_dev);
        if (pent_soft != NULL)
                free_entry(pent_soft);
        if (pent != NULL)
                free_entry(pent);

        free_entrylist(tmp_pdev);
        free_entrylist(tmp_psoft);

        if (pdevlist_kernel != NULL)
                free(pdevlist_kernel);
        if (psoftlist_kernel != NULL)
                free(psoftlist_kernel);

        return (FAILURE);
}

/*
 * Return configuration information for a kernel provider from kcf.conf.
 * For kernel software providers return a enabled list and disabled list.
 * For kernel hardware providers return just a disabled list.
 *
 * Parameters phardlist and psoftlist are supplied by get_kcfconf_info().
 * If NULL, this function calls get_kcfconf_info() internally.
 * Find the entry in the "kcf.conf" file with "provname" as the provider name.
 * Return the entry if found, otherwise return NULL.
 */
entry_t *
getent_kef(char *provname, entrylist_t *phardlist, entrylist_t *psoftlist)
getent_kef(char *provname)
{
        entrylist_t *pdevlist = NULL;
        entrylist_t *psoftlist = NULL;
        entry_t         *pent = NULL;
        boolean_t       memory_allocated = B_FALSE;

        if ((phardlist == NULL) || (psoftlist == NULL)) {
                if (get_kcfconf_info(&phardlist, &psoftlist) != SUCCESS) {
        if (get_kcfconf_info(&pdevlist, &psoftlist) != SUCCESS) {
                        return (NULL);
                }
                memory_allocated = B_TRUE;
        }

        if (is_device(provname)) {
                pent = getent(provname, phardlist);
                pent = getent(provname, pdevlist);
        } else {
                pent = getent(provname, psoftlist);
        }

        if (memory_allocated) {
                free_entrylist(phardlist);
        free_entrylist(pdevlist);
                free_entrylist(psoftlist);
        }

        return (pent);
}

/*
 * Print out the provider name and the mechanism list.
 */
void
print_mechlist(char *provname, mechlist_t *pmechlist)
{
        mechlist_t *ptr = NULL;
        mechlist_t *ptr;

        if (provname == NULL) {
                return;
        }

        (void) printf("%s: ", provname);
        if (pmechlist == NULL) {
                (void) printf(gettext("No mechanisms presented.\n"));
                return;
        }

        ptr = pmechlist;
        while (ptr != NULL) {
                (void) printf("%s", ptr->name);
                ptr = ptr->next;
                if (ptr == NULL) {
                        (void) printf("\n");
                } else {
                        (void) printf(",");
                }
        }
}
```

```
/*
 * Update the kcf.conf file based on the update mode:
 * - If update_mode is MODIFY_MODE, modify the entry with the same name.
 *   If not found, append a new entry to the kcf.conf file.
 * - If update_mode is DELETE_MODE, delete the entry with the same name.
 * - If update_mode is ADD_MODE, append a new entry to the kcf.conf file.
 * Update the kcf.conf file based on the specified entry and the update mode.
 * - If update_mode is MODIFY_MODE or DELETE_MODE, the entry with the same
 *   provider name will be modified or deleted.
 * - If update_mode is ADD_MODE, this must be a hardware provider without
 *   an entry in the kcf.conf file yet.  Need to locate its driver package
 *   bracket and insert an entry into the bracket.
 */
int
update_kcfconf(entry_t *pent, int update_mode)
{
        boolean_t       add_it = B_FALSE;
        boolean_t       delete_it = B_FALSE;
        boolean_t       found_package = B_FALSE;
        boolean_t       found_entry = B_FALSE;
        FILE            *pfile = NULL;
        FILE            *pfile_tmp = NULL;
        FILE    *pfile;
        FILE    *pfile_tmp;
        char            buffer[BUFSIZ];
        char            buffer2[BUFSIZ];
        char    devname[MAXNAMELEN];
        char            tmpfile_name[MAXPATHLEN];
        char            *name;
        char    *str;
        char            *new_str = NULL;
        int     inst_num;
        int             rc = SUCCESS;


        if (pent == NULL) {
                cryptoerror(LOG_STDERR, gettext("internal error."));
                return (FAILURE);
        }

        /* Check the update_mode */
        switch (update_mode) {
        case ADD_MODE:
        if (update_mode == ADD_MODE) {
                add_it = B_TRUE;
                /* FALLTHROUGH */
        case MODIFY_MODE:
                /* Convert the entry a string to add to kcf.conf  */
                /* Get the hardware provider name first */
                if (split_hw_provname(pent->name, devname, &inst_num) ==
                    FAILURE) {
                        return (FAILURE);
                }

                /* Convert the entry to be a string  */
                if ((new_str = ent2str(pent)) == NULL) {
                        return (FAILURE);
                }
                break;
        case DELETE_MODE:
        } else if (update_mode == DELETE_MODE) {
                delete_it = B_TRUE;
                break;
        default:
        } else if (update_mode != MODIFY_MODE) {
                cryptoerror(LOG_STDERR, gettext("internal error."));
                return (FAILURE);
        }


        /* Open the kcf.conf file */
        if ((pfile = fopen(_PATH_KCF_CONF, "r+")) == NULL) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to open %s for write.", _PATH_KCF_CONF);
                return (FAILURE);
        }

        /* Lock the kcf.conf file */
        if (lockf(fileno(pfile), F_TLOCK, 0) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                (void) fclose(pfile);
                return (FAILURE);
        }

        /*
         * Create a temporary file in the /etc/crypto directory to save
         * updated configuration file first.
         */
        (void) strlcpy(tmpfile_name, TMPFILE_TEMPLATE, sizeof (tmpfile_name));
        if (mkstemp(tmpfile_name) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to create a temporary file - %s"),
                    strerror(err));
                (void) fclose(pfile);
                return (FAILURE);
        }

        if ((pfile_tmp = fopen(tmpfile_name, "w")) == NULL) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext("failed to open %s - %s"),
                    tmpfile_name, strerror(err));
                (void) fclose(pfile);
                return (FAILURE);
        }

        /*
         * Loop thru the entire kcf.conf file, insert, modify or delete
         * an entry.
         */
        while (fgets(buffer, BUFSIZ, pfile) != NULL) {
                if (add_it) {
                        /* always keep the current line */
                        if (fputs(buffer, pfile_tmp) == EOF) {
                                err = errno;
                                cryptoerror(LOG_STDERR, gettext(
                                    "failed to write to a temp file: %s."),
                                    strerror(err));
                                rc = FAILURE;
                                break;
                        }

                        /*
                         * If the current position is the beginning of a driver
```

```
                         * package and if the driver name matches the hardware
                         * provider name, then we want to insert the entry
                         * here.
                         */
                        if ((strstr(buffer, HW_DRIVER_STRING) != NULL) &&
                            (strstr(buffer, devname) != NULL)) {
                                found_package = B_TRUE;
                                if (fputs(new_str, pfile_tmp) == EOF) {
                                        err = errno;
                                        cryptoerror(LOG_STDERR, gettext(
                                            "failed to write to a temp file: "
                                            "%s."), strerror(err));
                                        rc = FAILURE;
                                        break;
                                }
                        }
                } else { /* modify or delete */
                        found_entry = B_FALSE;

                        if (!(buffer[0] == '#' || buffer[0] == ' ' ||
                            buffer[0] == '\n'|| buffer[0] == '\t')) {
                                /*
                                 * Get the provider name from this line and
                                 * check if this is the entry to be updated
                                 * or deleted. Note: can not use "buffer"
                                 * directly because strtok will change its
                                 * value.
                                 */
                                (void) strlcpy(buffer2, buffer, BUFSIZ);
                                if ((name = strtok(buffer2, SEP_COLON)) ==
                                    NULL) {
                                        rc = FAILURE;
                                        break;
                                }

                                if (strcmp(pent->name, name) == 0) {
                                        found_entry = B_TRUE;
                                }
                        }

                        if (found_entry && !delete_it) {
                                /*
                                 * This is the entry to be updated; get the
                                 * updated string and place into buffer.
                                 */
                                (void) strlcpy(buffer, new_str, BUFSIZ);
                                free(new_str);
                                if ((str = ent2str(pent)) == NULL) {
                                        rc = FAILURE;
                                        break;
                                } else {
                                        (void) strlcpy(buffer, str, BUFSIZ);
                                        free(str);
                        }
                        }

                        if (!(found_entry && delete_it)) {
                                /* This is the entry to be updated/reserved */
                                if (fputs(buffer, pfile_tmp) == EOF) {
                                        err = errno;
                                        cryptoerror(LOG_STDERR, gettext(
                                            "failed to write to a temp file: "
                                            "%s."), strerror(err));
                                        rc = FAILURE;
                                        break;
                                }
                        }
                }
        }

        if ((!delete_it) && (rc != FAILURE)) {
                if (add_it || !found_entry) {
                        /* append new entry to end of file */
                        if (fputs(new_str, pfile_tmp) == EOF) {
        if (add_it) {
                free(new_str);
        }

        if ((add_it && !found_package) || (rc == FAILURE)) {
                if (add_it && !found_package) {
                        cryptoerror(LOG_STDERR,
                            gettext("failed to update configuration - no "
                            "driver package information."));
                }

                (void) fclose(pfile);
                (void) fclose(pfile_tmp);
                if (unlink(tmpfile_name) != 0) {
                                err = errno;
                                cryptoerror(LOG_STDERR, gettext(
                                    "failed to write to a temp file: %s."),
                                    strerror(err));
                                rc = FAILURE;
                            "(Warning) failed to remove %s: %s"),
                            tmpfile_name, strerror(err));
                        }
                        free(new_str);
                return (FAILURE);
                }
        }

        (void) fclose(pfile);
        if (fclose(pfile_tmp) != 0) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to close %s: %s"), tmpfile_name,
                    strerror(err));
                return (FAILURE);
        }

        /* Copy the temporary file to the kcf.conf file */
        if (rename(tmpfile_name, _PATH_KCF_CONF) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to rename %s to %s: %s", tmpfile,
                    _PATH_KCF_CONF, strerror(err));
                rc = FAILURE;
        } else if (chmod(_PATH_KCF_CONF,
            S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1) {
                err = errno;
                cryptoerror(LOG_STDERR,
                    gettext("failed to update the configuration - %s"),
                    strerror(err));
                cryptodebug("failed to chmod to %s: %s", _PATH_KCF_CONF,
                    strerror(err));
                rc = FAILURE;
        } else {
                rc = SUCCESS;
        }
```

```
        if ((rc == FAILURE) && (unlink(tmpfile_name) != 0)) {
                err = errno;
                cryptoerror(LOG_STDERR, gettext(
                    "(Warning) failed to remove %s: %s"),
                    tmpfile_name, strerror(err));
        }

        return (rc);
}


/*
 * Disable the mechanisms for the provider pointed by *ppent.  If allflag is
 * TRUE, disable all.  Otherwise, disable the mechanisms specified in the
 * dislist argument.  The "infolist" argument contains the mechanism list
 * supported by this provider.
 */
int
disable_mechs(entry_t **ppent, mechlist_t *infolist, boolean_t allflag,
mechlist_t *dislist)
{
        entry_t         *pent;
        mechlist_t      *plist = NULL;
        mechlist_t      *phead = NULL;
        mechlist_t      *pmech = NULL;
        mechlist_t *plist;
        mechlist_t *phead;
        mechlist_t *pmech;
        int             rc = SUCCESS;

        pent = *ppent;
        if (pent == NULL) {
                return (FAILURE);
        }

        if (allflag) {
                free_mechlist(pent->dislist);
                pent->dis_count = get_mech_count(infolist);
                if (!(pent->dislist = dup_mechlist(infolist))) {
                        return (FAILURE);
                } else {
                        return (SUCCESS);
                }
        }

        /*
         * Not disable all. Now loop thru the mechanisms specified in the
         * dislist.  If the mechanism is not supported by the provider,
         * ignore it with a warning.  If the mechanism is disabled already,
         * do nothing. Otherwise, prepend it to the beginning of the disabled
         * list of the provider.
         */
        plist = dislist;
        while (plist != NULL) {
                if (!is_in_list(plist->name, infolist)) {
                        cryptoerror(LOG_STDERR, gettext("(Warning) "
                            "%1$s is not a valid mechanism for %2$s."),
                            plist->name, pent->name);
                } else if (!is_in_list(plist->name, pent->dislist)) {
                        /* Add this mechanism into the disabled list */
                        if ((pmech = create_mech(plist->name)) == NULL) {
                                rc = FAILURE;
                                break;
                        }

                        if (pent->dislist == NULL) {
                                pent->dislist = pmech;
                        } else {
                                phead = pent->dislist;
                                pent->dislist = pmech;
                                pmech->next = phead;
                        }
                        pent->dis_count++;
                }
                plist = plist->next;
        }

        return (rc);
}
______unchanged_portion_omitted_



/*
 * Print out the mechanism policy for a kernel provider that has an entry
 * in the kcf.conf file.
 *
 * The flag has_random is set to B_TRUE if the provider does random
 * numbers. The flag has_mechs is set by the caller to B_TRUE if the provider
 * has some mechanisms.
 *
 * If pent is NULL, the provider doesn’t have a kcf.conf entry.
 */
void
print_kef_policy(char *provname, entry_t *pent, boolean_t has_random,
    boolean_t has_mechs)
print_kef_policy(entry_t *pent, boolean_t has_random, boolean_t has_mechs)
{
        mechlist_t      *ptr = NULL;
        mechlist_t *ptr;
        boolean_t       rnd_disabled = B_FALSE;

        if (pent != NULL) {
        if (pent == NULL) {
                return;
        }

                rnd_disabled = filter_mechlist(&pent->dislist, RANDOM);
                ptr = pent->dislist;
        }

        (void) printf("%s:", provname);
        (void) printf("%s:", pent->name);

        if (has_mechs == B_TRUE) {
                /*
                 * TRANSLATION_NOTE
                 * This code block may need to be modified a bit to avoid
                 * constructing the text message on the fly.
                 */
                (void) printf(gettext(" all mechanisms are enabled"));
                if (ptr != NULL)
                        (void) printf(gettext(", except "));
                while (ptr != NULL) {
                        (void) printf("%s", ptr->name);
                        ptr = ptr->next;
                        if (ptr != NULL)
                                (void) printf(",");
                }
                if (ptr == NULL)
                        (void) printf(".");
        }
```

```
        /*
         * TRANSLATION_NOTE
         * "random" is a keyword and not to be translated.
         */
        if (rnd_disabled)
                (void) printf(gettext(" %s is disabled."), "random");
        else if (has_random)
                (void) printf(gettext(" %s is enabled."), "random");
        (void) printf("\n");
}


/*
 * Check if a kernel software provider is in the kernel.
 *
 * Parameters:
 * provname             Provider name
 * psoftlist_kernel     Optional software provider list.  If NULL, it will be
 *                      obtained from get_soft_list().
 * in_kernel            Set to B_TRUE if device is in the kernel, else B_FALSE
 */
int
check_kernel_for_soft(char *provname, crypto_get_soft_list_t *psoftlist_kernel,
    boolean_t *in_kernel)
check_active_for_soft(char *provname, boolean_t *is_active)
{
        crypto_get_soft_list_t  *psoftlist_kernel = NULL;
        char            *ptr;
        int             i;
        boolean_t       psoftlist_allocated = B_FALSE;

        if (provname == NULL) {
                cryptoerror(LOG_STDERR, gettext("internal error."));
                return (FAILURE);
        }

        if (psoftlist_kernel == NULL) {
                if (get_soft_list(&psoftlist_kernel) == FAILURE) {
                        cryptodebug("failed to get the software provider list"
                        " from kernel.");
                cryptodebug("failed to get the software provider list from"
                    "kernel.");
                        return (FAILURE);
                }
                psoftlist_allocated = B_TRUE;
        }

        *in_kernel = B_FALSE;
        *is_active = B_FALSE;
        ptr = psoftlist_kernel->sl_soft_names;
        for (i = 0; i < psoftlist_kernel->sl_soft_count; i++) {
                if (strcmp(provname, ptr) == 0) {
                        *in_kernel = B_TRUE;
                        *is_active = B_TRUE;
                        break;
                }
                ptr = ptr + strlen(ptr) + 1;
        }

        if (psoftlist_allocated)
                free(psoftlist_kernel);

        return (SUCCESS);
}
```

```
/*
 * Check if a kernel hardware provider is in the kernel.
 *
 * Parameters:
 * provname     Provider name
 * pdevlist     Optional Hardware Crypto Device List.  If NULL, it will be
 *              obtained from get_dev_list().
 * in_kernel    Set to B_TRUE if device is in the kernel, otherwise B_FALSE
 */
int
check_kernel_for_hard(char *provname,
    crypto_get_dev_list_t *pdevlist, boolean_t *in_kernel)
check_active_for_hard(char *provname, boolean_t *is_active)
{
        crypto_get_dev_list_t   *pdevlist = NULL;
        char            devname[MAXNAMELEN];
        int             inst_num;
        int             i;
        boolean_t       dev_list_allocated = B_FALSE;

        if (provname == NULL) {
                cryptoerror(LOG_STDERR, gettext("internal error."));
                return (FAILURE);
        }

        if (split_hw_provname(provname, devname, &inst_num) == FAILURE) {
                return (FAILURE);
        }

        if (pdevlist == NULL) {
                if (get_dev_list(&pdevlist) == FAILURE) {
                        cryptoerror(LOG_STDERR, gettext("internal error."));
                        return (FAILURE);
                }
                dev_list_allocated = B_TRUE;
        }

        *in_kernel = B_FALSE;
        *is_active = B_FALSE;
        for (i = 0; i < pdevlist->dl_dev_count; i++) {
                if ((strcmp(pdevlist->dl_devs[i].le_dev_name, devname) == 0) &&
                    (pdevlist->dl_devs[i].le_dev_instance == inst_num)) {
                        *in_kernel = B_TRUE;
                        *is_active = B_TRUE;
                        break;
                }
        }

        if (dev_list_allocated)
                free(pdevlist);

        return (SUCCESS);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   42338 Tue Oct 28 16:45:32 2008
new/usr/src/cmd/cmd-crypto/cryptoadm/cryptoadm.c
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
______unchanged_portion_omitted_

/*
 * Get the provider type.  This function returns
 * - PROV_UEF_LIB if provname contains an absolute path name
 * - PROV_KEF_SOFT if provname is a base name only (e.g., "aes").
 * - PROV_KEF_SOFT if provname is a base name only
 * - PROV_KEF_HARD if provname contains one slash only and the slash is not
 *      the 1st character (e.g., "mca/0").
 *      the 1st character.
 * - PROV_BADNAME otherwise.
 */
static int
get_provider_type(char *provname)
{
        char *pslash1;
        char *pslash2;

        if (provname == NULL) {
                return (FAILURE);
        }

        if (provname[0] == '/') {
                return (PROV_UEF_LIB);
        } else if ((pslash1 = strchr(provname, SEP_SLASH)) == NULL) {
                /* no slash */
                return (PROV_KEF_SOFT);
        } else {
                pslash2 = strrchr(provname, SEP_SLASH);
                if (pslash1 == pslash2) {
                        return (PROV_KEF_HARD);
                } else {
                        return (PROV_BADNAME);
                }
        }
}
______unchanged_portion_omitted_

/*
 * The top level function for the "cryptoadm list" subcommand and options.
 * The top level function for the list subcommand and options.
 */
static int
do_list(int argc, char **argv)
{
        boolean_t               mflag = B_FALSE;
        boolean_t               pflag = B_FALSE;
        boolean_t               vflag = B_FALSE;
        char                    ch;
        cryptoadm_provider_t    *prov = NULL;
        int                     rc = SUCCESS;

        argc -= 1;
        argv += 1;

        if (argc == 1) {
                rc = list_simple_for_all(B_FALSE);
                goto out;
        }

        /*
         * cryptoadm list [-v] [-m] [-p] [provider=<>] [mechanism=<>]
         * [-v] [-m] [-p] [provider=<>] [mechanism=<>]
```

```
         */
        if (argc > 5) {
                usage();
                return (rc);
        }

        while ((ch = getopt(argc, argv, "mpv")) != EOF) {
                switch (ch) {
                case 'm':
                        mflag = B_TRUE;
                        if (pflag) {
                                rc = ERROR_USAGE;
                        }
                        break;
                case 'p':
                        pflag = B_TRUE;
                        if (mflag || vflag) {
                                rc = ERROR_USAGE;
                        }
                        break;
                case 'v':
                        vflag = B_TRUE;
                        if (pflag)
                                rc = ERROR_USAGE;
                        break;
                default:
                        rc = ERROR_USAGE;
                        break;
                }
        }

        if (rc == ERROR_USAGE) {
                usage();
                return (rc);
        }

        if ((rc = process_feature_operands(argc, argv)) != SUCCESS) {
                goto out;
        }

        prov = get_provider(argc, argv);

        if (mflag || vflag) {
                if (argc > 0) {
                        rc = process_mech_operands(argc, argv, B_TRUE);
                        if (rc == FAILURE)
                                goto out;
                        /* "-m" is implied when a mechanism list is given */
                        if (mecharglist != NULL || allflag)
                                mflag = B_TRUE;
                }
        }

        if (prov == NULL) {
                if (mflag) {
                        rc = list_mechlist_for_all(vflag);
                } else if (pflag) {
                        rc = list_policy_for_all();
                } else if (vflag) {
                        rc = list_simple_for_all(vflag);
                }
        } else if (prov->cp_type == METASLOT) {
                if ((!mflag) && (!vflag) && (!pflag)) {
                        /* no flag is specified, just list metaslot status */
                        rc = list_metaslot_info(mflag, vflag, mecharglist);
                } else if (mflag || vflag) {
                        rc = list_metaslot_info(mflag, vflag, mecharglist);
                } else if (pflag) {
                        rc = list_metaslot_policy();
                } else {
                        /* error message */
                        usage();
                        rc = ERROR_USAGE;
                }
        } else if (prov->cp_type == PROV_BADNAME) {
                usage();
                rc = ERROR_USAGE;
                goto out;
        } else { /* do the listing for a provider only */
                char    *provname = prov->cp_name;

                if (mflag || vflag) {
                        if (vflag)
                                (void) printf(gettext("Provider: %s\n"),
                                    provname);
                                        prov->cp_name);
                        switch (prov->cp_type) {
                        case PROV_UEF_LIB:
                                rc = list_mechlist_for_lib(provname,
                                    mecharglist, NULL, B_FALSE, vflag, mflag);
                                rc = list_mechlist_for_lib(prov->cp_name,
                                        mecharglist, NULL, B_FALSE,
                                        vflag, mflag);
                                break;
                        case PROV_KEF_SOFT:
                                rc = list_mechlist_for_soft(provname,
                                    NULL, NULL);
                                rc = list_mechlist_for_soft(prov->cp_name);
                                break;
                        case PROV_KEF_HARD:
                                rc = list_mechlist_for_hard(provname);
                                rc = list_mechlist_for_hard(prov->cp_name);
                                break;
                        default: /* should not come here */
                                rc = FAILURE;
                                break;
                        }
                } else if (pflag) {
                        switch (prov->cp_type) {
                        case PROV_UEF_LIB:
                                rc = list_policy_for_lib(provname);
                                rc = list_policy_for_lib(prov->cp_name);
                                break;
                        case PROV_KEF_SOFT:
                                if (getzoneid() == GLOBAL_ZONEID) {
                                        rc = list_policy_for_soft(provname,
                                            NULL, NULL);
                                        rc = list_policy_for_soft(
                                            prov->cp_name);
                                } else {
                                        /*
                                         * TRANSLATION_NOTE
                                         * "global" is keyword and not to
                                         * be translated.
                                         */
                                        cryptoerror(LOG_STDERR, gettext(
                                            "policy information for kernel "
                                            "providers is available "
                                            "in the %s zone only"), "global");
                                        rc = FAILURE;
                                }
                                break;
```

```
                        case PROV_KEF_HARD:
                                if (getzoneid() == GLOBAL_ZONEID) {
                                        rc = list_policy_for_hard(
                                            provname, NULL, NULL, NULL);
                                            prov->cp_name);
                                } else {
                                        /*
                                         * TRANSLATION_NOTE
                                         * "global" is keyword and not to
                                         * be translated.
                                         */
                                        cryptoerror(LOG_STDERR, gettext(
                                            "policy information for kernel "
                                            "providers is available "
                                            "in the %s zone only"), "global");
                                        rc = FAILURE;
                                }

                                break;
                        default: /* should not come here */
                                rc = FAILURE;
                                break;
                        }
                } else {
                        /* error message */
                        usage();
                        rc = ERROR_USAGE;
                }
        }

out:
        if (prov != NULL)
                free(prov);

        if (mecharglist != NULL)
                free_mechlist(mecharglist);
        return (rc);
}


/*
 * The top level function for the "cryptoadm disable" subcommand.
 * The top level function for the disable subcommand.
 */
static int
do_disable(int argc, char **argv)
{
        cryptoadm_provider_t    *prov = NULL;
        int                     rc = SUCCESS;
        boolean_t               auto_key_migrate_flag = B_FALSE;

        if ((argc < 3) || (argc > 5)) {
                usage();
                return (ERROR_USAGE);
        }

        prov = get_provider(argc, argv);
        if (prov == NULL) {
                usage();
                return (ERROR_USAGE);
        }
        if (prov->cp_type == PROV_BADNAME) {
                return (FAILURE);
        }

        if ((rc = process_feature_operands(argc, argv)) != SUCCESS) {
                goto out;
        }

        /*
         * If allflag or rndflag has already been set there is no reason to
         * process mech=
         */
        if (prov->cp_type == METASLOT) {
                if ((argc > 3) &&
                    (rc = process_metaslot_operands(argc, argv,
                    NULL, NULL, NULL, &auto_key_migrate_flag)) != SUCCESS) {
                        usage();
                        return (rc);
                }
        } else if (!allflag && !rndflag &&
            (rc = process_mech_operands(argc, argv, B_FALSE)) != SUCCESS) {
                        return (rc);
        }

        switch (prov->cp_type) {
        case METASLOT:
                rc = disable_metaslot(mecharglist, allflag,
                    auto_key_migrate_flag);
                break;
        case PROV_UEF_LIB:
                rc = disable_uef_lib(prov->cp_name, rndflag, allflag,
                    mecharglist);
                break;
        case PROV_KEF_SOFT:
                if (rndflag && !allflag) {
                        if ((mecharglist = create_mech(RANDOM)) == NULL) {
                                rc = FAILURE;
                                break;
                        }
                }
                if (getzoneid() == GLOBAL_ZONEID) {
                        rc = disable_kef_software(prov->cp_name, rndflag,
                            allflag, mecharglist);
                } else {
                        /*
                         * TRANSLATION_NOTE
                         * "disable" could be either a literal keyword
                         * and hence not to be translated, or a verb and
                         * translatable.  A choice was made to view it as
                         * a literal keyword.  "global" is keyword and not
                         * to be translated.
                         */
                        cryptoerror(LOG_STDERR, gettext("%1$s for kernel "
                            "providers is supported in the %2$s zone only"),
                            "disable", "global");
                        rc = FAILURE;
                }
                break;
        case PROV_KEF_HARD:
                if (rndflag && !allflag) {
                        if ((mecharglist = create_mech(RANDOM)) == NULL) {
                                rc = FAILURE;
                                break;
                        }
                }
                if (getzoneid() == GLOBAL_ZONEID) {
                        rc = disable_kef_hardware(prov->cp_name, rndflag,
                            allflag, mecharglist);
                } else {
                        /*
                         * TRANSLATION_NOTE
```

```
                         * "disable" could be either a literal keyword
                         * and hence not to be translated, or a verb and
                         * translatable.  A choice was made to view it as
                         * a literal keyword.  "global" is keyword and not
                         * to be translated.
                         */
                        cryptoerror(LOG_STDERR, gettext("%1$s for kernel "
                            "providers is supported in the %2$s zone only"),
                            "disable", "global");
                        rc = FAILURE;
                }
                break;
        default: /* should not come here */
                rc = FAILURE;
                break;
        }

out:
        free(prov);
        if (mecharglist != NULL) {
                free_mechlist(mecharglist);
        }
        return (rc);
}


/*
 * The top level function for the "cryptoadm enable" subcommand.
 * The top level function fo the enable subcommand.
 */
static int
do_enable(int argc, char **argv)
{
        cryptoadm_provider_t    *prov = NULL;
        int                     rc = SUCCESS;
        char                    *alt_token = NULL, *alt_slot = NULL;
        boolean_t               use_default = B_FALSE;
        boolean_t               auto_key_migrate_flag = B_FALSE;
        boolean_t use_default = B_FALSE, auto_key_migrate_flag = B_FALSE;

        if ((argc < 3) || (argc > 6)) {
                usage();
                return (ERROR_USAGE);
        }

        prov = get_provider(argc, argv);
        if (prov == NULL) {
                usage();
                return (ERROR_USAGE);
        }
        if ((prov->cp_type != METASLOT) && (argc != 4)) {
                usage();
                return (ERROR_USAGE);
        }
        if (prov->cp_type == PROV_BADNAME) {
                rc = FAILURE;
                goto out;
        }


        if (prov->cp_type == METASLOT) {
                if ((rc = process_metaslot_operands(argc, argv, &alt_token,
                    &alt_slot, &use_default, &auto_key_migrate_flag))
                    != SUCCESS) {
                        usage();
                        goto out;
```

```
		}
		if ((alt_slot || alt_token) && use_default) {
			usage();
			rc = FAILURE;
			goto out;
		}
	} else {
		if ((rc = process_feature_operands(argc, argv)) != SUCCESS) {
			goto out;
		}

		/*
		 * If allflag or rndflag has already been set there is
		 * no reason to process mech=
		 */
		if (!allflag && !rndflag &&
		    (rc = process_mech_operands(argc, argv, B_FALSE))
		    != SUCCESS) {
			goto out;
		}
	}

	switch (prov->cp_type) {
	case METASLOT:
		rc = enable_metaslot(alt_token, alt_slot, use_default,
		    mecharglist, allflag, auto_key_migrate_flag);
		break;
	case PROV_UEF_LIB:
		rc = enable_uef_lib(prov->cp_name, rndflag, allflag,
		    mecharglist);
		break;
	case PROV_KEF_SOFT:
	case PROV_KEF_HARD:
		if (rndflag && !allflag) {
			if ((mecharglist = create_mech(RANDOM)) == NULL) {
				rc = FAILURE;
				break;
			}
		}
		if (getzoneid() == GLOBAL_ZONEID) {
			rc = enable_kef(prov->cp_name, rndflag, allflag,
			    mecharglist);
		} else {
			/*
			 * TRANSLATION_NOTE
			 * "enable" could be either a literal keyword
			 * and hence not to be translated, or a verb and
			 * translatable.  A choice was made to view it as
			 * a literal keyword.  "global" is keyword and not
			 * to be translated.
			 */
			cryptoerror(LOG_STDERR, gettext("%1$s for kernel "
			    "providers is supported in the %2$s zone only"),
			    "enable", "global");
			rc = FAILURE;
		}
		break;
	default: /* should not come here */
		rc = FAILURE;
		break;
	}
out:
	free(prov);
	if (mecharglist != NULL) {
		free_mechlist(mecharglist);
	}
```

```
	if (alt_token != NULL) {
		free(alt_token);
	}
	if (alt_slot != NULL) {
		free(alt_slot);
	}
	return (rc);
}



/*
 * The top level function for the "cryptoadm install" subcommand.
 * The top level function fo the install subcommand.
 */
static int
do_install(int argc, char **argv)
{
	cryptoadm_provider_t	*prov = NULL;
	int	rc;

	if (argc < 3) {
		usage();
		return (ERROR_USAGE);
	}

	prov = get_provider(argc, argv);
	if (prov == NULL ||
	    prov->cp_type == PROV_BADNAME || prov->cp_type == PROV_KEF_HARD) {
		/*
		 * TRANSLATION_NOTE
		 * "install" could be either a literal keyword and hence
		 * not to be translated, or a verb and translatable.  A
		 * choice was made to view it as a literal keyword.
		 */
		cryptoerror(LOG_STDERR,
		    gettext("bad provider name for %s."), "install");
		rc = FAILURE;
		goto out;
	}

	if (prov->cp_type == PROV_UEF_LIB) {
		rc = install_uef_lib(prov->cp_name);
		goto out;
	}

	/* It is the PROV_KEF_SOFT type now  */

	/* check if there are mechanism operands */
	if (argc < 4) {
		/*
		 * TRANSLATION_NOTE
		 * "mechanism" could be either a literal keyword and hence
		 * not to be translated, or a descriptive word and
		 * translatable.  A choice was made to view it as a literal
		 * keyword.
		 */
		cryptoerror(LOG_STDERR,
		    gettext("need %s operands for installing a"
		    " kernel software provider."), "mechanism");
		rc = ERROR_USAGE;
		goto out;
	}

	if ((rc = process_mech_operands(argc, argv, B_FALSE)) != SUCCESS) {
		goto out;
```

```
        }

        if (allflag == B_TRUE) {
                /*
                 * TRANSLATION_NOTE
                 * "all", "mechanism", and "install" are all keywords and
                 * not to be translated.
                 */
                cryptoerror(LOG_STDERR,
                    gettext("can not use the %1$s keyword for %2$s "
                    "in the %3$s subcommand."), "all", "mechanism", "install");
                rc = ERROR_USAGE;
                goto out;
        }

        if (getzoneid() == GLOBAL_ZONEID) {
                rc = install_kef(prov->cp_name, mecharglist);
        } else {
                /*
                 * TRANSLATION_NOTE
                 * "install" could be either a literal keyword and hence
                 * not to be translated, or a verb and translatable.  A
                 * choice was made to view it as a literal keyword.
                 * "global" is keyword and not to be translated.
                 */
                cryptoerror(LOG_STDERR, gettext("%1$s for kernel providers "
                    "is supported in the %2$s zone only"), "install", "global");
                rc = FAILURE;
        }
out:
        free(prov);
        return (rc);
}



/*
 * The top level function for the "cryptoadm uninstall" subcommand.
 * The top level function for the uninstall subcommand.
 */
static int
do_uninstall(int argc, char **argv)
{
        cryptoadm_provider_t    *prov = NULL;
        int     rc = SUCCESS;

        if (argc != 3) {
                usage();
                return (ERROR_USAGE);
        }

        prov = get_provider(argc, argv);
        if (prov == NULL ||
            prov->cp_type == PROV_BADNAME || prov->cp_type == PROV_KEF_HARD) {
                /*
                 * TRANSLATION_NOTE
                 * "uninstall" could be either a literal keyword and hence
                 * not to be translated, or a verb and translatable.  A
                 * choice was made to view it as a literal keyword.
                 */
                cryptoerror(LOG_STDERR,
                    gettext("bad provider name for %s."), "uninstall");
                free(prov);
                return (FAILURE);
        }
```

```
        if (prov->cp_type == PROV_UEF_LIB) {
                rc = uninstall_uef_lib(prov->cp_name);

        } else if (prov->cp_type == PROV_KEF_SOFT) {
                if (getzoneid() == GLOBAL_ZONEID) {
                        /* unload and remove from kcf.conf */
                        rc = uninstall_kef(prov->cp_name);
                } else {
                        /*
                         * TRANSLATION_NOTE
                         * "uninstall" could be either a literal keyword and
                         * hence not to be translated, or a verb and
                         * translatable.  A choice was made to view it as a
                         * literal keyword.  "global" is keyword and not to
                         * be translated.
                         */
                        cryptoerror(LOG_STDERR, gettext("%1$s for kernel "
                            "providers is supported in the %2$s zone only"),
                            "uninstall", "global");
                        rc = FAILURE;
                }
        }

        free(prov);
        return (rc);
}


/*
 * The top level function for the "cryptoadm unload" subcommand.
 * The top level function for the unload subcommand.
 */
static int
do_unload(int argc, char **argv)
{
        cryptoadm_provider_t    *prov = NULL;
        entry_t                 *pent = NULL;
        boolean_t               in_kernel = B_FALSE;
        entry_t *pent;
        boolean_t       is_active;
        int                     rc = SUCCESS;
        char                    *provname = NULL;

        if (argc != 3) {
                usage();
                return (ERROR_USAGE);
        }

        /* check if it is a kernel software provider */
        prov = get_provider(argc, argv);
        if (prov == NULL) {
                cryptoerror(LOG_STDERR,
                    gettext("unable to determine provider name."));
                goto out;
        }
        provname = prov->cp_name;
        if (prov->cp_type != PROV_KEF_SOFT) {
                cryptoerror(LOG_STDERR,
                    gettext("%s is not a valid kernel software provider."),
                    provname);
                    prov->cp_name);
                rc = FAILURE;
                goto out;
        }

        if (getzoneid() != GLOBAL_ZONEID) {
                /*
                 * TRANSLATION_NOTE
                 * "unload" could be either a literal keyword and hence
                 * not to be translated, or a verb and translatable.
                 * A choice was made to view it as a literal keyword.
                 * "global" is keyword and not to be translated.
                 */
                cryptoerror(LOG_STDERR, gettext("%1$s for kernel providers "
                    "is supported in the %2$s zone only"), "unload", "global");
                rc = FAILURE;
                goto out;
        }

        if (check_kernel_for_soft(provname, NULL, &in_kernel) == FAILURE) {
                cryptodebug("internal error");
                rc = FAILURE;
                goto out;
        } else if (in_kernel == B_FALSE) {
        /* Check if it is in the kcf.conf file first */
        if ((pent = getent_kef(prov->cp_name)) == NULL) {
                cryptoerror(LOG_STDERR,
                    gettext("provider %s is not loaded or does not exist."),
                    provname);
                    gettext("provider %s does not exist."), prov->cp_name);
                rc = FAILURE;
                goto out;
        }
        free_entry(pent);

        /* Get kcf.conf entry.  If none, build a new entry */
        if ((pent = getent_kef(provname, NULL, NULL)) == NULL) {
                if ((pent = create_entry(provname)) == NULL) {
                        cryptoerror(LOG_STDERR, gettext("out of memory."));
        /* If it is unloaded already, return  */
        if (check_active_for_soft(prov->cp_name, &is_active) == FAILURE) {
                cryptodebug("internal error");
                cryptoerror(LOG_STDERR,
                    gettext("failed to unload %s."), prov->cp_name);
                        rc = FAILURE;
                        goto out;
                }
        }

        /* If it is unloaded already, return  */
        if (!pent->load) { /* unloaded already */
                cryptoerror(LOG_STDERR,
                    gettext("failed to unload %s."), provname);
                rc = FAILURE;
        if (is_active == B_FALSE) { /* unloaded already */
                rc = SUCCESS;
                goto out;
        } else if (unload_kef_soft(provname) != FAILURE) {
                /* Mark as unloaded in kcf.conf */
                pent->load = B_FALSE;
                rc = update_kcfconf(pent, MODIFY_MODE);
        } else {
        } else if (unload_kef_soft(prov->cp_name, B_TRUE) == FAILURE) {
                cryptoerror(LOG_STDERR,
                    gettext("failed to unload %s."), provname);
                    gettext("failed to unload %s."), prov->cp_name);
                rc = FAILURE;
        } else {
                rc = SUCCESS;
        }
out:
        free(prov);
```

```
        free_entry(pent);
        return (rc);
}



/*
 * The top level function for the "cryptoadm refresh" subcommand.
 * The top level function for the refresh subcommand.
 */
static int
do_refresh(int argc)
{
        if (argc != 2) {
                usage();
                return (ERROR_USAGE);
        }

        if (getzoneid() == GLOBAL_ZONEID) {
                return (refresh());
        } else { /* non-global zone */
                /*
                 * Note:  in non-global zone, this must silently return SUCCESS
                 * due to integration with SMF, for "svcadm refresh cryptosvc"
                 */
        if (getzoneid() != GLOBAL_ZONEID)
                return (SUCCESS);
        }

        return (refresh());
}


/*
 * The top level function for the "cryptoadm start" subcommand.
 * The top level function for the start subcommand.
 */
static int
do_start(int argc)
{
        int ret;

        if (argc != 2) {
                usage();
                return (ERROR_USAGE);
        }

        ret = do_refresh(argc);
        if (ret != SUCCESS)
                return (ret);

        return (start_daemon());
}

/*
 * The top level function for the "cryptoadm stop" subcommand.
 * The top level function for the stop subcommand.
 */
static int
do_stop(int argc)
{
        if (argc != 2) {
                usage();
                return (ERROR_USAGE);
        }
        return (stop_daemon());
}



/*
 * Print a list all the the providers.
 * Called for "cryptoadm list" or "cryptoadm list -v" (no -m or -p).
 * List all the providers.
 */
static int
list_simple_for_all(boolean_t verbose)
{
        uentrylist_t            *pliblist = NULL;
        uentrylist_t            *plibptr = NULL;
        entry_t                 *pent = NULL;
        uentrylist_t    *pliblist;
        uentrylist_t    *plibptr;
        entrylist_t     *pdevlist_conf;
        entrylist_t     *psoftlist_conf;
        entrylist_t     *pdevlist_zone;
        entrylist_t     *psoftlist_zone;
        entrylist_t     *ptr;
        crypto_get_dev_list_t   *pdevlist_kernel = NULL;
        int                     rc = SUCCESS;
        boolean_t       is_active;
        int     ru = SUCCESS;
        int     rs = SUCCESS;
        int     rd = SUCCESS;
        int                     i;

        /* get user-level providers */
        (void) printf(gettext("\nUser-level providers:\n"));
        if (get_pkcs11conf_info(&pliblist) != SUCCESS) {
                cryptoerror(LOG_STDERR, gettext(
                    "failed to retrieve the list of user-level providers."));
                rc = FAILURE;
                ru = FAILURE;
        }

        for (plibptr = pliblist; plibptr != NULL; plibptr = plibptr->next) {
        plibptr = pliblist;
        while (plibptr != NULL) {
                if (strcmp(plibptr->puent->name, METASLOT_KEYWORD) != 0) {
                        (void) printf(gettext("Provider: %s\n"),
                            plibptr->puent->name);
                        if (verbose) {
                                (void) list_mechlist_for_lib(
                                    plibptr->puent->name, mecharglist, NULL,
                                    B_FALSE, verbose, B_FALSE);
                                (void) printf("\n");
                        }
                }
                plibptr = plibptr->next;
        }
        free_uentrylist(pliblist);

        /* get kernel software providers */
        (void) printf(gettext("\nKernel software providers:\n"));

        if (getzoneid() == GLOBAL_ZONEID) {
                /* get kernel software providers from kernel ioctl */
                crypto_get_soft_list_t          *psoftlist_kernel = NULL;
                uint_t                          sl_soft_count;
                char                            *psoftname;
                entrylist_t                     *pdevlist_conf = NULL;
```

```
                entrylist_t                     *psoftlist_conf = NULL;
                /* use kcf.conf for kernel software providers in global zone */
                pdevlist_conf = NULL;
                psoftlist_conf = NULL;

                if (get_soft_list(&psoftlist_kernel) == FAILURE) {
                        cryptoerror(LOG_ERR, gettext("Failed to retrieve the "
                            "software provider list from kernel."));
                        rc = FAILURE;
                } else {
                        sl_soft_count = psoftlist_kernel->sl_soft_count;
                if (get_kcfconf_info(&pdevlist_conf, &psoftlist_conf) !=
                    SUCCESS) {
                        cryptoerror(LOG_STDERR,
                            gettext("failed to retrieve the "
                            "list of kernel software providers.\n"));
                        rs = FAILURE;
                }

                        if (get_kcfconf_info(&pdevlist_conf, &psoftlist_conf)
                ptr = psoftlist_conf;
                while (ptr != NULL) {
                        if (check_active_for_soft(ptr->pent->name, &is_active)
                            == FAILURE) {
                                cryptoerror(LOG_ERR,
                                    "failed to retrieve the providers' "
                                    "information from file kcf.conf - %s.",
                                    _PATH_KCF_CONF);
                                free(psoftlist_kernel);
                                rc = FAILURE;
                        } else {
                                rs = FAILURE;
                                cryptoerror(LOG_STDERR, gettext("failed to "
                                    "get the state of a kernel software "
                                    "providers.\n"));
                                break;
                        }

                                for (i = 0,
                                    psoftname = psoftlist_kernel->sl_soft_names;
                                    i < sl_soft_count;
                                    ++i, psoftname += strlen(psoftname) + 1) {
                                        pent = getent_kef(psoftname,
                                            pdevlist_conf, psoftlist_conf);
                                        (void) printf("\t%s%s\n", psoftname,
                                            (pent == NULL) || (pent->load) ?
                                            "" : gettext(" (inactive)"));
                        (void) printf("\t%s", ptr->pent->name);
                        if (is_active == B_FALSE) {
                                (void) printf(gettext(" (inactive)\n"));
                        } else {
                                (void) printf("\n");
                                }
                                free_entrylist(pdevlist_conf);
                                free_entrylist(psoftlist_conf);
                        ptr = ptr->next;
                        }
                        free(psoftlist_kernel);
                }

                free_entrylist(pdevlist_conf);
                free_entrylist(psoftlist_conf);
        } else {
                /* kcf.conf not there in non-global zone, use /dev/cryptoadm */
                entrylist_t     *pdevlist_zone = NULL;
                entrylist_t     *psoftlist_zone = NULL;
                entrylist_t     *ptr;
                pdevlist_zone = NULL;
                psoftlist_zone = NULL;

                if (get_admindev_info(&pdevlist_zone, &psoftlist_zone) !=
                    SUCCESS) {
                        cryptoerror(LOG_STDERR,
                            gettext("failed to retrieve the "
                            "list of kernel software providers.\n"));
                        rc = FAILURE;
                        rs = FAILURE;
                }

                ptr = psoftlist_zone;
                while (ptr != NULL) {
                        (void) printf("\t%s\n", ptr->pent->name);
                        ptr = ptr->next;
                }

                free_entrylist(pdevlist_zone);
                free_entrylist(psoftlist_zone);
        }

        /* get kernel hardware providers */
        (void) printf(gettext("\nKernel hardware providers:\n"));
        if (get_dev_list(&pdevlist_kernel) == FAILURE) {
                cryptoerror(LOG_STDERR, gettext("failed to retrieve "
                    "the list of kernel hardware providers.\n"));
                rc = FAILURE;
                rd = FAILURE;
        } else {
                for (i = 0; i < pdevlist_kernel->dl_dev_count; i++) {
                        (void) printf("\t%s/%d\n",
                            pdevlist_kernel->dl_devs[i].le_dev_name,
                            pdevlist_kernel->dl_devs[i].le_dev_instance);
                }
        }
        free(pdevlist_kernel);

        return (rc);
        if (ru == FAILURE || rs == FAILURE || rd == FAILURE) {
                return (FAILURE);
        } else {
                return (SUCCESS);
        }
}



/*
 * List all the providers. And for each provider, list the mechanism list.
 * Called for "cryptoadm list -m" or "cryptoadm list -mv" .
 */
static int
list_mechlist_for_all(boolean_t verbose)
{
        crypto_get_dev_list_t   *pdevlist_kernel = NULL;
        uentrylist_t            *pliblist = NULL;
        uentrylist_t            *plibptr = NULL;
        entry_t                 *pent = NULL;
        mechlist_t              *pmechlist = NULL;
        crypto_get_dev_list_t   *pdevlist_kernel;
        uentrylist_t    *pliblist;
        uentrylist_t    *plibptr;
        entrylist_t     *pdevlist_conf;
        entrylist_t     *psoftlist_conf;
```

```
        entrylist_t     *pdevlist_zone;
        entrylist_t     *psoftlist_zone;
        entrylist_t     *ptr;
        mechlist_t      *pmechlist;
        boolean_t       is_active;
        char                    provname[MAXNAMELEN];
        char                    devname[MAXNAMELEN];
        int                     inst_num;
        int                     count;
        int                     i;
        int                     rv;
        int                     rc = SUCCESS;

        /* get user-level providers */
        (void) printf(gettext("\nUser-level providers:\n"));
        /*
         * TRANSLATION_NOTE
         * Strictly for appearance's sake, this line should be as long as
         * the length of the translated text above.
         */
        (void) printf(gettext("=====================\n"));
        if (get_pkcs11conf_info(&pliblist) != SUCCESS) {
                cryptoerror(LOG_STDERR, gettext("failed to retrieve "
                    "the list of user-level providers.\n"));
                rc = FAILURE;
        }

        plibptr = pliblist;
        while (plibptr != NULL) {
                /* skip metaslot entry */
                if (strcmp(plibptr->puent->name, METASLOT_KEYWORD) != 0) {
                        (void) printf(gettext("\nProvider: %s\n"),
                            plibptr->puent->name);
                        rv = list_mechlist_for_lib(plibptr->puent->name,
                            mecharglist, NULL, B_FALSE, verbose, B_TRUE);
                        if (rv == FAILURE) {
                                rc = FAILURE;
                        }
                }
                plibptr = plibptr->next;
        }
        free_uentrylist(pliblist);

        /* get kernel software providers */
        (void) printf(gettext("\nKernel software providers:\n"));

        /*
         * TRANSLATION_NOTE
         * Strictly for appearance's sake, this line should be as long as
         * the length of the translated text above.
         */
        (void) printf(gettext("==========================\n"));
        if (getzoneid() == GLOBAL_ZONEID) {
                /* get kernel software providers from kernel ioctl */
                crypto_get_soft_list_t          *psoftlist_kernel = NULL;
                uint_t                          sl_soft_count;
                char                            *psoftname;
                int                             i;
                entrylist_t                     *pdevlist_conf = NULL;
                entrylist_t                     *psoftlist_conf = NULL;
                /* use kcf.conf for kernel software providers in global zone */
                pdevlist_conf = NULL;
                psoftlist_conf = NULL;

                if (get_soft_list(&psoftlist_kernel) == FAILURE) {
                        cryptoerror(LOG_ERR, gettext("Failed to retrieve the "
                            "software provider list from kernel."));
                        return (FAILURE);
                if (get_kcfconf_info(&pdevlist_conf, &psoftlist_conf) !=
                    SUCCESS) {
                        cryptoerror(LOG_STDERR, gettext("failed to retrieve "
                            "the list of kernel software providers.\n"));
                        rc = FAILURE;
                }
                sl_soft_count = psoftlist_kernel->sl_soft_count;

                if (get_kcfconf_info(&pdevlist_conf, &psoftlist_conf)
                    == FAILURE) {
                        cryptoerror(LOG_ERR,
                            "failed to retrieve the providers’ "
                            "information from file kcf.conf - %s.",
                            _PATH_KCF_CONF);
                        free(psoftlist_kernel);
                        return (FAILURE);
                }

                for (i = 0, psoftname = psoftlist_kernel->sl_soft_names;
                    i < sl_soft_count;
                    ++i, psoftname += strlen(psoftname) + 1) {
                        pent = getent_kef(psoftname, pdevlist_conf,
                            psoftlist_conf);
                        if ((pent == NULL) || (pent->load)) {
                                rv = list_mechlist_for_soft(psoftname,
                                    NULL, NULL);
                ptr = psoftlist_conf;
                while (ptr != NULL) {
                        if (check_active_for_soft(ptr->pent->name, &is_active)
                            == SUCCESS) {
                                if (is_active) {
                                        rv = list_mechlist_for_soft(
                                            ptr->pent->name);
                                if (rv == FAILURE) {
                                        rc = FAILURE;
                                }
                        } else {
                                (void) printf(gettext("%s: (inactive)\n"),
                                    psoftname);
                                        (void) printf(gettext(
                                            "%s: (inactive)\n"),
                                            ptr->pent->name);
                        }
                        } else {
                                /* should not happen */
                                (void) printf(gettext(
                                    "%s: failed to get the mechanism list.\n"),
                                    ptr->pent->name);
                                rc = FAILURE;
                }
                        ptr = ptr->next;
                }

                free(psoftlist_kernel);
                free_entrylist(pdevlist_conf);
                free_entrylist(psoftlist_conf);

        } else {
                /* kcf.conf not there in non-global zone, use /dev/cryptoadm */
                entrylist_t     *pdevlist_zone = NULL;
                entrylist_t     *psoftlist_zone = NULL;
                entrylist_t     *ptr;
                pdevlist_zone = NULL;
                psoftlist_zone = NULL;
```

```
                if (get_admindev_info(&pdevlist_zone, &psoftlist_zone) !=
                    SUCCESS) {
                        cryptoerror(LOG_STDERR, gettext("failed to retrieve "
                            "the list of kernel software providers.\n"));
                        rc = FAILURE;
                }

                for (ptr = psoftlist_zone; ptr != NULL; ptr = ptr->next) {
                        rv = list_mechlist_for_soft(ptr->pent->name,
                            pdevlist_zone, psoftlist_zone);
                ptr = psoftlist_zone;
                while (ptr != NULL) {
                        rv = list_mechlist_for_soft(ptr->pent->name);
                        if (rv == FAILURE) {
                                (void) printf(gettext(
                                    "%s: failed to get the mechanism list.\n"),
                                    ptr->pent->name);
                                rc = FAILURE;
                        }
                        ptr = ptr->next;
                }

                free_entrylist(pdevlist_zone);
                free_entrylist(psoftlist_zone);
        }

        /* Get kernel hardware providers and their mechanism lists */
        (void) printf(gettext("\nKernel hardware providers:\n"));
        /*
         * TRANSLATION_NOTE
         * Strictly for appearance’s sake, this line should be as long as
         * the length of the translated text above.
         */
        (void) printf(gettext("==========================\n"));
        if (get_dev_list(&pdevlist_kernel) != SUCCESS) {
                cryptoerror(LOG_STDERR, gettext("failed to retrieve "
                    "the list of hardware providers.\n"));
                return (FAILURE);
        }

        for (i = 0; i < pdevlist_kernel->dl_dev_count; i++) {
                (void) strlcpy(devname,
                    pdevlist_kernel->dl_devs[i].le_dev_name, MAXNAMELEN);
                inst_num = pdevlist_kernel->dl_devs[i].le_dev_instance;
                count = pdevlist_kernel->dl_devs[i].le_mechanism_count;
                (void) snprintf(provname, sizeof (provname), "%s/%d", devname,
                    inst_num);
                if (get_dev_info(devname, inst_num, count, &pmechlist) ==
                    SUCCESS) {
                        (void) filter_mechlist(&pmechlist, RANDOM);
                        print_mechlist(provname, pmechlist);
                        free_mechlist(pmechlist);
                } else {
                        (void) printf(gettext("%s: failed to get the mechanism"
                            " list.\n"), provname);
                        rc = FAILURE;
                }
        }
        free(pdevlist_kernel);
        return (rc);
}


/*
 * List all the providers. And for each provider, list the policy information.
 * Called for "cryptoadm list -p".
 */
static int
list_policy_for_all(void)
{
        crypto_get_dev_list_t   *pdevlist_kernel = NULL;
        uentrylist_t            *pliblist = NULL;
        entrylist_t             *pdevlist_conf = NULL;
        entrylist_t             *psoftlist_conf = NULL;
        entrylist_t             *ptr = NULL;
        entrylist_t             *phead = NULL;
        boolean_t               found = B_FALSE;
        crypto_get_dev_list_t   *pdevlist_kernel;
        uentrylist_t    *pliblist;
        uentrylist_t    *plibptr;
        entrylist_t     *pdevlist_conf;
        entrylist_t     *psoftlist_conf;
        entrylist_t     *ptr;
        entrylist_t     *phead;
        boolean_t       found;
        char                    provname[MAXNAMELEN];
        int                     i;
        int                     rc = SUCCESS;

        /* Get user-level providers */
        (void) printf(gettext("\nUser-level providers:\n"));
        /*
         * TRANSLATION_NOTE
         * Strictly for appearance's sake, this line should be as long as
         * the length of the translated text above.
         */
        (void) printf(gettext("=====================\n"));
        if (get_pkcs11conf_info(&pliblist) == FAILURE) {
                cryptoerror(LOG_STDERR, gettext("failed to retrieve "
                    "the list of user-level providers.\n"));
                rc = FAILURE;
        } else {
                uentrylist_t    *plibptr = pliblist;

                plibptr = pliblist;
                while (plibptr != NULL) {
                        /* skip metaslot entry */
                        if (strcmp(plibptr->puent->name,
                            METASLOT_KEYWORD) != 0) {
                                if (print_uef_policy(plibptr->puent)
                                    == FAILURE) {
                                        rc = FAILURE;
                                }
                        }
                        plibptr = plibptr->next;
                }
                free_uentrylist(pliblist);
        }

        /* kernel software providers */
        (void) printf(gettext("\nKernel software providers:\n"));
        /*
         * TRANSLATION_NOTE
         * Strictly for appearance's sake, this line should be as long as
         * the length of the translated text above.
         */
        (void) printf(gettext("==========================\n"));

        /* Get all entries from the kernel */
        /* Get all entries from the kcf.conf file */
        pdevlist_conf = NULL;
```

```
        if (getzoneid() == GLOBAL_ZONEID) {
                /* get kernel software providers from kernel ioctl */
                crypto_get_soft_list_t          *psoftlist_kernel = NULL;
                uint_t                          sl_soft_count;
                char                            *psoftname;
                int                             i;
                /* use kcf.conf for kernel software providers in global zone */
                psoftlist_conf = NULL;

                if (get_soft_list(&psoftlist_kernel) == FAILURE) {
                        cryptoerror(LOG_ERR, gettext("Failed to retrieve the "
                            "software provider list from kernel."));
                        rc = FAILURE;
                } else {
                        sl_soft_count = psoftlist_kernel->sl_soft_count;
                if (get_kcfconf_info(&pdevlist_conf, &psoftlist_conf) ==
                    FAILURE) {
                        cryptoerror(LOG_STDERR, gettext(
                            "failed to retrieve the list of kernel "
                            "providers.\n"));
                        return (FAILURE);
                }

                        for (i = 0, psoftname = psoftlist_kernel->sl_soft_names;
                            i < sl_soft_count;
                            ++i, psoftname += strlen(psoftname) + 1) {
                                (void) list_policy_for_soft(psoftname,
                                    pdevlist_conf, psoftlist_conf);
                ptr = psoftlist_conf;
                while (ptr != NULL) {
                        (void) list_policy_for_soft(ptr->pent->name);
                        ptr = ptr->next;
                        }
                        free(psoftlist_kernel);
                }

                free_entrylist(psoftlist_conf);
        } else {
                /* kcf.conf not there in non-global zone, no policy info */

                /*
                 * TRANSLATION_NOTE
                 * "global" is keyword and not to be translated.
                 */
                cryptoerror(LOG_STDERR, gettext(
                    "policy information for kernel software providers is "
                    "available in the %s zone only"), "global");
        }

        /* Kernel hardware providers */
        (void) printf(gettext("\nKernel hardware providers:\n"));
        /*
         * TRANSLATION_NOTE
         * Strictly for appearance's sake, this line should be as long as
         * the length of the translated text above.
         */
        (void) printf(gettext("==========================\n"));

        if (getzoneid() != GLOBAL_ZONEID) {
                /*
                 * TRANSLATION_NOTE
                 * "global" is keyword and not to be translated.
                 */
                cryptoerror(LOG_STDERR, gettext(
                    "policy information for kernel hardware providers is "
                    "available in the %s zone only"), "global");
                return (FAILURE);
        }

        /* Get the hardware provider list from kernel */
        if (get_dev_list(&pdevlist_kernel) != SUCCESS) {
                cryptoerror(LOG_STDERR, gettext(
                    "failed to retrieve the list of hardware providers.\n"));
                free_entrylist(pdevlist_conf);
                return (FAILURE);
        }

        if (get_kcfconf_info(&pdevlist_conf, &psoftlist_conf) == FAILURE) {
                cryptoerror(LOG_ERR, "failed to retrieve the providers' "
                    "information from file kcf.conf - %s.",
                    _PATH_KCF_CONF);
                return (FAILURE);
        }


        /*
         * For each hardware provider from kernel, check if it has an entry
         * in the config file.  If it has an entry, print out the policy from
         * config file and remove the entry from the hardware provider list
         * of the config file.  If it does not have an entry in the config
         * file, no mechanisms of it have been disabled. But, we still call
         * list_policy_for_hard() to account for the "random" feature.
         */
        for (i = 0; i < pdevlist_kernel->dl_dev_count; i++) {
                (void) snprintf(provname, sizeof (provname), "%s/%d",
                    pdevlist_kernel->dl_devs[i].le_dev_name,
                    pdevlist_kernel->dl_devs[i].le_dev_instance);

                found = B_FALSE;
                phead = ptr = pdevlist_conf;
                while (!found && ptr) {
                        if (strcmp(ptr->pent->name, provname) == 0) {
                                found = B_TRUE;
                        } else {
                                phead = ptr;
                                ptr = ptr->next;
                        }
                }

                if (found) {
                        (void) list_policy_for_hard(ptr->pent->name,
                            pdevlist_conf, psoftlist_conf, pdevlist_kernel);
                        (void) list_policy_for_hard(ptr->pent->name);
                        if (phead == ptr) {
                                pdevlist_conf = pdevlist_conf->next;
                        } else {
                                phead->next = ptr->next;
                        }
                        free_entry(ptr->pent);
                        free(ptr);
                } else {
                        (void) list_policy_for_hard(provname, pdevlist_conf,
                            psoftlist_conf, pdevlist_kernel);
                        (void) list_policy_for_hard(provname);
                }
        }

        /*
         * If there are still entries left in the pdevlist_conf list from
         * the config file, these providers must have been detached.
         * Should print out their policy information also.
         */
```

```
        for (ptr = pdevlist_conf; ptr != NULL; ptr = ptr->next) {
                print_kef_policy(ptr->pent->name, ptr->pent, B_FALSE, B_TRUE);
        ptr = pdevlist_conf;
        while (ptr != NULL) {
                print_kef_policy(ptr->pent, B_FALSE, B_TRUE);
                ptr = ptr->next;
        }

        free_entrylist(pdevlist_conf);
        free_entrylist(psoftlist_conf);
        free(pdevlist_kernel);

        return (rc);
}
______unchanged_portion_omitted_
```

```
**********************************************************
    6141 Tue Oct 28 16:45:36 2008
new/usr/src/cmd/cmd-crypto/cryptoadm/cryptoadm.h
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _CRYPTOADM_H
#define _CRYPTOADM_H

#include <sys/types.h>
#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/crypto/ioctladmin.h>
#include <cryptoutil.h>
#include <security/cryptoki.h>

#ifdef __cplusplus
extern "C" {
#endif

#define _PATH_KCF_CONF          "/etc/crypto/kcf.conf"
#define _PATH_KCFD              "/usr/lib/crypto/kcfd"
#define TMPFILE_TEMPLATE        "/etc/crypto/admXXXXXX"

#define ERROR_USAGE     2

/*
 * Common keywords and delimiters for pkcs11.conf and kcf.conf files are
 * defined in usr/lib/libcryptoutil/common/cryptoutil.h.  The following is
 * the extra keywords and delimiters used in kcf.conf file.
 */
#define SEP_SLASH               '/'
#define EF_SUPPORTED            "supportedlist="
#define EF_UNLOAD               "unload"
#define HW_DRIVER_STRING        "driver_names"
#define RANDOM                  "random"
#define UEF_FRAME_LIB           "/usr/lib/libpkcs11.so"

#define ADD_MODE        1
#define DELETE_MODE     2
#define MODIFY_MODE     3
```

```
typedef char prov_name_t[MAXNAMELEN];
typedef char mech_name_t[CRYPTO_MAX_MECH_NAME];

typedef struct mechlist {
        mech_name_t     name;
        struct mechlist *next;
} mechlist_t;


typedef struct entry {
        prov_name_t     name;
        mechlist_t      *suplist; /* supported list */
        uint_t          sup_count;
        mechlist_t      *dislist; /* disabled list */
        uint_t          dis_count;
        boolean_t       load; /* B_FALSE after cryptoadm unload */
} entry_t;
______unchanged_portion_omitted_

extern int errno;

/* adm_util */
extern boolean_t is_in_list(char *, mechlist_t *);
extern mechlist_t *create_mech(char *);
extern void free_mechlist(mechlist_t *);

/* adm_kef_util */
extern boolean_t is_device(char *);
extern char *ent2str(entry_t *);
extern entry_t *getent_kef(char *provname,
                entrylist_t *pdevlist, entrylist_t *psoftlist);
extern int check_kernel_for_soft(char *provname,
                crypto_get_soft_list_t *psoftlist, boolean_t *in_kernel);
extern int check_kernel_for_hard(char *provname,
                crypto_get_dev_list_t *pdevlist, boolean_t *in_kernel);
extern entry_t *getent_kef(char *);
extern int check_active_for_soft(char *, boolean_t *);
extern int check_active_for_hard(char *, boolean_t *);
extern int disable_mechs(entry_t **, mechlist_t *, boolean_t, mechlist_t *);
extern int enable_mechs(entry_t **, boolean_t, mechlist_t *);
extern int get_kcfconf_info(entrylist_t **, entrylist_t **);
extern int get_admindev_info(entrylist_t **, entrylist_t **);
extern int get_mech_count(mechlist_t *);
extern entry_t *create_entry(char *provname);
extern int insert_kcfconf(entry_t *);
extern int split_hw_provname(char *, char *, int *);
extern int update_kcfconf(entry_t *, int);
extern void free_entry(entry_t *);
extern void free_entrylist(entrylist_t *);
extern void print_mechlist(char *, mechlist_t *);
extern void print_kef_policy(char *provname, entry_t *pent,
                boolean_t has_random, boolean_t has_mechs);
extern void print_kef_policy(entry_t *, boolean_t, boolean_t);
extern boolean_t filter_mechlist(mechlist_t **, const char *);
extern uentry_t *getent_uef(char *);


/* adm_uef */
extern int list_mechlist_for_lib(char *, mechlist_t *, flag_val_t *,
                boolean_t, boolean_t, boolean_t);
extern int list_policy_for_lib(char *);
extern int disable_uef_lib(char *, boolean_t, boolean_t, mechlist_t *);
extern int enable_uef_lib(char *, boolean_t, boolean_t, mechlist_t *);
extern int install_uef_lib(char *);
extern int uninstall_uef_lib(char *);
extern int print_uef_policy(uentry_t *);
extern void display_token_flags(CK_FLAGS flags);
extern int convert_mechlist(CK_MECHANISM_TYPE **, CK_ULONG *, mechlist_t *);
extern void display_verbose_mech_header();
extern void display_mech_info(CK_MECHANISM_INFO *);
extern int display_policy(uentry_t *);
extern int update_pkcs11conf(uentry_t *);
extern int update_policylist(uentry_t *, mechlist_t *, int);

/* adm_kef */
extern int list_mechlist_for_soft(char *provname,
                entrylist_t *phardlist, entrylist_t *psoftlist);
extern int list_mechlist_for_soft(char *);
extern int list_mechlist_for_hard(char *);
extern int list_policy_for_soft(char *provname,
                entrylist_t *phardlist, entrylist_t *psoftlist);
extern int list_policy_for_hard(char *provname,
                entrylist_t *phardlist, entrylist_t *psoftlist,
                crypto_get_dev_list_t *pdevlist);
extern int list_policy_for_soft(char *);
extern int list_policy_for_hard(char *);
extern int disable_kef_software(char *, boolean_t, boolean_t, mechlist_t *);
extern int disable_kef_hardware(char *, boolean_t, boolean_t, mechlist_t *);
extern int enable_kef(char *, boolean_t, boolean_t, mechlist_t *);
extern int install_kef(char *, mechlist_t *);
extern int uninstall_kef(char *);
extern int unload_kef_soft(char *provname);
extern int unload_kef_soft(char *, boolean_t);
extern int refresh(void);
extern int start_daemon(void);
extern int stop_daemon(void);

/* adm_ioctl */
extern crypto_load_soft_config_t *setup_soft_conf(entry_t *);
extern crypto_load_soft_disabled_t *setup_soft_dis(entry_t *);
extern crypto_load_dev_disabled_t *setup_dev_dis(entry_t *);
extern crypto_unload_soft_module_t *setup_unload_soft(entry_t *);
extern int get_dev_info(char *, int, int, mechlist_t **);
extern int get_dev_list(crypto_get_dev_list_t **);
extern int get_soft_info(char *provname, mechlist_t **ppmechlist,
                entrylist_t *phardlist, entrylist_t *psoftlist);
extern int get_soft_info(char *, mechlist_t **);
extern int get_soft_list(crypto_get_soft_list_t **);

/* adm_metaslot */
extern int list_metaslot_info(boolean_t, boolean_t, mechlist_t *);
extern int list_metaslot_policy();
extern int disable_metaslot(mechlist_t *, boolean_t, boolean_t);
extern int enable_metaslot(char *, char *, boolean_t, mechlist_t *, boolean_t,
    boolean_t);

#ifdef __cplusplus
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
    1332 Tue Oct 28 16:45:40 2008
new/usr/src/cmd/cmd-crypto/etc/kcf.conf
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
#
# 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 2008 Sun Microsystems, Inc.  All rights reserved.
#
# Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# ident "%Z%%M% %I%     %E% SMI"
#
# /etc/crypto/kcf.conf
#
# Do NOT edit this file by hand.   An administrator should use cryptoadm(1m)
# to administer the cryptographic framework.
# to administer the cryptographic framework.  A developer for a kernel software
# provider package or a cryptographic provider device driver(s) package should
# provide an input file and use the {i,r}.kcfconf class action scripts to
# update this file during the installation and removal of the package.
#
# This document does not constitute an API.  The /etc/crypto/kcf.conf file may
# not exist or may have a different content or interpretation in a future
# release.  The existence of this notice does not imply that any other
# documentation that lacks this notice constitutes an API.
#
#
#
# Start SUNWcsr
des:supportedlist=CKM_DES_CBC,CKM_DES_ECB,CKM_DES3_CBC,CKM_DES3_ECB
aes:supportedlist=CKM_AES_ECB,CKM_AES_CBC,CKM_AES_CTR
arcfour:supportedlist=CKM_RC4
blowfish:supportedlist=CKM_BLOWFISH_ECB,CKM_BLOWFISH_CBC
ecc:supportedlist=CKM_EC_KEY_PAIR_GEN,CKM_ECDH1_DERIVE,CKM_ECDSA,CKM_ECDSA_SHA1
sha1:supportedlist=CKM_SHA_1,CKM_SHA_1_HMAC_GENERAL,CKM_SHA_1_HMAC
sha2:supportedlist=CKM_SHA256,CKM_SHA256_HMAC,CKM_SHA256_HMAC_GENERAL,CKM_SHA384
md4:supportedlist=CKM_MD4
md5:supportedlist=CKM_MD5,CKM_MD5_HMAC_GENERAL,CKM_MD5_HMAC
rsa:supportedlist=CKM_RSA_PKCS,CKM_RSA_X_509,CKM_MD5_RSA_PKCS,CKM_SHA1_RSA_PKCS,
swrand:supportedlist=random
# End SUNWcsr
```

```
**********************************************************
    2168 Tue Oct 28 16:45:46 2008
new/usr/src/pkgdefs/SUNWcryptoint/prototype_com
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
```

```
#
# 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 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# This required package information file contains a list of package contents.
# The 'pkgmk' command uses this file to identify the contents of a package
# and their location on the development machine when building the package.
# Can be created via a text editor or through use of the 'pkgproto' command.

#!search <pathname pathname ...>        # where to find pkg objects
#!include <filename>                    # include another 'prototype' file
#!default <mode> <owner> <group>        # default used if not specified on entry
#!<param>=<value>                       # puts parameter in pkg environment

# packaging files
i pkginfo
i copyright
i depend
i postinstall
i preremove
#
# source locations relative to the prototype file
#
# SUNWcryptoint
#
# CRYPT DELETE START
d none etc 755 root sys
d none etc/certs 755 root sys
f none etc/certs/SUNWosnetSolaris 644 root sys
f none etc/certs/SUNWosnetSE 644 root sys
d none etc/crypto 755 root sys
d none etc/crypto/certs 755 root sys
f none etc/crypto/certs/SUNWosnet 644 root sys
f none etc/crypto/certs/SUNWosnetLimited 644 root sys
f none etc/crypto/certs/SUNWosnetCF 644 root sys
f none etc/crypto/certs/SUNWosnetCFLimited 644 root sys
# CRYPT DELETE END
d none kernel 755 root sys
d none kernel/crypto 755 root sys
d none kernel/drv 755 root sys
f none kernel/drv/dprov.conf 644 root sys
```

****
1763 Tue Oct 28 16:45:53 2008
new/usr/src/pkgdefs/SUNWdcar/postinstall
6414175 kcf.conf’s supportedlist not providing much usefulness
****

```
#! /bin/sh
#
# 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 2008 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#pragma ident   "%Z%%M% %I%     %E% SMI"
#

#
# install the Sun Crypto Accelerator 1000 device drivers.
#

PATH="/usr/bin:/usr/sbin:${PATH}"
export PATH

# Add hardware provider section for the dca driver
# to /etc/crypto/kcf.conf

pkg_start="# Start $PKGINST"
pkg_end="# End $PKGINST"
kcfconf=${BASEDIR}/etc/crypto/kcf.conf
tmpfile=/tmp/$$kcfconf
error=no

#
# If /etc/crypto/kcf.conf doesn’t exist, bail immediately
#
if [ ! -f "$kcfconf" ]
then
        echo "$0: ERROR - $kcfconf doesn’t exist"
        exit 2
fi

#
# If the package has been already installed, remove old entries
#
start=0
end=0
egrep -s "$pkg_start" $kcfconf && start=1
```

```
egrep -s "$pkg_end" $kcfconf && end=1

if [ $start -ne $end ] ; then
        echo "$0: missing Start or End delimiters for $PKGINST in $kcfconf."
        echo "$0: $kcfconf may be corrupted and was not updated."
        error=yes
        exit 2
fi

# to preserve the gid
cp -p $kcfconf $tmpfile || error=yes
if [ $start -eq 1 ]
then
        sed -e "/$pkg_start/,/$pkg_end/d" $kcfconf > $tmpfile || error=yes
fi

#
# Append the delimiters for this package
#
echo "$pkg_start driver_names=dca" >> $tmpfile || error=yes
echo "$pkg_end" >> $tmpfile || error=yes

#
# Install the updated config file and clean up the tmp file
#
if [ "$error" = no ]
then
        mv $tmpfile $kcfconf || error=yes
fi
rm -f $tmpfile

#
# All done, if any of the steps above fail, report the error
#
if [ "$error" = yes ]
then
        echo "$0: ERROR - failed to update $kcfconf."
        exit 2
fi

NAMEMAJOR="${BASEDIR}/etc/name_to_major"

#
# Is the hardware there?
#
check_hardware()
{
        for i in "pci14e4,5820" "pci14e4,5821" "pci14e4,5822" "pci14e4,5825" \
                "pci108e,5454" "pci108e,5455" "pci108e,5456" "pci108e,5457"
        do
                prtconf -pv | egrep -s "$i"
                if [ $? -eq 0 ]
                then
                        return 1
                fi
        done
        return 0
}
```

______***unchanged_portion_omitted_***

```
************************************************************
    1426 Tue Oct 28 16:45:58 2008
new/usr/src/pkgdefs/SUNWdcar/preremove
6414175 kcf.conf's supportedlist not providing much usefulness
************************************************************
#! /bin/sh
#
# 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 2008 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

#pragma ident   "%Z%%M% %I%     %E% SMI"
#
#
# Preremove script for SUNWdcar.
#
# This script removes the driver with rem_drv(1M) if necessary.
# This script removes the hardware provider section for the dca
# driver from /etc/crypto/kcf.conf
#
DRV=dca

NAMEMAJOR="${BASEDIR}/etc/name_to_major"

#
# Determine if we are on an alternate BASEDIR
#
if [ "${BASEDIR:=/}" = "/" ]
then
        REM_DRV="/usr/sbin/rem_drv"
else
        REM_DRV="/usr/sbin/rem_drv -b ${BASEDIR}"
fi

#
# Remove the driver, but only if this has not already been done.
#
grep -w "${DRV}" ${NAMEMAJOR} > /dev/null 2>&1
if [ $? -eq 0 ]; then
    ${REM_DRV} ${DRV} || exit 1
fi

pkg_start="# Start $PKGINST"
pkg_end="# End $PKGINST"
```

```
kcfconf=${BASEDIR}/etc/crypto/kcf.conf
tmpfile=/tmp/$$kcfconf
error=no

#
# If /etc/crypto/kcf.conf doesn't exist, bail immediately
#
if [ ! -f "$kcfconf" ]
then
        echo "$0: ERROR - $kcfconf doesn't exist"
        exit 2
fi

#
# Strip all entries belonging to this package
#
start=0
end=0
egrep -s "$pkg_start" $kcfconf && start=1
egrep -s "$pkg_end" $kcfconf && end=1

if [ $start -ne $end ] ; then
        echo "$0: missing Start or End delimiters for $PKGINST in $kcfconf."
        echo "$0: $kcfconf may be corrupted and was not updated."
        error=yes
        exit 2
fi

if [ $start -eq 1 ]
then
        # To preserve the gid
        cp -p $kcfconf $tmpfile
        sed -e "/$pkg_start/,/$pkg_end/d" $kcfconf > $tmpfile || error=yes
        if [ "$error" = no ]
        then
                mv $tmpfile $kcfconf || error=yes
        fi
        rm -f $tmpfile
else
        echo "$0: WARNING - no entries to be removed from $kcfconf"
        exit 0
fi

if [ "$error" = yes ]
then
        echo "$0: ERROR - failed to update $kcfconf."
        exit 2
fi
exit 0
```

```
**********************************************************
    1315 Tue Oct 28 16:46:02 2008
new/usr/src/pkgdefs/SUNWn2cp.v/postinstall
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
#! /bin/sh
#
# 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 2008 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# ident "%Z%%M% %I%     %E% SMI"
#

#
# install the UltraSPARC-T2 Crypto Provider device driver
#

PATH="/usr/bin:/usr/sbin:${PATH}"
export PATH

# Add hardware provider section for the n2cp driver
# to /etc/crypto/kcf.conf

pkg_start="# Start $PKGINST"
pkg_end="# End $PKGINST"
kcfconf=${BASEDIR}/etc/crypto/kcf.conf
tmpfile=/tmp/$$kcfconf
error=no

#
# If /etc/crypto/kcf.conf doesn't exist, bail immediately
#
if [ ! -f "$kcfconf" ]
then
        echo "$0: ERROR - $kcfconf doesn't exist"
        exit 2
fi

#
# If the package has been already installed, remove old entries
#
start=0
end=0
egrep -s "$pkg_start" $kcfconf && start=1
```

```
egrep -s "$pkg_end" $kcfconf && end=1

if [ $start -ne $end ] ; then
        echo "$0: missing Start or End delimiters for $PKGINST in $kcfconf."
        echo "$0: $kcfconf may be corrupted and was not updated."
        error=yes
        exit 2
fi

if [ $start -eq 1 ]
then
        cp -p $kcfconf $tmpfile || error=yes
        sed -e "/$pkg_start/,/$pkg_end/d" $kcfconf > $tmpfile || error=yes
else
        cp -p $kcfconf $tmpfile || error=yes
fi

#
# Append the delimiters for this package
#
echo "$pkg_start driver_names=n2cp" >> $tmpfile || error=yes
echo "$pkg_end" >> $tmpfile || error=yes

#
# Install the updated config file and clean up the tmp file
#
if [ "$error" = no ]
then
        mv $tmpfile $kcfconf || error=yes
fi
rm -f $tmpfile

#
# All done, if any of the steps above fail, report the error
#
if [ "$error" = yes ]
then
        echo "$0: ERROR - failed to update $kcfconf."
        exit 2
fi
NAMEMAJOR="${BASEDIR}/etc/name_to_major"

if [ "${BASEDIR:=/}" = "/" ]
then
        ADD_DRV="/usr/sbin/add_drv"
else
        ADD_DRV="/usr/sbin/add_drv -b ${BASEDIR}"
fi

grep -w n2cp ${NAMEMAJOR} > /dev/null 2>&1
if [ $? -ne 0 ]
then
    $ADD_DRV -i '"SUNW,n2-cwq" "SUNW,vf-cwq"' n2cp || exit 1
fi

exit 0
```

```
**********************************************************
    1425 Tue Oct 28 16:46:07 2008
new/usr/src/pkgdefs/SUNWn2cp.v/preremove
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
#! /bin/sh
#
# 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 2008 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

# ident "%Z%%M% %I%     %E% SMI"
#
#
# Preremove script for SUNWn2cp.v.
#
# This script removes driver with rem_drv(1M) if necessary.
# This script removes the hardware provider section for the n2cp
# driver from /etc/crypto/kcf.conf
#
DRV=n2cp

NAMEMAJOR="${BASEDIR}/etc/name_to_major"

#
# Determine if we are on an alternate BASEDIR
#
if [ "${BASEDIR:=/}" = "/" ]
then
        REM_DRV="/usr/sbin/rem_drv"
else
        REM_DRV="/usr/sbin/rem_drv -b ${BASEDIR}"
fi

#
# Remove the driver, but only if this has not already been done.
#
grep -w "${DRV}" ${NAMEMAJOR} > /dev/null 2>&1
if [ $? -eq 0 ]; then
    ${REM_DRV} ${DRV} || exit 1
fi

pkg_start="# Start $PKGINST"
pkg_end="# End $PKGINST"
```

```
kcfconf=${BASEDIR}/etc/crypto/kcf.conf
tmpfile=/tmp/$$kcfconf
error=no

#
# If /etc/crypto/kcf.conf doesn't exist, bail immediately
#
if [ ! -f "$kcfconf" ]
then
        echo "$0: ERROR - $kcfconf doesn't exist"
        exit 2
fi

#
# Strip all entries belonging to this package
#
start=0
end=0
egrep -s "$pkg_start" $kcfconf && start=1
egrep -s "$pkg_end" $kcfconf && end=1

if [ $start -ne $end ] ; then
        echo "$0: missing Start or End delimiters for $PKGINST in $kcfconf."
        echo "$0: $kcfconf may be corrupted and was not updated."
        error=yes
        exit 2
fi

if [ $start -eq 1 ]
then
        cp -p $kcfconf $tmpfile || error=yes
        sed -e "/$pkg_start/,/$pkg_end/d" $kcfconf > $tmpfile || error=yes
        if [ "$error" = no ]
        then
                mv $tmpfile $kcfconf || error=yes
        fi
        rm -f $tmpfile
else
        exit 0
fi

if [ "$error" = yes ]
then
        echo "$0: ERROR - failed to update $kcfconf."
        exit 2
fi
exit 0
```

```
************************************************************
   31872 Tue Oct 28 16:46:18 2008
new/usr/src/uts/common/crypto/core/kcf_cryptoadm.c
6414175 kcf.conf's supportedlist not providing much usefulness
************************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * Core KCF (Kernel Cryptographic Framework). This file implements
 * the cryptoadm entry points.
 */

#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/rwlock.h>
#include <sys/kmem.h>
#include <sys/modctl.h>
#include <sys/sunddi.h>
#include <sys/door.h>
#include <sys/crypto/common.h>
#include <sys/crypto/api.h>
#include <sys/crypto/spi.h>
#include <sys/crypto/impl.h>
#include <sys/crypto/sched_impl.h>

/* protects the the soft_config_list. */
kmutex_t soft_config_mutex;

/*
 * This linked list contains software configuration entries.
 * The initial list is just software providers loaded by kcf_soft_config_init().
 * Additional entries may appear for both hardware and software providers
 * from kcf.conf.  These come from "cryptoadm start", which reads file kcf.conf
 * and updates this table using the CRYPTO_LOAD_SOFT_CONFIG ioctl.
 * Further cryptoadm commands modify this file and update this table with ioctl.
 * This list is protected by the soft_config_mutex.
 * This linked list contains software configuration entries that
 * are loaded into the kernel by the CRYPTO_LOAD_SOFT_CONFIG ioctl.
 * It is protected by the soft_config_mutex.
 */
```

```
kcf_soft_conf_entry_t *soft_config_list;

static int add_soft_config(char *, uint_t, crypto_mech_name_t *);
static int dup_mech_names(kcf_provider_desc_t *, crypto_mech_name_t **,
    uint_t *, int);
static void free_soft_config_entry(kcf_soft_conf_entry_t *);

#define KCF_MAX_CONFIG_ENTRIES 512 /* maximum entries in soft_config_list */

#if DEBUG
extern int kcf_frmwrk_debug;
static void kcf_soft_config_dump(char *message);
#endif /* DEBUG */

/*
 * Count and return the number of mechanisms in an array of crypto_mech_name_t
 * (excluding final NUL-character string element).
 */
static int
count_mechanisms(crypto_mech_name_t mechs[]) {
        int     count;
        for (count = 0; mechs[count][0] != '\0'; ++count);
        return (count);
}

/*
 * Initialize a mutex and populate soft_config_list with default entries
 * of kernel software providers.
 * Called from kcf module _init().
 */
void
kcf_soft_config_init(void)
{
        typedef struct {
                char                    *name;
                crypto_mech_name_t      *mechs;
        } initial_soft_config_entry_t;

        /*
         * This provides initial default values to soft_config_list.
         * It is equivalent to these lines in /etc/crypto/kcf.conf
         * (without line breaks and indenting):
         *
         * # /etc/crypto/kcf.conf
         * des:supportedlist=CKM_DES_CBC,CKM_DES_ECB,CKM_DES3_CBC,CKM_DES3_ECB
         * aes:supportedlist=CKM_AES_ECB,CKM_AES_CBC,CKM_AES_CTR,CKM_AES_CCM
         * arcfour:supportedlist=CKM_RC4
         * blowfish:supportedlist=CKM_BLOWFISH_ECB,CKM_BLOWFISH_CBC
         * ecc:supportedlist=CKM_EC_KEY_PAIR_GEN,CKM_ECDH1_DERIVE,CKM_ECDSA,\
         * CKM_ECDSA_SHA1
         * sha1:supportedlist=CKM_SHA_1,CKM_SHA_1_HMAC_GENERAL,CKM_SHA_1_HMAC
         * sha2:supportedlist=CKM_SHA256,CKM_SHA256_HMAC,
         * CKM_SHA256_HMAC_GENERAL,CKM_SHA384,CKM_SHA384_HMAC,\
         * CKM_SHA384_HMAC_GENERAL,CKM_SHA512,CKM_SHA512_HMAC,\
         * CKM_SHA512_HMAC_GENERAL
         * md4:supportedlist=CKM_MD4
         * md5:supportedlist=CKM_MD5,CKM_MD5_HMAC_GENERAL,CKM_MD5_HMAC
         * rsa:supportedlist=CKM_RSA_PKCS,CKM_RSA_X_509,CKM_MD5_RSA_PKCS,\
         * CKM_SHA1_RSA_PKCS,CKM_SHA256_RSA_PKCS,CKM_SHA384_RSA_PKCS,\
         * CKM_SHA512_RSA_PKCS
         * swrand:supportedlist=random
         *
         * WARNING: If you add a new kernel crypto provider or mechanism,
         * you must update these constants.
         *
         * 1. To add a new mechanism to a provider add the string to the
         * appropriate array below.
         *
         * 2. To add a new provider, create a new *_mechs array listing the
         * provider's mechanism(s).  For example:
         *      sha3_mechs[SHA3_MECH_COUNT] = {"CKM_SHA_3"};
         * Add the new *_mechs array to initial_soft_config_entry[].
         */
        static crypto_mech_name_t       des_mechs[] = {
            "CKM_DES_CBC", "CKM_DES_ECB", "CKM_DES3_CBC", "CKM_DES3_ECB", ""};
        static crypto_mech_name_t       aes_mechs[] = {
            "CKM_AES_ECB", "CKM_AES_CBC", "CKM_AES_CTR", "CKM_AES_CCM", ""};
        static crypto_mech_name_t       arcfour_mechs[] = {
            "CKM_RC4", ""};
        static crypto_mech_name_t       blowfish_mechs[] = {
            "CKM_BLOWFISH_ECB", "CKM_BLOWFISH_CBC", ""};
        static crypto_mech_name_t       ecc_mechs[] = {
            "CKM_EC_KEY_PAIR_GEN", "CKM_ECDH1_DERIVE", "CKM_ECDSA",
            "CKM_ECDSA_SHA1", ""};
        static crypto_mech_name_t       sha1_mechs[] = {
            "CKM_SHA_1", "CKM_SHA_1_HMAC_GENERAL", "CKM_SHA_1_HMAC", ""};
        static crypto_mech_name_t       sha2_mechs[] = {
            "CKM_SHA256", "CKM_SHA256_HMAC", "CKM_SHA256_HMAC_GENERAL",
            "CKM_SHA384", "CKM_SHA384_HMAC", "CKM_SHA384_HMAC_GENERAL",
            "CKM_SHA512", "CKM_SHA512_HMAC", "CKM_SHA512_HMAC_GENERAL", ""};
        static crypto_mech_name_t       md4_mechs[] = {
            "CKM_MD4", ""};
        static crypto_mech_name_t       md5_mechs[] = {
            "CKM_MD5", "CKM_MD5_HMAC_GENERAL", "CKM_MD5_HMAC", ""};
        static crypto_mech_name_t       rsa_mechs[] = {
            "CKM_RSA_PKCS", "CKM_RSA_X_509", "CKM_MD5_RSA_PKCS",
            "CKM_SHA1_RSA_PKCS", "CKM_SHA256_RSA_PKCS", "CKM_SHA384_RSA_PKCS",
            "CKM_SHA512_RSA_PKCS", ""};
        static crypto_mech_name_t       swrand_mechs[] = {
            "random", NULL};
        static initial_soft_config_entry_t
            initial_soft_config_entry[] = {
                "des", des_mechs,
                "aes", aes_mechs,
                "arcfour", arcfour_mechs,
                "blowfish", blowfish_mechs,
                "ecc", ecc_mechs,
                "sha1", sha1_mechs,
                "sha2", sha2_mechs,
                "md4", md4_mechs,
                "md5", md5_mechs,
                "rsa", rsa_mechs,
                "swrand", swrand_mechs
        };
        const int       initial_soft_config_entries =
            sizeof (initial_soft_config_entry)
            / sizeof (initial_soft_config_entry_t);
        int             i;

        mutex_init(&soft_config_mutex, NULL, MUTEX_DRIVER, NULL);

        /*
         * Initialize soft_config_list with default providers.
         * Populate the linked list backwards so the first entry appears first.
         */
        for (i = initial_soft_config_entries - 1; i >= 0; --i) {
                initial_soft_config_entry_t *p = &initial_soft_config_entry[i];
                crypto_mech_name_t      *mechsp;
                char                    *namep;
                uint_t                  namelen, alloc_size;
                int                     mech_count, r;
```

```
                /* allocate/initialize memory for name and mechanism list */
                namelen = strlen(p->name) + 1;
                namep = kmem_alloc(namelen, KM_SLEEP);
                (void) strlcpy(namep, p->name, namelen);
                mech_count = count_mechanisms(p->mechs);
                alloc_size = mech_count * CRYPTO_MAX_MECH_NAME;
                mechsp = kmem_alloc(alloc_size, KM_SLEEP);
                bcopy(p->mechs, mechsp, alloc_size);

                r = add_soft_config(namep, mech_count, mechsp);
                if (r != 0)
                        cmn_err(CE_WARN,
                            "add_soft_config(%s) failed; returned %d\n",
                            namep, r);
        }
#if DEBUG
        if (kcf_frmwrk_debug >= 1)
                kcf_soft_config_dump("kcf_soft_config_init");
#endif /* DEBUG */
}


#if DEBUG
/*
 * Dump soft_config_list, containing a list of kernel software providers
 * and (optionally) hardware providers, with updates from kcf.conf.
 * Dump mechanism lists too if kcf_frmwrk_debug is >= 2.
 */
static void
kcf_soft_config_dump(char *message)
{
        kcf_soft_conf_entry_t   *p;
        uint_t                  i;

        mutex_enter(&soft_config_mutex);
        printf("Soft provider config list soft_config_list: %s\n",
            message != NULL ? message : "");

        for (p = soft_config_list; p != NULL; p = p->ce_next) {
                printf("ce_name: %s, %d ce_mechs\n", p->ce_name, p->ce_count);
                if (kcf_frmwrk_debug >= 2) {
                        printf("\tce_mechs: ");
                        for (i = 0; i < p->ce_count; i++) {
                                printf("%s ", p->ce_mechs[i]);
                        }
                        printf("\n");
                }
        }
        printf("(end of soft_config_list)\n");

        mutex_exit(&soft_config_mutex);
}
#endif /* DEBUG */


/*
 * Utility routine to identify the providers to filter out and
 * present only one provider. This happens when a hardware provider
 * registers multiple units of the same device instance.
 *
 * Called from crypto_get_dev_list().
 */
static void
filter_providers(uint_t count, kcf_provider_desc_t **provider_array,
        char *skip_providers, int *mech_counts, int *new_count)
{
        int i, j;
        kcf_provider_desc_t *prov1, *prov2;
        int n = 0;

        for (i = 0; i < count; i++) {
                if (skip_providers[i] == 1)
                        continue;

                prov1 = provider_array[i];
                mech_counts[i] = prov1->pd_mech_list_count;
                for (j = i + 1; j < count; j++) {
                        prov2 = provider_array[j];
                        if (strncmp(prov1->pd_name, prov2->pd_name,
                            MAXNAMELEN) == 0 &&
                            prov1->pd_instance == prov2->pd_instance) {
                                skip_providers[j] = 1;
                                mech_counts[i] += prov2->pd_mech_list_count;
                        }
                }
                n++;
        }

        *new_count = n;
}


/*
 * Return a list of kernel hardware providers and a count of each
 * provider's supported mechanisms.
 * Called from the CRYPTO_GET_DEV_LIST ioctl.
 */
/* called from the CRYPTO_GET_DEV_LIST ioctl */
int
crypto_get_dev_list(uint_t *count, crypto_dev_list_entry_t **array)
{
        kcf_provider_desc_t **provider_array;
        kcf_provider_desc_t *pd;
        crypto_dev_list_entry_t *p;
        size_t skip_providers_size, mech_counts_size;
        char *skip_providers;
        uint_t provider_count;
        int rval, i, j, new_count, *mech_counts;

        /*
         * Take snapshot of provider table returning only hardware providers
         * that are in a usable state. Logical providers not included.
         */
        rval = kcf_get_hw_prov_tab(&provider_count, &provider_array, KM_SLEEP,
            NULL, 0, B_FALSE);
        if (rval != CRYPTO_SUCCESS)
                return (rval);

        if (provider_count == 0) {
                *array = NULL;
                *count = 0;
                return (CRYPTO_SUCCESS);
        }

        skip_providers_size = provider_count * sizeof (char);
        mech_counts_size = provider_count * sizeof (int);

        skip_providers = kmem_zalloc(skip_providers_size, KM_SLEEP);
        mech_counts = kmem_zalloc(mech_counts_size, KM_SLEEP);
        filter_providers(provider_count, provider_array, skip_providers,
            mech_counts, &new_count);
```

```
        p = kmem_alloc(new_count * sizeof (crypto_dev_list_entry_t), KM_SLEEP);
        for (i = 0, j = 0; i < provider_count; i++) {
                if (skip_providers[i] == 1) {
                        ASSERT(mech_counts[i] == 0);
                        continue;
                }
                pd = provider_array[i];
                p[j].le_mechanism_count = mech_counts[i];
                p[j].le_dev_instance = pd->pd_instance;
                (void) strncpy(p[j].le_dev_name, pd->pd_name, MAXNAMELEN);
                j++;
        }

        kcf_free_provider_tab(provider_count, provider_array);
        kmem_free(skip_providers, skip_providers_size);
        kmem_free(mech_counts, mech_counts_size);

        *array = p;
        *count = new_count;
        return (CRYPTO_SUCCESS);
}

/*
 * Return a buffer containing the null terminated names of software providers
 * Called from the CRYPTO_GET_SOFT_LIST ioctl, this routine returns
 * a buffer containing the null terminated names of software providers
 * loaded by CRYPTO_LOAD_SOFT_CONFIG.
 * Called from the CRYPTO_GET_SOFT_LIST ioctl.
 */
int
crypto_get_soft_list(uint_t *count, char **array, size_t *len)
{
        char *names = NULL, *namep, *end;
        kcf_soft_conf_entry_t *p;
        uint_t n = 0, cnt = 0, final_count = 0;
        size_t name_len, final_size = 0;

        /* first estimate */
        mutex_enter(&soft_config_mutex);
        for (p = soft_config_list; p != NULL; p = p->ce_next) {
                n += strlen(p->ce_name) + 1;
                cnt++;
        }
        mutex_exit(&soft_config_mutex);

        if (cnt == 0)
                goto out;

again:
        namep = names = kmem_alloc(n, KM_SLEEP);
        end = names + n;
        final_size = 0;
        final_count = 0;

        mutex_enter(&soft_config_mutex);
        for (p = soft_config_list; p != NULL; p = p->ce_next) {
                name_len = strlen(p->ce_name) + 1;
                /* check for enough space */
                if ((namep + name_len) > end) {
                        mutex_exit(&soft_config_mutex);
                        kmem_free(names, n);
                        n = n << 1;
                        goto again;
                }
                (void) strcpy(namep, p->ce_name);
                namep += name_len;
                final_size += name_len;
                final_count++;
        }
        mutex_exit(&soft_config_mutex);

        ASSERT(final_size <= n);

        /* check if buffer we allocated is too large */
        if (final_size < n) {
                char *final_buffer;

                final_buffer = kmem_alloc(final_size, KM_SLEEP);
                bcopy(names, final_buffer, final_size);
                kmem_free(names, n);
                names = final_buffer;
        }
out:
        *array = names;
        *count = final_count;
        *len = final_size;
        return (CRYPTO_SUCCESS);
}

/*
 * Check if a mechanism name is already in a mechanism name array
 * Called by crypto_get_dev_info().
 */
static boolean_t
duplicate(char *name, crypto_mech_name_t *array, int count)
{
        int i;

        for (i = 0; i < count; i++) {
                if (strncmp(name, &array[i][0],
                    sizeof (crypto_mech_name_t)) == 0)
                        return (B_TRUE);
        }
        return (B_FALSE);
}

/*
 * Return a list of kernel hardware providers for a given name and instance.
 * For each entry, also return a list of their supported mechanisms.
 * Called from the CRYPTO_GET_DEV_INFO ioctl.
 */
/* called from the CRYPTO_GET_DEV_INFO ioctl */
int
crypto_get_dev_info(char *name, uint_t instance, uint_t *count,
    crypto_mech_name_t **array)
{
        int rv;
        crypto_mech_name_t *mech_names, *resized_array;
        int i, j, k = 0, max_count;
        uint_t provider_count;
        kcf_provider_desc_t **provider_array;
        kcf_provider_desc_t *pd;

        /*
         * Get provider table entries matching name and instance
         * for hardware providers that are in a usable state.
         * Logical providers not included. NULL name matches
         * all hardware providers.
         */
        rv = kcf_get_hw_prov_tab(&provider_count, &provider_array, KM_SLEEP,
            name, instance, B_FALSE);
        if (rv != CRYPTO_SUCCESS)
```

```
                return (rv);

        if (provider_count == 0)
                return (CRYPTO_ARGUMENTS_BAD);

        /* Count all mechanisms supported by all providers */
        max_count = 0;
        for (i = 0; i < provider_count; i++)
                max_count += provider_array[i]->pd_mech_list_count;

        if (max_count == 0) {
                mech_names = NULL;
                goto out;
        }

        /* Allocate space and copy mech names */
        mech_names = kmem_alloc(max_count * sizeof (crypto_mech_name_t),
            KM_SLEEP);

        k = 0;
        for (i = 0; i < provider_count; i++) {
                pd = provider_array[i];
                for (j = 0; j < pd->pd_mech_list_count; j++) {
                        /* check for duplicate */
                        if (duplicate(&pd->pd_mechanisms[j].cm_mech_name[0],
                            mech_names, k))
                                continue;
                        bcopy(&pd->pd_mechanisms[j].cm_mech_name[0],
                            &mech_names[k][0], sizeof (crypto_mech_name_t));
                        k++;
                }
        }

        /* resize */
        if (k != max_count) {
                resized_array =
                    kmem_alloc(k * sizeof (crypto_mech_name_t), KM_SLEEP);
                bcopy(mech_names, resized_array,
                    k * sizeof (crypto_mech_name_t));
                kmem_free(mech_names,
                    max_count * sizeof (crypto_mech_name_t));
                mech_names = resized_array;
        }

out:
        kcf_free_provider_tab(provider_count, provider_array);
        *count = k;
        *array = mech_names;

        return (CRYPTO_SUCCESS);
}

/*
 * Given a kernel software provider name, return a list of mechanisms
 * it supports.
 * Called from the CRYPTO_GET_SOFT_INFO ioctl.
 */
/* called from the CRYPTO_GET_SOFT_INFO ioctl */
int
crypto_get_soft_info(caddr_t name, uint_t *count, crypto_mech_name_t **array)
{
        ddi_modhandle_t modh = NULL;
        kcf_provider_desc_t *provider;
        int rv;

        provider = kcf_prov_tab_lookup_by_name(name);
        if (provider == NULL) {
                if (in_soft_config_list(name)) {
                char *tmp;
                int name_len;

                /* strlen("crypto/") + NULL terminator == 8 */
                name_len = strlen(name);
                tmp = kmem_alloc(name_len + 8, KM_SLEEP);
                bcopy("crypto/", tmp, 7);
                bcopy(name, &tmp[7], name_len);
                tmp[name_len + 7] = '\0';

                modh = ddi_modopen(tmp, KRTLD_MODE_FIRST, NULL);
                kmem_free(tmp, name_len + 8);

                if (modh == NULL) {
                        return (CRYPTO_ARGUMENTS_BAD);
                }

                provider = kcf_prov_tab_lookup_by_name(name);
                if (provider == NULL) {
                        return (CRYPTO_ARGUMENTS_BAD);
                }
                } else {
                        return (CRYPTO_ARGUMENTS_BAD);
        }
        }

        rv = dup_mech_names(provider, array, count, KM_SLEEP);
        KCF_PROV_REFRELE(provider);
        if (modh != NULL)
                (void) ddi_modclose(modh);
        return (rv);
}


/*
 * Change the mechanism list for a provider.
 * If "direction" is CRYPTO_MECH_ADDED, add new mechanisms.
 * If "direction" is CRYPTO_MECH_REMOVED, remove the mechanism list.
 * Called from crypto_load_dev_disabled().
 */
static void
kcf_change_mechs(kcf_provider_desc_t *provider, uint_t count,
    crypto_mech_name_t *array, crypto_event_change_t direction)
{
        crypto_notify_event_change_t ec;
        crypto_mech_info_t *mi;
        kcf_prov_mech_desc_t *pmd;
        char *mech;
        int i, j, n;

        ASSERT(direction == CRYPTO_MECH_ADDED ||
            direction == CRYPTO_MECH_REMOVED);

        if (provider == NULL) {
                /*
                 * Nothing to add or remove from the tables since
                 * the provider isn't registered.
                 */
                return;
        }

        for (i = 0; i < count; i++) {
                if (array[i][0] == '\0')
                        continue;
```

```

                mech = &array[i][0];

                n = provider->pd_mech_list_count;
                for (j = 0; j < n; j++) {
                        mi = &provider->pd_mechanisms[j];
                        if (strncmp(mi->cm_mech_name, mech,
                            CRYPTO_MAX_MECH_NAME) == 0)
                                break;
                }
                if (j == n)
                        continue;

                switch (direction) {
                case CRYPTO_MECH_ADDED:
                        (void) kcf_add_mech_provider(j, provider, &pmd);
                        break;

                case CRYPTO_MECH_REMOVED:
                        kcf_remove_mech_provider(mech, provider);
                        break;
                }

                /* Inform interested clients of the event */
                ec.ec_provider_type = provider->pd_prov_type;
                ec.ec_change = direction;

                (void) strncpy(ec.ec_mech_name, mech, CRYPTO_MAX_MECH_NAME);
                kcf_walk_ntfylist(CRYPTO_EVENT_MECHS_CHANGED, &ec);
        }
}
```

______***unchanged_portion_omitted_***

```
/*
 * Called from CRYPTO_LOAD_SOFT_DISABLED ioctl.
 * If new_count is 0, then completely remove the entry.
 */
int
crypto_load_soft_disabled(char *name, uint_t new_count,
    crypto_mech_name_t *new_array)
{
        kcf_provider_desc_t *provider = NULL;
        crypto_mech_name_t *prev_array;
        uint_t prev_count = 0;
        int rv;

        provider = kcf_prov_tab_lookup_by_name(name);
        if (provider != NULL) {
                mutex_enter(&provider->pd_lock);
                /*
                 * Check if any other thread is disabling or removing
                 * this provider. We return if this is the case.
                 */
                if (provider->pd_state >= KCF_PROV_DISABLED) {
                        mutex_exit(&provider->pd_lock);
                        KCF_PROV_REFRELE(provider);
                        return (CRYPTO_BUSY);
                }
                provider->pd_state = KCF_PROV_DISABLED;
                mutex_exit(&provider->pd_lock);

                undo_register_provider(provider, B_TRUE);
                KCF_PROV_REFRELE(provider);
                if (provider->pd_kstat != NULL)
                        KCF_PROV_REFRELE(provider);
```

```
		mutex_enter(&provider->pd_lock);
		/* Wait till the existing requests complete. */
		while (provider->pd_state != KCF_PROV_FREED) {
			cv_wait(&provider->pd_remove_cv, &provider->pd_lock);
		}
		mutex_exit(&provider->pd_lock);
	}

	if (new_count == 0) {
		kcf_policy_remove_by_name(name, &prev_count, &prev_array);
		crypto_free_mech_list(prev_array, prev_count);
		rv = CRYPTO_SUCCESS;
		goto out;
	}

	/* put disabled mechanisms into policy table */
	if ((rv = kcf_policy_load_soft_disabled(name, new_count, new_array,
	    &prev_count, &prev_array)) == CRYPTO_SUCCESS) {
		crypto_free_mech_list(prev_array, prev_count);
	}

out:
	if (provider != NULL) {
		redo_register_provider(provider);
		if (provider->pd_kstat != NULL)
			KCF_PROV_REFHOLD(provider);
		mutex_enter(&provider->pd_lock);
		provider->pd_state = KCF_PROV_READY;
		mutex_exit(&provider->pd_lock);
	} else if (rv == CRYPTO_SUCCESS) {
		/*
		 * There are some cases where it is useful to kCF clients
		 * to have a provider whose mechanism is enabled now to be
		 * available. So, we attempt to load it here.
		 *
		 * The check, new_count < prev_count, ensures that we do this
		 * only in the case where a mechanism(s) is now enabled.
		 * This check assumes that enable and disable are separate
		 * administrative actions and are not done in a single action.
		 */
		if ((new_count < prev_count) &&
		if (new_count < prev_count && (in_soft_config_list(name)) &&
		    (modload("crypto", name) != -1)) {
			struct modctl *mcp;
			boolean_t load_again = B_FALSE;

			if ((mcp = mod_hold_by_name(name)) != NULL) {
				mcp->mod_loadflags |= MOD_NOAUTOUNLOAD;

				/* memory pressure may have unloaded module */
				if (!mcp->mod_installed)
					load_again = B_TRUE;
				mod_release_mod(mcp);

				if (load_again)
					(void) modload("crypto", name);
			}
		}
	}

	return (rv);
}
```
______***unchanged_portion_omitted_***

```
/*
 * Unload a kernel software crypto module.
```

```
 * Called from the CRYPTO_UNLOAD_SOFT_MODULE ioctl.
 */
/* called from the CRYPTO_UNLOAD_SOFT_MODULE ioctl */
int
crypto_unload_soft_module(caddr_t name)
{
	int error;
	modid_t id;
	kcf_provider_desc_t *provider;
	struct modctl *mcp;

	/* verify that 'name' refers to a registered crypto provider */
	if ((provider = kcf_prov_tab_lookup_by_name(name)) == NULL)
		return (CRYPTO_UNKNOWN_PROVIDER);

	/*
	 * We save the module id and release the reference. We need to
	 * do this as modunload() calls unregister which waits for the
	 * refcnt to drop to zero.
	 */
	id = provider->pd_module_id;
	KCF_PROV_REFRELE(provider);

	if ((mcp = mod_hold_by_name(name)) != NULL) {
		mcp->mod_loadflags &= ~(MOD_NOAUTOUNLOAD);
		mod_release_mod(mcp);
	}

	if ((error = modunload(id)) != 0) {
		return (error == EBUSY ? CRYPTO_BUSY : CRYPTO_FAILED);
	}

	return (CRYPTO_SUCCESS);
}

/*
 * Free the list of kernel hardware crypto providers.
 * Called by get_dev_list() for the CRYPTO_GET_DEV_LIST ioctl.
 */
/* called from CRYPTO_GET_DEV_LIST ioctl */
void
crypto_free_dev_list(crypto_dev_list_entry_t *array, uint_t count)
{
	if (count == 0 || array == NULL)
		return;

	kmem_free(array, count * sizeof (crypto_dev_list_entry_t));
}
```
______***unchanged_portion_omitted_***

```
/*
 * Free memory for elements in a kcf_soft_config_entry_t.  This entry must
 * have been previously removed from the soft_config_list linked list.
 */
static void
free_soft_config_entry(kcf_soft_conf_entry_t *p)
{
	kmem_free(p->ce_name, strlen(p->ce_name) + 1);
	crypto_free_mech_list(p->ce_mechs, p->ce_count);
	kmem_free(p, sizeof (kcf_soft_conf_entry_t));
}

/*
 * Store configuration information for software providers in a linked list.
 * Called from the CRYPTO_LOAD_SOFT_CONFIG ioctl, this routine stores
 * configuration information for software providers in a linked list.
 * If the list already contains an entry for the specified provider
 * and the specified mechanism list has at least one mechanism, then
 * the mechanism list for the provider is updated. If the mechanism list
 * is empty, the entry for the provider is removed.
 *
 * Called from kcf_soft_config_init() (to initially populate the list
 * with default kernel providers) and from crypto_load_soft_config() for
 * the CRYPTO_LOAD_SOFT_CONFIG ioctl (for third-party kernel modules).
 *
 * Important note: the name and array arguments must be allocated memory
 * and are consumed in soft_config_list.
 * Important note: the array argument is consumed.
 */
static int
add_soft_config(char *name, uint_t count, crypto_mech_name_t *array)
{
        static uint_t soft_config_count = 0;
        kcf_soft_conf_entry_t *prev = NULL, *entry = NULL, *new_entry, *p;
        size_t name_len;

        /*
         * Allocate storage for a new entry.
         * Free later if an entry already exists.
         */
        name_len = strlen(name) + 1;
        new_entry = kmem_zalloc(sizeof (kcf_soft_conf_entry_t), KM_SLEEP);
        new_entry->ce_name = kmem_alloc(name_len, KM_SLEEP);
        (void) strcpy(new_entry->ce_name, name);

        mutex_enter(&soft_config_mutex);
        p = soft_config_list;
        if (p != NULL) {
                do {
                        if (strncmp(name, p->ce_name, MAXNAMELEN) == 0) {
                                entry = p;
                                break;
                        }
                        prev = p;

                } while ((p = p->ce_next) != NULL);
        }

        if (entry == NULL) {
                if (count == 0) {
                        mutex_exit(&soft_config_mutex);
                        kmem_free(new_entry->ce_name, name_len);
                        kmem_free(new_entry, sizeof (kcf_soft_conf_entry_t));
                        return (CRYPTO_SUCCESS);
                }

                if (soft_config_count > KCF_MAX_CONFIG_ENTRIES) {
                        mutex_exit(&soft_config_mutex);
                        kmem_free(new_entry->ce_name, name_len);
                        kmem_free(new_entry, sizeof (kcf_soft_conf_entry_t));
                        cmn_err(CE_WARN, "out of soft_config_list entries");
                        return (CRYPTO_FAILED);
                }

                /* add to head of list */
                new_entry->ce_next = soft_config_list;
                soft_config_list = new_entry;
                soft_config_count++;
                entry = new_entry;
        } else { /* mechanism already in list */
        } else {
                kmem_free(new_entry->ce_name, name_len);
```

```
                kmem_free(new_entry, sizeof (kcf_soft_conf_entry_t));
        }

        /* mechanism count == 0 means remove entry from list */
        if (count == 0) {
                if (prev == NULL) {
                        /* remove first in list */
                        soft_config_list = entry->ce_next;
                } else {
                        prev->ce_next = entry->ce_next;
                }
                soft_config_count--;
                mutex_exit(&soft_config_mutex);

                /* free entry */
                free_soft_config_entry(entry);

                return (CRYPTO_SUCCESS);
        }


        /* replace mechanisms */
        if (entry->ce_mechs != NULL)
                crypto_free_mech_list(entry->ce_mechs, entry->ce_count);

        entry->ce_mechs = array;
        entry->ce_count = count;
        mutex_exit(&soft_config_mutex);

        return (CRYPTO_SUCCESS);
}

/*
 * This routine searches the soft_config_list for the first entry that
 * has the specified mechanism in its mechanism list.  If found,
 * a buffer containing the name of the software module that implements
 * the mechanism is allocated and stored in ’name’.
 */
int
get_sw_provider_for_mech(crypto_mech_name_t mech, char **name)
{
        kcf_soft_conf_entry_t *p, *next;
        char tmp_name[MAXNAMELEN];
        size_t name_len = 0;
        int i;

        mutex_enter(&soft_config_mutex);
        p = soft_config_list;
        while (p != NULL) {
                next = p->ce_next;
                for (i = 0; i < p->ce_count; i++) {
                        if (strcmp(mech, &p->ce_mechs[i][0]) == 0) {
                                name_len = strlen(p->ce_name) + 1;
                                bcopy(p->ce_name, tmp_name, name_len);
                                break;
                        }
                }
                p = next;
        }
        mutex_exit(&soft_config_mutex);

        if (name_len == 0)
                return (CRYPTO_FAILED);

        *name = kmem_alloc(name_len, KM_SLEEP);
        bcopy(tmp_name, *name, name_len);
        return (CRYPTO_SUCCESS);
}

/*
 * This routine searches the soft_config_list for the specified
 * software provider, returning B_TRUE if it is in the list.
 */
boolean_t
in_soft_config_list(char *provider_name)
{
        kcf_soft_conf_entry_t *p;
        boolean_t rv = B_FALSE;

        mutex_enter(&soft_config_mutex);
        for (p = soft_config_list; p != NULL; p = p->ce_next) {
                if (strcmp(provider_name, p->ce_name) == 0) {
                        rv = B_TRUE;
                        break;
                }
        }
        mutex_exit(&soft_config_mutex);
        return (rv);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   25851 Tue Oct 28 16:46:23 2008
new/usr/src/uts/common/crypto/core/kcf_prov_tabs.c
6414175 kcf.conf's supportedlist not providing much usefulness
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * This file is part of the core Kernel Cryptographic Framework.
 * It implements the management of tables of Providers. Entries to
 * added and removed when cryptographic providers register with
 * and unregister from the framework, respectively. The KCF scheduler
 * and ioctl pseudo driver call this function to obtain the list
 * of available providers.
 *
 * The provider table is indexed by crypto_provider_id_t. Each
 * element of the table contains a pointer to a provider descriptor,
 * or NULL if the entry is free.
 *
 * This file also implements helper functions to allocate and free
 * provider descriptors.
 */

#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/ksynch.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>
#include <sys/crypto/sched_impl.h>
#include <sys/crypto/spi.h>

#define KCF_MAX_PROVIDERS       512     /* max number of providers */

/*
 * Prov_tab is an array of providers which is updated when
 * a crypto provider registers with kcf. The provider calls the
 * SPI routine, crypto_register_provider(), which in turn calls
```

```
 * kcf_prov_tab_add_provider().
 *
 * A provider unregisters by calling crypto_unregister_provider()
 * which triggers the removal of the prov_tab entry.
 * It also calls kcf_remove_mech_provider().
 *
 * prov_tab entries are not updated from kcf.conf or by cryptoadm(1M).
 */
static kcf_provider_desc_t **prov_tab = NULL;
static kmutex_t prov_tab_mutex; /* ensure exclusive access to the table */
static kcf_provider_desc_t **prov_tab = NULL;
static uint_t prov_tab_num = 0; /* number of providers in table */
static uint_t prov_tab_max = KCF_MAX_PROVIDERS;

#if DEBUG
extern int kcf_frmwrk_debug;
static void kcf_prov_tab_dump(char *message);
static void kcf_prov_tab_dump(void);
#endif /* DEBUG */


/*
 * Initialize a mutex and the KCF providers table, prov_tab.
 * The providers table is dynamically allocated with prov_tab_max entries.
 * Called from kcf module _init().
 * Initialize the providers table. The providers table is dynamically
 * allocated with prov_tab_max entries.
 */
void
kcf_prov_tab_init(void)
{
        mutex_init(&prov_tab_mutex, NULL, MUTEX_DRIVER, NULL);

        prov_tab = kmem_zalloc(prov_tab_max * sizeof (kcf_provider_desc_t *),
            KM_SLEEP);
}

/*
 * Add a provider to the provider table. If no free entry can be found
 * for the new provider, returns CRYPTO_HOST_MEMORY. Otherwise, add
 * the provider to the table, initialize the pd_prov_id field
 * of the specified provider descriptor to the index in that table,
 * and return CRYPTO_SUCCESS. Note that a REFHOLD is done on the
 * provider when pointed to by a table entry.
 */
int
kcf_prov_tab_add_provider(kcf_provider_desc_t *prov_desc)
{
        uint_t i;

        ASSERT(prov_tab != NULL);

        mutex_enter(&prov_tab_mutex);

        /* find free slot in providers table */
        for (i = 0; i < KCF_MAX_PROVIDERS && prov_tab[i] != NULL; i++)
                ;
        if (i == KCF_MAX_PROVIDERS) {
                /* ran out of providers entries */
                mutex_exit(&prov_tab_mutex);
                cmn_err(CE_WARN, "out of providers entries");
                return (CRYPTO_HOST_MEMORY);
        }

        /* initialize entry */
        prov_tab[i] = prov_desc;
        KCF_PROV_REFHOLD(prov_desc);
        KCF_PROV_IREFHOLD(prov_desc);
        prov_tab_num++;

        mutex_exit(&prov_tab_mutex);

        /* update provider descriptor */
        prov_desc->pd_prov_id = i;

        /*
         * The KCF-private provider handle is defined as the internal
         * provider id.
         */
        prov_desc->pd_kcf_prov_handle =
            (crypto_kcf_provider_handle_t)prov_desc->pd_prov_id;

#if DEBUG
        if (kcf_frmwrk_debug >= 1)
                kcf_prov_tab_dump("kcf_prov_tab_add_provider");
                kcf_prov_tab_dump();
#endif /* DEBUG */

        return (CRYPTO_SUCCESS);
}

/*
 * Remove the provider specified by its id. A REFRELE is done on the
 * corresponding provider descriptor before this function returns.
 * Returns CRYPTO_UNKNOWN_PROVIDER if the provider id is not valid.
 */
int
kcf_prov_tab_rem_provider(crypto_provider_id_t prov_id)
{
        kcf_provider_desc_t *prov_desc;

        ASSERT(prov_tab != NULL);
        ASSERT(prov_tab_num >= 0);

        /*
         * Validate provider id, since it can be specified by a 3rd-party
         * provider.
         */

        mutex_enter(&prov_tab_mutex);
        if (prov_id >= KCF_MAX_PROVIDERS ||
            ((prov_desc = prov_tab[prov_id]) == NULL)) {
                mutex_exit(&prov_tab_mutex);
                return (CRYPTO_INVALID_PROVIDER_ID);
        }
        mutex_exit(&prov_tab_mutex);

        /*
         * The provider id must remain valid until the associated provider
         * descriptor is freed. For this reason, we simply release our
         * reference to the descriptor here. When the reference count
         * reaches zero, kcf_free_provider_desc() will be invoked and
         * the associated entry in the providers table will be released
         * at that time.
         */

        KCF_PROV_REFRELE(prov_desc);
        KCF_PROV_IREFRELE(prov_desc);

#if DEBUG
        if (kcf_frmwrk_debug >= 1)
                kcf_prov_tab_dump("kcf_prov_tab_rem_provider");
```

```
                kcf_prov_tab_dump();
#endif /* DEBUG */

        return (CRYPTO_SUCCESS);
}
```
______unchanged_portion_omitted_

```
#if DEBUG
/*
 * Dump the Kernel crypto providers table, prov_tab.
 * If kcf_frmwrk_debug is >=2, also dump the mechanism lists.
 */

static void
kcf_prov_tab_dump(char *message)
kcf_prov_tab_dump(void)
{
        uint_t i, j;
        uint_t i;

        mutex_enter(&prov_tab_mutex);
        printf("Providers table prov_tab at %s:\n",
            message != NULL ? message : "");

        printf("Providers table:\n");
        for (i = 0; i < KCF_MAX_PROVIDERS; i++) {
                kcf_provider_desc_t *p = prov_tab[i];
                if (p != NULL) {
                        printf("[%d]: (%s) %d mechanisms, %s\n", i,
                            (p->pd_prov_type == CRYPTO_HW_PROVIDER) ?
                            "HW" : "SW",
                            p->pd_mech_list_count, p->pd_description);
                        if (kcf_frmwrk_debug >= 2) {
                                printf("\tpd_mechanisms: ");
                                for (j = 0; j < p->pd_mech_list_count; ++j) {
                                        printf("%s \n",
                                            p->pd_mechanisms[j].cm_mech_name);
                if (prov_tab[i] != NULL) {
                        printf("[%d]: (%s) %s\n",
                            i, (prov_tab[i]->pd_prov_type ==
                            CRYPTO_HW_PROVIDER) ? "HW" : "SW",
                            prov_tab[i]->pd_description);
                                }
                                printf("\n");
                        }
                }
        }
        printf("(end of providers table)\n");

        mutex_exit(&prov_tab_mutex);
}
```
______unchanged_portion_omitted_