```
**********************************************************
   32686 Fri Jun 14 15:23:20 2013
new/usr/src/uts/common/os/cred.c
3691 setgroups() needs a sorted GID list for more than 16 groups
Reviewed-By: Marcel Telka <marcel@telka.sk>
Reviewed-By: Richard Lowe <richlowe@richlowe.net>
**********************************************************
```

```
/*
 * 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 (c) 2013, Ira Cooper.  All rights reserved.
 */
/*
#endif /* ! codereview */
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/cred_impl.h>
#include <sys/policy.h>
#include <sys/vnode.h>
#include <sys/errno.h>
#include <sys/kmem.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/syscall.h>
#include <sys/debug.h>
#include <sys/atomic.h>
#include <sys/ucred.h>
#include <sys/prsystm.h>
#include <sys/modctl.h>
#include <sys/avl.h>
```



```
#include <sys/door.h>
#include <c2/audit.h>
#include <sys/zone.h>
#include <sys/tsol/label.h>
#include <sys/sid.h>
#include <sys/idmap.h>
#include <sys/klpd.h>
#include <sys/varargs.h>
#include <sys/sysconf.h>
#include <util/qsort.h>


/* Ephemeral IDs Zones specific data */
typedef struct ephemeral_zsd {
        uid_t           min_uid;
        uid_t           last_uid;
        gid_t           min_gid;
        gid_t           last_gid;
        kmutex_t        eph_lock;
        cred_t          *eph_nobody;
} ephemeral_zsd_t;

static void crgrphold(credgrp_t *);

#define CREDGRPSZ(ngrp) (sizeof (credgrp_t) + ((ngrp - 1) * sizeof (gid_t)))

static kmutex_t         ephemeral_zone_mutex;
static zone_key_t       ephemeral_zone_key;

static struct kmem_cache *cred_cache;
static size_t           crsize = 0;
static int              audoff = 0;
uint32_t                ucredsize;
cred_t                  *kcred;
static cred_t           *dummycr;

int rstlink;            /* link(2) restricted to files owned by user? */

static int get_c2audit_load(void);

#define CR_AUINFO(c)    (auditinfo_addr_t *)((audoff == 0) ? NULL : \
                            ((char *)(c)) + audoff)

#define REMOTE_PEER_CRED(c)     ((c)->cr_gid == -1)

#define BIN_GROUP_SEARCH_CUTOFF 16

static boolean_t hasephids = B_FALSE;

static ephemeral_zsd_t *
get_ephemeral_zsd(zone_t *zone)
{
        ephemeral_zsd_t *eph_zsd;

        eph_zsd = zone_getspecific(ephemeral_zone_key, zone);
        if (eph_zsd != NULL) {
                return (eph_zsd);
        }

        mutex_enter(&ephemeral_zone_mutex);
        eph_zsd = zone_getspecific(ephemeral_zone_key, zone);
        if (eph_zsd == NULL) {
                eph_zsd = kmem_zalloc(sizeof (ephemeral_zsd_t), KM_SLEEP);
                eph_zsd->min_uid = MAXUID;
                eph_zsd->last_uid = IDMAP_WK__MAX_UID;
                eph_zsd->min_gid = MAXUID;
```

```
126                 eph_zsd->last_gid = IDMAP_WK__MAX_GID;
127                 mutex_init(&eph_zsd->eph_lock, NULL, MUTEX_DEFAULT, NULL);

129                 /*
130                  * nobody is used to map SID containing CRs.
131                  */
132                 eph_zsd->eph_nobody = crdup(zone->zone_kcred);
133                 (void) crsetugid(eph_zsd->eph_nobody, UID_NOBODY, GID_NOBODY);
134                 CR_FLAGS(eph_zsd->eph_nobody) = 0;
135                 eph_zsd->eph_nobody->cr_zone = zone;

137                 (void) zone_setspecific(ephemeral_zone_key, zone, eph_zsd);
138         }
139         mutex_exit(&ephemeral_zone_mutex);
140         return (eph_zsd);
141 }

143 static cred_t *crdup_flags(const cred_t *, int);
144 static cred_t *cralloc_flags(int);

146 /*
147  * This function is called when a zone is destroyed
148  */
149 static void
150 /* ARGSUSED */
151 destroy_ephemeral_zsd(zoneid_t zone_id, void *arg)
152 {
153         ephemeral_zsd_t *eph_zsd = arg;
154         if (eph_zsd != NULL) {
155                 mutex_destroy(&eph_zsd->eph_lock);
156                 crfree(eph_zsd->eph_nobody);
157                 kmem_free(eph_zsd, sizeof (ephemeral_zsd_t));
158         }
159 }



163 /*
164  * Initialize credentials data structures.
165  */

167 void
168 cred_init(void)
169 {
170         priv_init();

172         crsize = sizeof (cred_t);

174         if (get_c2audit_load() > 0) {
175 #ifdef _LP64
176                 /* assure audit context is 64-bit aligned */
177                 audoff = (crsize +
178                     sizeof (int64_t) - 1) & ~(sizeof (int64_t) - 1);
179 #else   /* _LP64 */
180                 audoff = crsize;
181 #endif  /* _LP64 */
182                 crsize = audoff + sizeof (auditinfo_addr_t);
183                 crsize = (crsize + sizeof (int) - 1) & ~(sizeof (int) - 1);
184         }

186         cred_cache = kmem_cache_create("cred_cache", crsize, 0,
187             NULL, NULL, NULL, NULL, NULL, 0);

189         /*
190          * dummycr is used to copy initial state for creds.
191          */
```



```
192         dummycr = cralloc();
193         bzero(dummycr, crsize);
194         dummycr->cr_ref = 1;
195         dummycr->cr_uid = (uid_t)-1;
196         dummycr->cr_gid = (gid_t)-1;
197         dummycr->cr_ruid = (uid_t)-1;
198         dummycr->cr_rgid = (gid_t)-1;
199         dummycr->cr_suid = (uid_t)-1;
200         dummycr->cr_sgid = (gid_t)-1;


203         /*
204          * kcred is used by anything that needs all privileges; it's
205          * also the template used for crget as it has all the compatible
206          * sets filled in.
207          */
208         kcred = cralloc();

210         bzero(kcred, crsize);
211         kcred->cr_ref = 1;

213         /* kcred is never freed, so we don't need zone_cred_hold here */
214         kcred->cr_zone = &zone0;

216         priv_fillset(&CR_LPRIV(kcred));
217         CR_IPRIV(kcred) = *priv_basic;

219         /* Not a basic privilege, if chown is not restricted add it to I0 */
220         if (!rstchown)
221                 priv_addset(&CR_IPRIV(kcred), PRIV_FILE_CHOWN_SELF);

223         /* Basic privilege, if link is restricted remove it from I0 */
224         if (rstlink)
225                 priv_delset(&CR_IPRIV(kcred), PRIV_FILE_LINK_ANY);

227         CR_EPRIV(kcred) = CR_PPRIV(kcred) = CR_IPRIV(kcred);

229         CR_FLAGS(kcred) = NET_MAC_AWARE;

231         /*
232          * Set up credentials of p0.
233          */
234         ttoproc(curthread)->p_cred = kcred;
235         curthread->t_cred = kcred;

237         ucredsize = UCRED_SIZE;

239         mutex_init(&ephemeral_zone_mutex, NULL, MUTEX_DEFAULT, NULL);
240         zone_key_create(&ephemeral_zone_key, NULL, NULL, destroy_ephemeral_zsd);
241 }

243 /*
244  * Allocate (nearly) uninitialized cred_t.
245  */
246 static cred_t *
247 cralloc_flags(int flgs)
248 {
249         cred_t *cr = kmem_cache_alloc(cred_cache, flgs);

251         if (cr == NULL)
252                 return (NULL);

254         cr->cr_ref = 1;         /* So we can crfree() */
255         cr->cr_zone = NULL;
256         cr->cr_label = NULL;
257         cr->cr_ksid = NULL;
```

```
258         cr->cr_klpd = NULL;
259         cr->cr_grps = NULL;
260         return (cr);
261 }

263 cred_t *
264 cralloc(void)
265 {
266         return (cralloc_flags(KM_SLEEP));
267 }

269 /*
270  * As cralloc but prepared for ksid change (if appropriate).
271  */
272 cred_t *
273 cralloc_ksid(void)
274 {
275         cred_t *cr = cralloc();
276         if (hasephids)
277                 cr->cr_ksid = kcrsid_alloc();
278         return (cr);
279 }

281 /*
282  * Allocate a initialized cred structure and crhold() it.
283  * Initialized means: all ids 0, group count 0, L=Full, E=P=I=I0
284  */
285 cred_t *
286 crget(void)
287 {
288         cred_t *cr = kmem_cache_alloc(cred_cache, KM_SLEEP);

290         bcopy(kcred, cr, crsize);
291         cr->cr_ref = 1;
292         zone_cred_hold(cr->cr_zone);
293         if (cr->cr_label)
294                 label_hold(cr->cr_label);
295         ASSERT(cr->cr_klpd == NULL);
296         ASSERT(cr->cr_grps == NULL);
297         return (cr);
298 }

300 /*
301  * Broadcast the cred to all the threads in the process.
302  * The current thread's credentials can be set right away, but other
303  * threads must wait until the start of the next system call or trap.
304  * This avoids changing the cred in the middle of a system call.
305  *
306  * The cred has already been held for the process and the thread (2 holds),
307  * and p->p_cred set.
308  *
309  * p->p_crlock shouldn't be held here, since p_lock must be acquired.
310  */
311 void
312 crset(proc_t *p, cred_t *cr)
313 {
314         kthread_id_t    t;
315         kthread_id_t    first;
316         cred_t *oldcr;

318         ASSERT(p == curproc);   /* assumes p_lwpcnt can't change */

320         /*
321          * DTrace accesses t_cred in probe context.  t_cred must always be
322          * either NULL, or point to a valid, allocated cred structure.
323          */
```



```
324         t = curthread;
325         oldcr = t->t_cred;
326         t->t_cred = cr;         /* the cred is held by caller for this thread */
327         crfree(oldcr);          /* free the old cred for the thread */

329         /*
330          * Broadcast to other threads, if any.
331          */
332         if (p->p_lwpcnt > 1) {
333                 mutex_enter(&p->p_lock);        /* to keep thread list safe */
334                 first = curthread;
335                 for (t = first->t_forw; t != first; t = t->t_forw)
336                         t->t_pre_sys = 1; /* so syscall will get new cred */
337                 mutex_exit(&p->p_lock);
338         }
339 }

341 /*
342  * Put a hold on a cred structure.
343  */
344 void
345 crhold(cred_t *cr)
346 {
347         ASSERT(cr->cr_ref != 0xdeadbeef && cr->cr_ref != 0);
348         atomic_add_32(&cr->cr_ref, 1);
349 }

351 /*
352  * Release previous hold on a cred structure.  Free it if refcnt == 0.
353  * If cred uses label different from zone label, free it.
354  */
355 void
356 crfree(cred_t *cr)
357 {
358         ASSERT(cr->cr_ref != 0xdeadbeef && cr->cr_ref != 0);
359         if (atomic_add_32_nv(&cr->cr_ref, -1) == 0) {
360                 ASSERT(cr != kcred);
361                 if (cr->cr_label)
362                         label_rele(cr->cr_label);
363                 if (cr->cr_klpd)
364                         crklpd_rele(cr->cr_klpd);
365                 if (cr->cr_zone)
366                         zone_cred_rele(cr->cr_zone);
367                 if (cr->cr_ksid)
368                         kcrsid_rele(cr->cr_ksid);
369                 if (cr->cr_grps)
370                         crgrprele(cr->cr_grps);

372                 kmem_cache_free(cred_cache, cr);
373         }
374 }

376 /*
377  * Copy a cred structure to a new one and free the old one.
378  *      The new cred will have two references.  One for the calling process,
379  *      and one for the thread.
380  */
381 cred_t *
382 crcopy(cred_t *cr)
383 {
384         cred_t *newcr;

386         newcr = cralloc();
387         bcopy(cr, newcr, crsize);
388         if (newcr->cr_zone)
389                 zone_cred_hold(newcr->cr_zone);
```

```
390         if (newcr->cr_label)
391                 label_hold(newcr->cr_label);
392         if (newcr->cr_ksid)
393                 kcrsid_hold(newcr->cr_ksid);
394         if (newcr->cr_klpd)
395                 crklpd_hold(newcr->cr_klpd);
396         if (newcr->cr_grps)
397                 crgrphold(newcr->cr_grps);
398         crfree(cr);
399         newcr->cr_ref = 2;              /* caller gets two references */
400         return (newcr);
401 }

403 /*
404  * Copy a cred structure to a new one and free the old one.
405  *      The new cred will have two references.  One for the calling process,
406  *      and one for the thread.
407  * This variation on crcopy uses a pre-allocated structure for the
408  * "new" cred.
409  */
410 void
411 crcopy_to(cred_t *oldcr, cred_t *newcr)
412 {
413         credsid_t *nkcr = newcr->cr_ksid;

415         bcopy(oldcr, newcr, crsize);
416         if (newcr->cr_zone)
417                 zone_cred_hold(newcr->cr_zone);
418         if (newcr->cr_label)
419                 label_hold(newcr->cr_label);
420         if (newcr->cr_klpd)
421                 crklpd_hold(newcr->cr_klpd);
422         if (newcr->cr_grps)
423                 crgrphold(newcr->cr_grps);
424         if (nkcr) {
425                 newcr->cr_ksid = nkcr;
426                 kcrsidcopy_to(oldcr->cr_ksid, newcr->cr_ksid);
427         } else if (newcr->cr_ksid)
428                 kcrsid_hold(newcr->cr_ksid);
429         crfree(oldcr);
430         newcr->cr_ref = 2;              /* caller gets two references */
431 }

433 /*
434  * Dup a cred struct to a new held one.
435  *      The old cred is not freed.
436  */
437 static cred_t *
438 crdup_flags(const cred_t *cr, int flgs)
439 {
440         cred_t *newcr;

442         newcr = cralloc_flags(flgs);

444         if (newcr == NULL)
445                 return (NULL);

447         bcopy(cr, newcr, crsize);
448         if (newcr->cr_zone)
449                 zone_cred_hold(newcr->cr_zone);
450         if (newcr->cr_label)
451                 label_hold(newcr->cr_label);
452         if (newcr->cr_klpd)
453                 crklpd_hold(newcr->cr_klpd);
454         if (newcr->cr_ksid)
455                 kcrsid_hold(newcr->cr_ksid);
```



```
456         if (newcr->cr_grps)
457                 crgrphold(newcr->cr_grps);
458         newcr->cr_ref = 1;
459         return (newcr);
460 }

462 cred_t *
463 crdup(cred_t *cr)
464 {
465         return (crdup_flags(cr, KM_SLEEP));
466 }

468 /*
469  * Dup a cred struct to a new held one.
470  *      The old cred is not freed.
471  * This variation on crdup uses a pre-allocated structure for the
472  * "new" cred.
473  */
474 void
475 crdup_to(cred_t *oldcr, cred_t *newcr)
476 {
477         credsid_t *nkcr = newcr->cr_ksid;

479         bcopy(oldcr, newcr, crsize);
480         if (newcr->cr_zone)
481                 zone_cred_hold(newcr->cr_zone);
482         if (newcr->cr_label)
483                 label_hold(newcr->cr_label);
484         if (newcr->cr_klpd)
485                 crklpd_hold(newcr->cr_klpd);
486         if (newcr->cr_grps)
487                 crgrphold(newcr->cr_grps);
488         if (nkcr) {
489                 newcr->cr_ksid = nkcr;
490                 kcrsidcopy_to(oldcr->cr_ksid, newcr->cr_ksid);
491         } else if (newcr->cr_ksid)
492                 kcrsid_hold(newcr->cr_ksid);
493         newcr->cr_ref = 1;
494 }

496 /*
497  * Return the (held) credentials for the current running process.
498  */
499 cred_t *
500 crgetcred(void)
501 {
502         cred_t *cr;
503         proc_t *p;

505         p = ttoproc(curthread);
506         mutex_enter(&p->p_crlock);
507         crhold(cr = p->p_cred);
508         mutex_exit(&p->p_crlock);
509         return (cr);
510 }

512 /*
513  * Backward compatibility check for suser().
514  * Accounting flag is now set in the policy functions; auditing is
515  * done through use of privilege in the audit trail.
516  */
517 int
518 suser(cred_t *cr)
519 {
520         return (PRIV_POLICY(cr, PRIV_SYS_SUSER_COMPAT, B_FALSE, EPERM, NULL)
521             == 0);
```

```
}

/*
 * Determine whether the supplied group id is a member of the group
 * described by the supplied credentials.
 */
int
groupmember(gid_t gid, const cred_t *cr)
{
        if (gid == cr->cr_gid)
                return (1);
        return (supgroupmember(gid, cr));
}

/*
 * As groupmember but only check against the supplemental groups.
 */
int
supgroupmember(gid_t gid, const cred_t *cr)
{
        int hi, lo;
        credgrp_t *grps = cr->cr_grps;
        const gid_t *gp, *endgp;

        if (grps == NULL)
                return (0);

        /* For a small number of groups, use sequentials search. */
        if (grps->crg_ngroups <= BIN_GROUP_SEARCH_CUTOFF) {
                endgp = &grps->crg_groups[grps->crg_ngroups];
                for (gp = grps->crg_groups; gp < endgp; gp++)
                        if (*gp == gid)
                                return (1);
                return (0);
        }

        /* We use binary search when we have many groups. */
        lo = 0;
        hi = grps->crg_ngroups - 1;
        gp = grps->crg_groups;

        do {
                int m = (lo + hi) / 2;

                if (gid > gp[m])
                        lo = m + 1;
                else if (gid < gp[m])
                        hi = m - 1;
                else
                        return (1);
        } while (lo <= hi);

        return (0);
}

/*
 * This function is called to check whether the credentials set
 * "scrp" has permission to act on credentials set "tcrp".  It enforces the
 * permission requirements needed to send a signal to a process.
 * The same requirements are imposed by other system calls, however.
 *
 * The rules are:
 * (1) if the credentials are the same, the check succeeds
 * (2) if the zone ids don't match, and scrp is not in the global zone or
 *     does not have the PRIV_PROC_ZONE privilege, the check fails
 * (3) if the real or effective user id of scrp matches the real or saved
```



```
 *     user id of tcrp or scrp has the PRIV_PROC_OWNER privilege, the check
 *     succeeds
 * (4) otherwise, the check fails
 */
int
hasprocperm(const cred_t *tcrp, const cred_t *scrp)
{
        if (scrp == tcrp)
                return (1);
        if (scrp->cr_zone != tcrp->cr_zone &&
            (scrp->cr_zone != global_zone ||
            secpolicy_proc_zone(scrp) != 0))
                return (0);
        if (scrp->cr_uid == tcrp->cr_ruid ||
            scrp->cr_ruid == tcrp->cr_ruid ||
            scrp->cr_uid  == tcrp->cr_suid ||
            scrp->cr_ruid == tcrp->cr_suid ||
            !PRIV_POLICY(scrp, PRIV_PROC_OWNER, B_FALSE, EPERM, "hasprocperm"))
                return (1);
        return (0);
}

/*
 * This interface replaces hasprocperm; it works like hasprocperm but
 * additionally returns success if the proc_t's match
 * It is the preferred interface for most uses.
 * And it will acquire p_crlock itself, so it assert's that it shouldn't
 * be held.
 */
int
prochasprocperm(proc_t *tp, proc_t *sp, const cred_t *scrp)
{
        int rets;
        cred_t *tcrp;

        ASSERT(MUTEX_NOT_HELD(&tp->p_crlock));

        if (tp == sp)
                return (1);

        if (tp->p_sessp != sp->p_sessp && secpolicy_basic_proc(scrp) != 0)
                return (0);

        mutex_enter(&tp->p_crlock);
        crhold(tcrp = tp->p_cred);
        mutex_exit(&tp->p_crlock);
        rets = hasprocperm(tcrp, scrp);
        crfree(tcrp);

        return (rets);
}

/*
 * This routine is used to compare two credentials to determine if
 * they refer to the same "user".  If the pointers are equal, then
 * they must refer to the same user.  Otherwise, the contents of
 * the credentials are compared to see whether they are equivalent.
 *
 * This routine returns 0 if the credentials refer to the same user,
 * 1 if they do not.
 */
int
crcmp(const cred_t *cr1, const cred_t *cr2)
{
        credgrp_t *grp1, *grp2;
```

```
654         if (cr1 == cr2)
655                 return (0);

657         if (cr1->cr_uid == cr2->cr_uid &&
658             cr1->cr_gid == cr2->cr_gid &&
659             cr1->cr_ruid == cr2->cr_ruid &&
660             cr1->cr_rgid == cr2->cr_rgid &&
661             cr1->cr_zone == cr2->cr_zone &&
662             ((grp1 = cr1->cr_grps) == (grp2 = cr2->cr_grps) ||
663             (grp1 != NULL && grp2 != NULL &&
664             grp1->crg_ngroups == grp2->crg_ngroups &&
665             bcmp(grp1->crg_groups, grp2->crg_groups,
666             grp1->crg_ngroups * sizeof (gid_t)) == 0))) {
667                 return (!priv_isequalset(&CR_OEPRIV(cr1), &CR_OEPRIV(cr2)));
668         }
669         return (1);
670 }

672 /*
673  * Read access functions to cred_t.
674  */
675 uid_t
676 crgetuid(const cred_t *cr)
677 {
678         return (cr->cr_uid);
679 }

681 uid_t
682 crgetruid(const cred_t *cr)
683 {
684         return (cr->cr_ruid);
685 }

687 uid_t
688 crgetsuid(const cred_t *cr)
689 {
690         return (cr->cr_suid);
691 }

693 gid_t
694 crgetgid(const cred_t *cr)
695 {
696         return (cr->cr_gid);
697 }

699 gid_t
700 crgetrgid(const cred_t *cr)
701 {
702         return (cr->cr_rgid);
703 }

705 gid_t
706 crgetsgid(const cred_t *cr)
707 {
708         return (cr->cr_sgid);
709 }

711 const auditinfo_addr_t *
712 crgetauinfo(const cred_t *cr)
713 {
714         return ((const auditinfo_addr_t *)CR_AUINFO(cr));
715 }

717 auditinfo_addr_t *
718 crgetauinfo_modifiable(cred_t *cr)
719 {
```



```
720         return (CR_AUINFO(cr));
721 }

723 zoneid_t
724 crgetzoneid(const cred_t *cr)
725 {
726         return (cr->cr_zone == NULL ?
727             (cr->cr_uid == -1 ? (zoneid_t)-1 : GLOBAL_ZONEID) :
728             cr->cr_zone->zone_id);
729 }

731 projid_t
732 crgetprojid(const cred_t *cr)
733 {
734         return (cr->cr_projid);
735 }

737 zone_t *
738 crgetzone(const cred_t *cr)
739 {
740         return (cr->cr_zone);
741 }

743 struct ts_label_s *
744 crgetlabel(const cred_t *cr)
745 {
746         return (cr->cr_label ?
747             cr->cr_label :
748             (cr->cr_zone ? cr->cr_zone->zone_slabel : NULL));
749 }

751 boolean_t
752 crisremote(const cred_t *cr)
753 {
754         return (REMOTE_PEER_CRED(cr));
755 }

757 #define BADUID(x, zn)   ((x) != -1 && !VALID_UID((x), (zn)))
758 #define BADGID(x, zn)   ((x) != -1 && !VALID_GID((x), (zn)))

760 int
761 crsetresuid(cred_t *cr, uid_t r, uid_t e, uid_t s)
762 {
763         zone_t  *zone = crgetzone(cr);

765         ASSERT(cr->cr_ref <= 2);

767         if (BADUID(r, zone) || BADUID(e, zone) || BADUID(s, zone))
768                 return (-1);

770         if (r != -1)
771                 cr->cr_ruid = r;
772         if (e != -1)
773                 cr->cr_uid = e;
774         if (s != -1)
775                 cr->cr_suid = s;

777         return (0);
778 }

780 int
781 crsetresgid(cred_t *cr, gid_t r, gid_t e, gid_t s)
782 {
783         zone_t  *zone = crgetzone(cr);

785         ASSERT(cr->cr_ref <= 2);
```

```
787         if (BADGID(r, zone) || BADGID(e, zone) || BADGID(s, zone))
788                 return (-1);
789
790         if (r != -1)
791                 cr->cr_rgid = r;
792         if (e != -1)
793                 cr->cr_gid = e;
794         if (s != -1)
795                 cr->cr_sgid = s;
796
797         return (0);
798 }
799
800 int
801 crsetugid(cred_t *cr, uid_t uid, gid_t gid)
802 {
803         zone_t  *zone = crgetzone(cr);
804
805         ASSERT(cr->cr_ref <= 2);
806
807         if (!VALID_UID(uid, zone) || !VALID_GID(gid, zone))
808                 return (-1);
809
810         cr->cr_uid = cr->cr_ruid = cr->cr_suid = uid;
811         cr->cr_gid = cr->cr_rgid = cr->cr_sgid = gid;
812
813         return (0);
814 }
815
816 static int
817 gidcmp(const void *v1, const void *v2)
818 {
819         gid_t g1 = *(gid_t *)v1;
820         gid_t g2 = *(gid_t *)v2;
821
822         if (g1 < g2)
823                 return (-1);
824         else if (g1 > g2)
825                 return (1);
826         else
827                 return (0);
828 }
829
830 int
831 crsetgroups(cred_t *cr, int n, gid_t *grp)
832 {
833         ASSERT(cr->cr_ref <= 2);
834
835         if (n > ngroups_max || n < 0)
836                 return (-1);
837
838         if (cr->cr_grps != NULL)
839                 crgrprele(cr->cr_grps);
840
841         if (n > 0) {
842                 cr->cr_grps = kmem_alloc(CREDGRPSZ(n), KM_SLEEP);
843                 bcopy(grp, cr->cr_grps->crg_groups, n * sizeof (gid_t));
844                 cr->cr_grps->crg_ref = 1;
845                 cr->cr_grps->crg_ngroups = n;
846                 qsort(cr->cr_grps->crg_groups, n, sizeof (gid_t), gidcmp);
847         } else {
848                 cr->cr_grps = NULL;
849         }
850
851         return (0);
```



```
852 }
853
854 void
855 crsetprojid(cred_t *cr, projid_t projid)
856 {
857         ASSERT(projid >= 0 && projid <= MAXPROJID);
858         cr->cr_projid = projid;
859 }
860
861 /*
862  * This routine returns the pointer to the first element of the crg_groups
863  * array.  It can move around in an implementation defined way.
864  * Note that when we have no grouplist, we return one element but the
865  * caller should never reference it.
866  */
867 const gid_t *
868 crgetgroups(const cred_t *cr)
869 {
870         return (cr->cr_grps == NULL ? &cr->cr_gid : cr->cr_grps->crg_groups);
871 }
872
873 int
874 crgetngroups(const cred_t *cr)
875 {
876         return (cr->cr_grps == NULL ? 0 : cr->cr_grps->crg_ngroups);
877 }
878
879 void
880 cred2prcred(const cred_t *cr, prcred_t *pcrp)
881 {
882         pcrp->pr_euid = cr->cr_uid;
883         pcrp->pr_ruid = cr->cr_ruid;
884         pcrp->pr_suid = cr->cr_suid;
885         pcrp->pr_egid = cr->cr_gid;
886         pcrp->pr_rgid = cr->cr_rgid;
887         pcrp->pr_sgid = cr->cr_sgid;
888         pcrp->pr_groups[0] = 0; /* in case ngroups == 0 */
889         pcrp->pr_ngroups = cr->cr_grps == NULL ? 0 : cr->cr_grps->crg_ngroups;
890
891         if (pcrp->pr_ngroups != 0)
892                 bcopy(cr->cr_grps->crg_groups, pcrp->pr_groups,
893                     sizeof (gid_t) * pcrp->pr_ngroups);
894 }
895
896 static int
897 cred2ucaud(const cred_t *cr, auditinfo64_addr_t *ainfo, const cred_t *rcr)
898 {
899         auditinfo_addr_t        *ai;
900         au_tid_addr_t   tid;
901
902         if (secpolicy_audit_getattr(rcr, B_TRUE) != 0)
903                 return (-1);
904
905         ai = CR_AUINFO(cr);     /* caller makes sure this is non-NULL */
906         tid = ai->ai_termid;
907
908         ainfo->ai_auid = ai->ai_auid;
909         ainfo->ai_mask = ai->ai_mask;
910         ainfo->ai_asid = ai->ai_asid;
911
912         ainfo->ai_termid.at_type = tid.at_type;
913         bcopy(&tid.at_addr, &ainfo->ai_termid.at_addr, 4 * sizeof (uint_t));
914
915         ainfo->ai_termid.at_port.at_major = (uint32_t)getmajor(tid.at_port);
916         ainfo->ai_termid.at_port.at_minor = (uint32_t)getminor(tid.at_port);
```

```
        return (0);
}

void
cred2uclabel(const cred_t *cr, bslabel_t *labelp)
{
        ts_label_t      *tslp;

        if ((tslp = crgetlabel(cr)) != NULL)
                bcopy(&tslp->tsl_label, labelp, sizeof (bslabel_t));
}

/*
 * Convert a credential into a "ucred".  Allow the caller to specify
 * and aligned buffer, e.g., in an mblk, so we don't have to allocate
 * memory and copy it twice.
 *
 * This function may call cred2ucaud(), which calls CRED(). Since this
 * can be called from an interrupt thread, receiver's cred (rcr) is needed
 * to determine whether audit info should be included.
 */
struct ucred_s *
cred2ucred(const cred_t *cr, pid_t pid, void *buf, const cred_t *rcr)
{
        struct ucred_s *uc;
        uint32_t realsz = ucredminsize(cr);
        ts_label_t *tslp = is_system_labeled() ? crgetlabel(cr) : NULL;

        /* The structure isn't always completely filled in, so zero it */
        if (buf == NULL) {
                uc = kmem_zalloc(realsz, KM_SLEEP);
        } else {
                bzero(buf, realsz);
                uc = buf;
        }
        uc->uc_size = realsz;
        uc->uc_pid = pid;
        uc->uc_projid = cr->cr_projid;
        uc->uc_zoneid = crgetzoneid(cr);

        if (REMOTE_PEER_CRED(cr)) {
                /*
                 * Other than label, the rest of cred info about a
                 * remote peer isn't available. Copy the label directly
                 * after the header where we generally copy the prcred.
                 * That's why we use sizeof (struct ucred_s).  The other
                 * offset fields are initialized to 0.
                 */
                uc->uc_labeloff = tslp == NULL ? 0 : sizeof (struct ucred_s);
        } else {
                uc->uc_credoff = UCRED_CRED_OFF;
                uc->uc_privoff = UCRED_PRIV_OFF;
                uc->uc_audoff = UCRED_AUD_OFF;
                uc->uc_labeloff = tslp == NULL ? 0 : UCRED_LABEL_OFF;

                cred2prcred(cr, UCCRED(uc));
                cred2prpriv(cr, UCPRIV(uc));

                if (audoff == 0 || cred2ucaud(cr, UCAUD(uc), rcr) != 0)
                        uc->uc_audoff = 0;
        }
        if (tslp != NULL)
                bcopy(&tslp->tsl_label, UCLABEL(uc), sizeof (bslabel_t));

        return (uc);
}
```



```
/*
 * Don't allocate the non-needed group entries.  Note: this function
 * must match the code in cred2ucred; they must agree about the
 * minimal size of the ucred.
 */
uint32_t
ucredminsize(const cred_t *cr)
{
        int ndiff;

        if (cr == NULL)
                return (ucredsize);

        if (REMOTE_PEER_CRED(cr)) {
                if (is_system_labeled())
                        return (sizeof (struct ucred_s) + sizeof (bslabel_t));
                else
                        return (sizeof (struct ucred_s));
        }

        if (cr->cr_grps == NULL)
                ndiff = ngroups_max - 1;        /* Needs one for prcred_t */
        else
                ndiff = ngroups_max - cr->cr_grps->crg_ngroups;

        return (ucredsize - ndiff * sizeof (gid_t));
}

/*
 * Get the "ucred" of a process.
 */
struct ucred_s *
pgetucred(proc_t *p)
{
        cred_t *cr;
        struct ucred_s *uc;

        mutex_enter(&p->p_crlock);
        cr = p->p_cred;
        crhold(cr);
        mutex_exit(&p->p_crlock);

        uc = cred2ucred(cr, p->p_pid, NULL, CRED());
        crfree(cr);

        return (uc);
}

/*
 * If the reply status is NFSERR_EACCES, it may be because we are
 * root (no root net access).  Check the real uid, if it isn't root
 * make that the uid instead and retry the call.
 * Private interface for NFS.
 */
cred_t *
crnetadjust(cred_t *cr)
{
        if (cr->cr_uid == 0 && cr->cr_ruid != 0) {
                cr = crdup(cr);
                cr->cr_uid = cr->cr_ruid;
                return (cr);
        }
        return (NULL);
}
```

```
1050 /*
1051  * The reference count is of interest when you want to check
1052  * whether it is ok to modify the credential in place.
1053  */
1054 uint_t
1055 crgetref(const cred_t *cr)
1056 {
1057         return (cr->cr_ref);
1058 }

1060 static int
1061 get_c2audit_load(void)
1062 {
1063         static int      gotit = 0;
1064         static int      c2audit_load;

1066         if (gotit)
1067                 return (c2audit_load);
1068         c2audit_load = 1;               /* set default value once */
1069         if (mod_sysctl(SYS_CHECK_EXCLUDE, "c2audit") != 0)
1070                 c2audit_load = 0;
1071         gotit++;

1073         return (c2audit_load);
1074 }

1076 int
1077 get_audit_ucrsize(void)
1078 {
1079         return (get_c2audit_load() ? sizeof (auditinfo64_addr_t) : 0);
1080 }

1082 /*
1083  * Set zone pointer in credential to indicated value.  First adds a
1084  * hold for the new zone, then drops the hold on previous zone (if any).
1085  * This is done in this order in case the old and new zones are the
1086  * same.
1087  */
1088 void
1089 crsetzone(cred_t *cr, zone_t *zptr)
1090 {
1091         zone_t *oldzptr = cr->cr_zone;

1093         ASSERT(cr != kcred);
1094         ASSERT(cr->cr_ref <= 2);
1095         cr->cr_zone = zptr;
1096         zone_cred_hold(zptr);
1097         if (oldzptr)
1098                 zone_cred_rele(oldzptr);
1099 }

1101 /*
1102  * Create a new cred based on the supplied label
1103  */
1104 cred_t *
1105 newcred_from_bslabel(bslabel_t *blabel, uint32_t doi, int flags)
1106 {
1107         ts_label_t *lbl = labelalloc(blabel, doi, flags);
1108         cred_t *cr = NULL;

1110         if (lbl != NULL) {
1111                 if ((cr = crdup_flags(dummycr, flags)) != NULL) {
1112                         cr->cr_label = lbl;
1113                 } else {
1114                         label_rele(lbl);
1115                 }
```



```
1116         }

1118         return (cr);
1119 }

1121 /*
1122  * Derive a new cred from the existing cred, but with a different label.
1123  * To be used when a cred is being shared, but the label needs to be changed
1124  * by a caller without affecting other users
1125  */
1126 cred_t *
1127 copycred_from_tslabel(const cred_t *cr, ts_label_t *label, int flags)
1128 {
1129         cred_t *newcr = NULL;

1131         if ((newcr = crdup_flags(cr, flags)) != NULL) {
1132                 if (newcr->cr_label != NULL)
1133                         label_rele(newcr->cr_label);
1134                 label_hold(label);
1135                 newcr->cr_label = label;
1136         }

1138         return (newcr);
1139 }

1141 /*
1142  * Derive a new cred from the existing cred, but with a different label.
1143  */
1144 cred_t *
1145 copycred_from_bslabel(const cred_t *cr, bslabel_t *blabel,
1146     uint32_t doi, int flags)
1147 {
1148         ts_label_t *lbl = labelalloc(blabel, doi, flags);
1149         cred_t  *newcr = NULL;

1151         if (lbl != NULL) {
1152                 newcr = copycred_from_tslabel(cr, lbl, flags);
1153                 label_rele(lbl);
1154         }

1156         return (newcr);
1157 }

1159 /*
1160  * This function returns a pointer to the kcred-equivalent in the current zone.
1161  */
1162 cred_t *
1163 zone_kcred(void)
1164 {
1165         zone_t *zone;

1167         if ((zone = CRED()->cr_zone) != NULL)
1168                 return (zone->zone_kcred);
1169         else
1170                 return (kcred);
1171 }

1173 boolean_t
1174 valid_ephemeral_uid(zone_t *zone, uid_t id)
1175 {
1176         ephemeral_zsd_t *eph_zsd;
1177         if (id <= IDMAP_WK__MAX_UID)
1178                 return (B_TRUE);

1180         eph_zsd = get_ephemeral_zsd(zone);
1181         ASSERT(eph_zsd != NULL);
```

```
1182         membar_consumer();
1183         return (id > eph_zsd->min_uid && id <= eph_zsd->last_uid);
1184 }

1186 boolean_t
1187 valid_ephemeral_gid(zone_t *zone, gid_t id)
1188 {
1189         ephemeral_zsd_t *eph_zsd;
1190         if (id <= IDMAP_WK__MAX_GID)
1191                 return (B_TRUE);

1193         eph_zsd = get_ephemeral_zsd(zone);
1194         ASSERT(eph_zsd != NULL);
1195         membar_consumer();
1196         return (id > eph_zsd->min_gid && id <= eph_zsd->last_gid);
1197 }

1199 int
1200 eph_uid_alloc(zone_t *zone, int flags, uid_t *start, int count)
1201 {
1202         ephemeral_zsd_t *eph_zsd = get_ephemeral_zsd(zone);

1204         ASSERT(eph_zsd != NULL);

1206         mutex_enter(&eph_zsd->eph_lock);

1208         /* Test for unsigned integer wrap around */
1209         if (eph_zsd->last_uid + count < eph_zsd->last_uid) {
1210                 mutex_exit(&eph_zsd->eph_lock);
1211                 return (-1);
1212         }

1214         /* first call or idmap crashed and state corrupted */
1215         if (flags != 0)
1216                 eph_zsd->min_uid = eph_zsd->last_uid;

1218         hasephids = B_TRUE;
1219         *start = eph_zsd->last_uid + 1;
1220         atomic_add_32(&eph_zsd->last_uid, count);
1221         mutex_exit(&eph_zsd->eph_lock);
1222         return (0);
1223 }

1225 int
1226 eph_gid_alloc(zone_t *zone, int flags, gid_t *start, int count)
1227 {
1228         ephemeral_zsd_t *eph_zsd = get_ephemeral_zsd(zone);

1230         ASSERT(eph_zsd != NULL);

1232         mutex_enter(&eph_zsd->eph_lock);

1234         /* Test for unsigned integer wrap around */
1235         if (eph_zsd->last_gid + count < eph_zsd->last_gid) {
1236                 mutex_exit(&eph_zsd->eph_lock);
1237                 return (-1);
1238         }

1240         /* first call or idmap crashed and state corrupted */
1241         if (flags != 0)
1242                 eph_zsd->min_gid = eph_zsd->last_gid;

1244         hasephids = B_TRUE;
1245         *start = eph_zsd->last_gid + 1;
1246         atomic_add_32(&eph_zsd->last_gid, count);
1247         mutex_exit(&eph_zsd->eph_lock);
```



```
1248         return (0);
1249 }

1251 /*
1252  * IMPORTANT.The two functions get_ephemeral_data() and set_ephemeral_data()
1253  * are project private functions that are for use of the test system only and
1254  * are not to be used for other purposes.
1255  */

1257 void
1258 get_ephemeral_data(zone_t *zone, uid_t *min_uid, uid_t *last_uid,
1259         gid_t *min_gid, gid_t *last_gid)
1260 {
1261         ephemeral_zsd_t *eph_zsd = get_ephemeral_zsd(zone);

1263         ASSERT(eph_zsd != NULL);

1265         mutex_enter(&eph_zsd->eph_lock);

1267         *min_uid = eph_zsd->min_uid;
1268         *last_uid = eph_zsd->last_uid;
1269         *min_gid = eph_zsd->min_gid;
1270         *last_gid = eph_zsd->last_gid;

1272         mutex_exit(&eph_zsd->eph_lock);
1273 }


1276 void
1277 set_ephemeral_data(zone_t *zone, uid_t min_uid, uid_t last_uid,
1278         gid_t min_gid, gid_t last_gid)
1279 {
1280         ephemeral_zsd_t *eph_zsd = get_ephemeral_zsd(zone);

1282         ASSERT(eph_zsd != NULL);

1284         mutex_enter(&eph_zsd->eph_lock);

1286         if (min_uid != 0)
1287                 eph_zsd->min_uid = min_uid;
1288         if (last_uid != 0)
1289                 eph_zsd->last_uid = last_uid;
1290         if (min_gid != 0)
1291                 eph_zsd->min_gid = min_gid;
1292         if (last_gid != 0)
1293                 eph_zsd->last_gid = last_gid;

1295         mutex_exit(&eph_zsd->eph_lock);
1296 }

1298 /*
1299  * If the credential user SID or group SID is mapped to an ephemeral
1300  * ID, map the credential to nobody.
1301  */
1302 cred_t *
1303 crgetmapped(const cred_t *cr)
1304 {
1305         ephemeral_zsd_t *eph_zsd;
1306         /*
1307          * Someone incorrectly passed a NULL cred to a vnode operation
1308          * either on purpose or by calling CRED() in interrupt context.
1309          */
1310         if (cr == NULL)
1311                 return (NULL);

1313         if (cr->cr_ksid != NULL) {
```

```
1314                 if (cr->cr_ksid->kr_sidx[KSID_USER].ks_id > MAXUID) {
1315                         eph_zsd = get_ephemeral_zsd(crgetzone(cr));
1316                         return (eph_zsd->eph_nobody);
1317                 }
1318
1319                 if (cr->cr_ksid->kr_sidx[KSID_GROUP].ks_id > MAXUID) {
1320                         eph_zsd = get_ephemeral_zsd(crgetzone(cr));
1321                         return (eph_zsd->eph_nobody);
1322                 }
1323         }
1324
1325         return ((cred_t *)cr);
1326 }
1327
1328 /* index should be in range for a ksidindex_t */
1329 void
1330 crsetsid(cred_t *cr, ksid_t *ksp, int index)
1331 {
1332         ASSERT(cr->cr_ref <= 2);
1333         ASSERT(index >= 0 && index < KSID_COUNT);
1334         if (cr->cr_ksid == NULL && ksp == NULL)
1335                 return;
1336         cr->cr_ksid = kcrsid_setsid(cr->cr_ksid, ksp, index);
1337 }
1338
1339 void
1340 crsetsidlist(cred_t *cr, ksidlist_t *ksl)
1341 {
1342         ASSERT(cr->cr_ref <= 2);
1343         if (cr->cr_ksid == NULL && ksl == NULL)
1344                 return;
1345         cr->cr_ksid = kcrsid_setsidlist(cr->cr_ksid, ksl);
1346 }
1347
1348 ksid_t *
1349 crgetsid(const cred_t *cr, int i)
1350 {
1351         ASSERT(i >= 0 && i < KSID_COUNT);
1352         if (cr->cr_ksid != NULL && cr->cr_ksid->kr_sidx[i].ks_domain)
1353                 return ((ksid_t *)&cr->cr_ksid->kr_sidx[i]);
1354         return (NULL);
1355 }
1356
1357 ksidlist_t *
1358 crgetsidlist(const cred_t *cr)
1359 {
1360         if (cr->cr_ksid != NULL)
1361                 return (cr->cr_ksid->kr_sidlist);
1362         return (NULL);
1363 }
1364
1365 /*
1366  * Interface to set the effective and permitted privileges for
1367  * a credential; this interface does no security checks and is
1368  * intended for kernel (file)servers creating credentials with
1369  * specific privileges.
1370  */
1371 int
1372 crsetpriv(cred_t *cr, ...)
1373 {
1374         va_list ap;
1375         const char *privnm;
1376
1377         ASSERT(cr->cr_ref <= 2);
1378
1379         priv_set_PA(cr);
```



```
1380
1381         va_start(ap, cr);
1382
1383         while ((privnm = va_arg(ap, const char *)) != NULL) {
1384                 int priv = priv_getbyname(privnm, 0);
1385                 if (priv < 0)
1386                         return (-1);
1387
1388                 priv_addset(&CR_PPRIV(cr), priv);
1389                 priv_addset(&CR_EPRIV(cr), priv);
1390         }
1391         priv_adjust_PA(cr);
1392         va_end(ap);
1393         return (0);
1394 }
1395
1396 /*
1397  * Interface to effectively set the PRIV_ALL for
1398  * a credential; this interface does no security checks and is
1399  * intended for kernel (file)servers to extend the user credentials
1400  * to be ALL, like either kcred or zcred.
1401  */
1402 void
1403 crset_zone_privall(cred_t *cr)
1404 {
1405         zone_t  *zone = crgetzone(cr);
1406
1407         priv_fillset(&CR_LPRIV(cr));
1408         CR_EPRIV(cr) = CR_PPRIV(cr) = CR_IPRIV(cr) = CR_LPRIV(cr);
1409         priv_intersect(zone->zone_privset, &CR_LPRIV(cr));
1410         priv_intersect(zone->zone_privset, &CR_EPRIV(cr));
1411         priv_intersect(zone->zone_privset, &CR_IPRIV(cr));
1412         priv_intersect(zone->zone_privset, &CR_PPRIV(cr));
1413 }
1414
1415 struct credklpd *
1416 crgetcrklpd(const cred_t *cr)
1417 {
1418         return (cr->cr_klpd);
1419 }
1420
1421 void
1422 crsetcrklpd(cred_t *cr, struct credklpd *crklpd)
1423 {
1424         ASSERT(cr->cr_ref <= 2);
1425
1426         if (cr->cr_klpd != NULL)
1427                 crklpd_rele(cr->cr_klpd);
1428         cr->cr_klpd = crklpd;
1429 }
1430
1431 credgrp_t *
1432 crgrpcopyin(int n, gid_t *gidset)
1433 {
1434         credgrp_t *mem;
1435         size_t sz = CREDGRPSZ(n);
1436
1437         ASSERT(n > 0);
1438
1439         mem = kmem_alloc(sz, KM_SLEEP);
1440
1441         if (copyin(gidset, mem->crg_groups, sizeof (gid_t) * n)) {
1442                 kmem_free(mem, sz);
1443                 return (NULL);
1444         }
1445         mem->crg_ref = 1;
```

```
1446         mem->crg_ngroups = n;
1447         qsort(mem->crg_groups, n, sizeof (gid_t), gidcmp);
1448 #endif /* ! codereview */
1449         return (mem);
1450 }

1452 const gid_t *
1453 crgetggroups(const credgrp_t *grps)
1454 {
1455         return (grps->crg_groups);
1456 }

1458 void
1459 crsetcredgrp(cred_t *cr, credgrp_t *grps)
1460 {
1461         ASSERT(cr->cr_ref <= 2);

1463         if (cr->cr_grps != NULL)
1464                 crgrprele(cr->cr_grps);

1466         cr->cr_grps = grps;
1467 }

1469 void
1470 crgrprele(credgrp_t *grps)
1471 {
1472         if (atomic_add_32_nv(&grps->crg_ref, -1) == 0)
1473                 kmem_free(grps, CREDGRPSZ(grps->crg_ngroups));
1474 }

1476 static void
1477 crgrphold(credgrp_t *grps)
1478 {
1479         atomic_add_32(&grps->crg_ref, 1);
1480 }
```