```c
/*	$OpenBSD: kern_unveil.c,v 1.51 2021/09/09 13:02:36 claudio Exp $	*/

/*
 * Copyright (c) 2017-2019 Bob Beck <beck@openbsd.org>
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */

#include <sys/param.h>

#include <sys/acct.h>
#include <sys/mount.h>
#include <sys/filedesc.h>
#include <sys/proc.h>
#include <sys/namei.h>
#include <sys/pool.h>
#include <sys/vnode.h>
#include <sys/ktrace.h>
#include <sys/types.h>
#include <sys/malloc.h>
#include <sys/tree.h>
#include <sys/lock.h>

#include <sys/conf.h>
#include <sys/syscall.h>
#include <sys/syscallargs.h>
#include <sys/systm.h>

#include <sys/pledge.h>

struct unvname {
    char             *un_name;
    size_t            un_namesize;
    u_char           un_flags;
    RBT_ENTRY(unvnmae)    un_rbt;
};

RBT_HEAD(unvname_rbt, unvname);

struct unveil {
    struct vnode        *uv_vp;
    ssize_t            uv_cover;
    struct unvname_rbt    uv_names;
    struct rwlock        uv_lock;
    u_char            uv_flags;
};

/* #define DEBUG_UNVEIL */

#define UNVEIL_MAX_VNODES    128
#define UNVEIL_MAX_NAMES    128

static inline int
unvname_compare(const struct unvname *n1, const struct unvname *n2)
{
    if (n1->un_namesize == n2->un_namesize)
        return (memcmp(n1->un_name, n2->un_name, n1->un_namesize));
    else
        return (n1->un_namesize - n2->un_namesize);
}

struct unvname *
unvname_new(const char *name, size_t size, u_char flags)
{
    struct unvname *ret = malloc(sizeof(struct unvname), M_PROC, M_WAITOK);
    ret->un_name = malloc(size, M_PROC, M_WAITOK);
    memcpy(ret->un_name, name, size);
    ret->un_namesize = size;
    ret->un_flags = flags;
    return ret;
}

void
unvname_delete(struct unvname *name)
{
    free(name->un_name, M_PROC, name->un_namesize);
    free(name, M_PROC, sizeof(struct unvname));
}

RBT_PROTOTYPE(unvname_rbt, unvname, un_rbt, unvname_compare);
RBT_GENERATE(unvname_rbt, unvname, un_rbt, unvname_compare);

int
unveil_delete_names(struct unveil *uv)
{
    struct unvname *unvn, *next;
    int ret = 0;

    rw_enter_write(&uv->uv_lock);
    RBT_FOREACH_SAFE(unvn, unvname_rbt, &uv->uv_names, next) {
        RBT_REMOVE(unvname_rbt, &uv->uv_names, unvn);
        unvname_delete(unvn);
        ret++;
    }
```

```c
    rw_exit_write(&uv->uv_lock);
#ifdef DEBUG_UNVEIL
    printf("deleted %d names\n", ret);
#endif
    return ret;
}

int
unveil_add_name_unlocked(struct unveil *uv, char *name, u_char flags)
{
    struct unvname *unvn;

    unvn = unvname_new(name, strlen(name) + 1, flags);
    if (RBT_INSERT(unvname_rbt, &uv->uv_names, unvn) != NULL) {
        /* Name already present. */
        unvname_delete(unvn);
        return 0;
    }
#ifdef DEBUG_UNVEIL
    printf("added name %s underneath vnode %p\n", name, uv->uv_vp);
#endif
    return 1;
}

int
unveil_add_name(struct unveil *uv, char *name, u_char flags)
{
    int ret;

    rw_enter_write(&uv->uv_lock);
    ret = unveil_add_name_unlocked(uv, name, flags);
    rw_exit_write(&uv->uv_lock);
    return ret;
}

struct unvname *
unveil_namelookup(struct unveil *uv, char *name)
{
    struct unvname n, *ret = NULL;

    rw_enter_read(&uv->uv_lock);

#ifdef DEBUG_UNVEIL
    printf("unveil_namelookup: looking up name %s (%p) in vnode %p\n",
        name, name, uv->uv_vp);
#endif

    KASSERT(uv->uv_vp != NULL);

    n.un_name = name;
    n.un_namesize = strlen(name) + 1;
    ret = RBT_FIND(unvname_rbt, &uv->uv_names, &n);

    rw_exit_read(&uv->uv_lock);

#ifdef DEBUG_UNVEIL
    if (ret == NULL)
        printf("unveil_namelookup: no match for name %s in vnode %p\n",
            name, uv->uv_vp);
    else
        printf("unveil_namelookup: matched name %s in vnode %p\n",
            name, uv->uv_vp);
#endif
    return ret;
}

void
unveil_destroy(struct process *ps)
{
    size_t i;

    for (i = 0; ps->ps_uvpaths != NULL && i < ps->ps_uvvcount; i++) {
        struct unveil *uv = ps->ps_uvpaths + i;

        struct vnode *vp = uv->uv_vp;
        /* skip any vnodes zapped by unveil_removevnode */
        if (vp != NULL) {
            vp->v_uvcount--;
#ifdef DEBUG_UNVEIL
            printf("unveil: %s(%d): removing vnode %p uvcount %d "
                "in position %ld\n",
                ps->ps_comm, ps->ps_pid, vp, vp->v_uvcount, i);
#endif
            vrele(vp);
        }
        ps->ps_uvncount -= unveil_delete_names(uv);
        uv->uv_vp = NULL;
        uv->uv_flags = 0;
    }

    KASSERT(ps->ps_uvncount == 0);
    free(ps->ps_uvpaths, M_PROC, UNVEIL_MAX_VNODES *
        sizeof(struct unveil));
    ps->ps_uvvcount = 0;
    ps->ps_uvpaths = NULL;
}

void
unveil_copy(struct process *parent, struct process *child)
{
    size_t i;

    child->ps_uvdone = parent->ps_uvdone;
    if (parent->ps_uvvcount == 0)
        return;

    child->ps_uvpaths = mallocarray(UNVEIL_MAX_VNODES,
        sizeof(struct unveil), M_PROC, M_WAITOK|M_ZERO);

    child->ps_uvncount = 0;
    for (i = 0; parent->ps_uvpaths != NULL && i < parent->ps_uvvcount;
         i++) {
        struct unveil *from = parent->ps_uvpaths + i;
        struct unveil *to = child->ps_uvpaths + i;
        struct unvname *unvn, *next;

        to->uv_vp = from->uv_vp;
        if (to->uv_vp != NULL) {
            vref(to->uv_vp);
            to->uv_vp->v_uvcount++;
        }
        rw_init(&to->uv_lock, "unveil");
        RBT_INIT(unvname_rbt, &to->uv_names);
        rw_enter_read(&from->uv_lock);
        RBT_FOREACH_SAFE(unvn, unvname_rbt, &from->uv_names, next) {
            if (unveil_add_name_unlocked(&child->ps_uvpaths[i],
                    unvn->un_name, unvn->un_flags))
                child->ps_uvncount++;
        }
        rw_exit_read(&from->uv_lock);
        to->uv_flags = from->uv_flags;
        to->uv_cover = from->uv_cover;
    }
    child->ps_uvvcount = parent->ps_uvvcount;
}

/*
 * Walk up from vnode dp, until we find a matching unveil, or the root vnode
 * returns -1 if no unveil to be found above dp or if dp is the root vnode.
 */
ssize_t
unveil_find_cover(struct vnode *dp, struct proc *p)
{
    struct vnode *vp = NULL, *parent = NULL, *root;
    ssize_t ret = -1;
    int error;

    /* use the correct root to stop at, chrooted or not.. */
    root = p->p_fd->fd_rdir ? p->p_fd->fd_rdir : rootvnode;
    vp = dp;

    while (vp != root) {
        struct componentname cn = {
            .cn_nameiop = LOOKUP,
            .cn_flags = ISLASTCN | ISDOTDOT | RDONLY,
            .cn_proc = p,
            .cn_cred = p->p_ucred,
            .cn_pnbuf = NULL,
            .cn_nameptr = "..",
            .cn_namelen = 2,
            .cn_consume = 0
        };

        /*
         * If we are at the root of a filesystem, and we are
         * still mounted somewhere, take the .. in the above
         * filesystem.
         */
        if (vp != root && (vp->v_flag & VROOT)) {
            if (vp->v_mount == NULL)
                return -1;
            vp = vp->v_mount->mnt_vnodecovered ?
                vp->v_mount->mnt_vnodecovered : vp;
        }

        if (vget(vp, LK_EXCLUSIVE|LK_RETRY) != 0)
            return -1;
        /* Get parent vnode of vp using lookup of '..' */
        /* This returns with vp unlocked but ref'ed*/
        error = VOP_LOOKUP(vp, &parent, &cn);
        if (error) {
            if (!(cn.cn_flags & PDIRUNLOCK))
                vput(vp);
            else {
                /*
                 * This corner case should not happen because
                 * we have not set LOCKPARENT in the flags
                 */
                printf("vnode %p PDIRUNLOCK on error\n", vp);
                vrele(vp);
            }
            break;
        }

        vrele(vp);
        (void) unveil_lookup(parent, p->p_p, &ret);
        vput(parent);

        if (ret >= 0)
            break;

        if (vp == parent) {
            ret = -1;
            break;
        }
        vp = parent;
        parent = NULL;
```

```c
    }
    return ret;
}


struct unveil *
unveil_lookup(struct vnode *vp, struct process *pr, ssize_t *position)
{
    struct unveil *uv = pr->ps_uvpaths;
    ssize_t i;

    if (position != NULL)
        *position = -1;

    if (vp->v_uvcount == 0)
        return NULL;

    for (i = 0; i < pr->ps_uvvcount; i++) {
        if (vp == uv[i].uv_vp) {
            KASSERT(uv[i].uv_vp->v_uvcount > 0);
            KASSERT(uv[i].uv_vp->v_usecount > 0);
            if (position != NULL)
                *position = i;
            return &uv[i];
        }
    }
    return NULL;
}

int
unveil_parsepermissions(const char *permissions, u_char *perms)
{
    size_t i = 0;
    char c;

    *perms = 0;
    while ((c = permissions[i++]) != '\0') {
        switch (c) {
        case 'r':
            *perms |= UNVEIL_READ;
            break;
        case 'w':
            *perms |= UNVEIL_WRITE;
            break;
        case 'x':
            *perms |= UNVEIL_EXEC;
            break;
        case 'c':
            *perms |= UNVEIL_CREATE;
            break;
        default:
            return -1;
        }
    }
    return 0;
}

int
unveil_setflags(u_char *flags, u_char nflags)
{
#if 0
    if (((~(*flags)) & nflags) != 0) {
#ifdef DEBUG_UNVEIL
        printf("Flags escalation %llX -> %llX\n", *flags, nflags);
#endif
        return 1;
    }
#endif
    *flags = nflags;
    return 1;
}

struct unveil *
unveil_add_vnode(struct proc *p, struct vnode *vp)
{
    struct process *pr = p->p_p;
    struct unveil *uv = NULL;
    ssize_t i;

    KASSERT(pr->ps_uvvcount < UNVEIL_MAX_VNODES);

    uv = &pr->ps_uvpaths[pr->ps_uvvcount++];
    rw_init(&uv->uv_lock, "unveil");
    RBT_INIT(unvname_rbt, &uv->uv_names);
    uv->uv_vp = vp;
    uv->uv_flags = 0;

    /* find out what we are covered by */
    uv->uv_cover = unveil_find_cover(vp, p);

    /*
     * Find anyone covered by what we are covered by
     * and re-check what covers them (we could have
     * interposed a cover)
     */
    for (i = 0; i < pr->ps_uvvcount - 1; i++) {
        if (pr->ps_uvpaths[i].uv_cover == uv->uv_cover)
            pr->ps_uvpaths[i].uv_cover =
                unveil_find_cover(pr->ps_uvpaths[i].uv_vp, p);
    }

    return (uv);
}
```

```c
int
unveil_add(struct proc *p, struct nameidata *ndp, const char *permissions)
{
    struct process *pr = p->p_p;
    struct vnode *vp;
    struct unveil *uv;
    int directory_add;
    int ret = EINVAL;
    u_char flags;

    KASSERT(ISSET(ndp->ni_cnd.cn_flags, HASBUF)); /* must have SAVENAME */

    if (unveil_parsepermissions(permissions, &flags) == -1)
        goto done;

    if (pr->ps_uvpaths == NULL) {
        pr->ps_uvpaths = mallocarray(UNVEIL_MAX_VNODES,
            sizeof(struct unveil), M_PROC, M_WAITOK|M_ZERO);
    }

    if (pr->ps_uvvcount >= UNVEIL_MAX_VNODES ||
        pr->ps_uvncount >= UNVEIL_MAX_NAMES) {
        ret = E2BIG;
        goto done;
    }

    /* Are we a directory? or something else */
    directory_add = ndp->ni_vp != NULL && ndp->ni_vp->v_type == VDIR;

    if (directory_add)
        vp = ndp->ni_vp;
    else
        vp = ndp->ni_dvp;

    KASSERT(vp->v_type == VDIR);
    vref(vp);
    vp->v_uvcount++;
    if ((uv = unveil_lookup(vp, pr, NULL)) != NULL) {
        /*
         * We already have unveiled this directory
         * vnode
         */
        vp->v_uvcount--;
        vrele(vp);

        /*
         * If we are adding a directory which was already
         * unveiled containing only specific terminals,
         * unrestrict it.
         */
        if (directory_add) {
#ifdef DEBUG_UNVEIL
```

```
            printf("unveil: %s(%d): updating directory vnode %p"
                " to unrestricted uvcount %d\n",
                pr->ps_comm, pr->ps_pid, vp, vp->v_uvcount);
#endif
            if (!unveil_setflags(&uv->uv_flags, flags))
                ret = EPERM;
            else
                ret = 0;
            goto done;
        }

        /*
         * If we are adding a terminal that is already unveiled, just
         * replace the flags and we are done
         */
        if (!directory_add) {
            struct unvname *tname;
            if ((tname = unveil_namelookup(uv,
                ndp->ni_cnd.cn_nameptr)) != NULL) {
#ifdef DEBUG_UNVEIL
                printf("unveil: %s(%d): changing flags for %s"
                    "in vnode %p, uvcount %d\n",
                    pr->ps_comm, pr->ps_pid, tname->un_name, vp,
                    vp->v_uvcount);
#endif
                if (!unveil_setflags(&tname->un_flags, flags))
                    ret = EPERM;
                else
                    ret = 0;
                goto done;
            }
        }

    } else {
        /*
         * New unveil involving this directory vnode.
         */
        uv = unveil_add_vnode(p, vp);
    }

    /*
     * At this stage with have a unveil in uv with a vnode for a
     * directory. If the component we are adding is a directory,
     * we are done. Otherwise, we add the component name the name
     * list in uv.
     */

    if (directory_add) {
        uv->uv_flags = flags;
        ret = 0;
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d): added unrestricted directory vnode %p"
            ", uvcount %d\n",
            pr->ps_comm, pr->ps_pid, vp, vp->v_uvcount);
#endif
        goto done;
    }

    if (unveil_add_name(uv, ndp->ni_cnd.cn_nameptr, flags))
        pr->ps_uvncount++;
    ret = 0;

#ifdef DEBUG_UNVEIL
    printf("unveil: %s(%d): added name %s beneath %s vnode %p,"
        " uvcount %d\n",
        pr->ps_comm, pr->ps_pid, ndp->ni_cnd.cn_nameptr,
        uv->uv_flags ? "unrestricted" : "restricted",
        vp, vp->v_uvcount);
#endif

 done:
    return ret;
}

/*
 * XXX this will probably change.
 * XXX collapse down later once debug surely unneeded
 */
int
unveil_flagmatch(struct nameidata *ni, u_char flags)
{
    if (flags == 0) {
#ifdef DEBUG_UNVEIL
        printf("All operations forbidden for 0 flags\n");
#endif
        return 0;
    }
    if (ni->ni_unveil & UNVEIL_READ) {
        if ((flags & UNVEIL_READ) == 0) {
#ifdef DEBUG_UNVEIL
            printf("unveil lacks UNVEIL_READ\n");
#endif
            return 0;
        }
    }
    if (ni->ni_unveil & UNVEIL_WRITE) {
        if ((flags & UNVEIL_WRITE) == 0) {
#ifdef DEBUG_UNVEIL
            printf("unveil lacks UNVEIL_WRITE\n");
#endif
            return 0;
        }
    }
    if (ni->ni_unveil & UNVEIL_EXEC) {
```

```c
        if ((flags & UNVEIL_EXEC) == 0) {
#ifdef DEBUG_UNVEIL
            printf("unveil lacks UNVEIL_EXEC\n");
#endif
            return 0;
        }
    }
    if (ni->ni_unveil & UNVEIL_CREATE) {
        if ((flags & UNVEIL_CREATE) == 0) {
#ifdef DEBUG_UNVEIL
            printf("unveil lacks UNVEIL_CREATE\n");
#endif
            return 0;
        }
    }
    return 1;
}

/*
 * When traversing up towards the root figure out the proper unveil for
 * the parent directory.
 */
struct unveil *
unveil_covered(struct unveil *uv, struct vnode *dvp, struct proc *p)
{
    if (uv && uv->uv_vp == dvp) {
        /* if at the root, chrooted or not, return the current uv */
        if (dvp == (p->p_fd->fd_rdir ? p->p_fd->fd_rdir : rootvnode))
            return uv;
        if (uv->uv_cover >=0) {
            KASSERT(uv->uv_cover < p->p_p->ps_uvvcount);
            return &p->p_p->ps_uvpaths[uv->uv_cover];
        }
        return NULL;
    }
    return uv;
}


/*
 * Start a relative path lookup. Ensure we find whatever unveil covered
 * where we start from, either by having a saved current working directory
 * unveil, or by walking up and finding a cover the hard way if we are
 * doing a non AT_FDCWD relative lookup. Caller passes a NULL dp
 * if we are using AT_FDCWD.
 */
void
unveil_start_relative(struct proc *p, struct nameidata *ni, struct vnode *dp)
{
    struct process *pr = p->p_p;
    struct unveil *uv = NULL;
    ssize_t uvi;
    if (pr->ps_uvpaths == NULL)
        return;

    uv = unveil_lookup(dp, pr, NULL);
    if (uv == NULL) {
        uvi = unveil_find_cover(dp, p);
        if (uvi >= 0) {
            KASSERT(uvi < pr->ps_uvvcount);
            uv = &pr->ps_uvpaths[uvi];
        }
    }

    /*
     * Store this match for later use. Flags are checked at the end.
     */
    if (uv) {
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d): relative unveil at %p matches",
            pr->ps_comm, pr->ps_pid, uv);
#endif
        ni->ni_unveil_match = uv;
    }
}

/*
 * unveil checking - for component directories in a namei lookup.
 */
void
unveil_check_component(struct proc *p, struct nameidata *ni, struct vnode *dp)
{
    struct process *pr = p->p_p;
    struct unveil *uv = NULL;

    if (ni->ni_pledge == PLEDGE_UNVEIL || pr->ps_uvpaths == NULL)
        return;
    if (ni->ni_cnd.cn_flags & BYPASSUNVEIL)
        return;

    if (ni->ni_cnd.cn_flags & ISDOTDOT) {
        /*
         * adjust unveil match as necessary
         */
        uv = unveil_covered(ni->ni_unveil_match, dp, p);

        /* clear the match when we DOTDOT above it */
        if (ni->ni_unveil_match && ni->ni_unveil_match->uv_vp == dp)
            ni->ni_unveil_match = NULL;
    } else
        uv = unveil_lookup(dp, pr, NULL);

    if (uv != NULL) {
        /* update match */
        ni->ni_unveil_match = uv;
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d): component directory match for "
            "vnode %p\n", pr->ps_comm, pr->ps_pid, dp);
#endif
    }
}

/*
 * unveil checking - only done after namei lookup has succeeded on
 * the last component of a namei lookup.
 */
int
unveil_check_final(struct proc *p, struct nameidata *ni)
{
    struct process *pr = p->p_p;
    struct unveil *uv = NULL, *nuv;
    struct unvname *tname = NULL;

    if (ni->ni_pledge == PLEDGE_UNVEIL || pr->ps_uvpaths == NULL)
        return (0);

    if (ni->ni_cnd.cn_flags & BYPASSUNVEIL) {
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d): BYPASSUNVEIL.\n",
            pr->ps_comm, pr->ps_pid);
#endif
        return (0);
    }

    if (ni->ni_vp != NULL && ni->ni_vp->v_type == VDIR) {
        /* We are matching a directory terminal component */
        uv = unveil_lookup(ni->ni_vp, pr, NULL);
        if (uv == NULL) {
#ifdef DEBUG_UNVEIL
            printf("unveil: %s(%d) no match for vnode %p\n",
                pr->ps_comm, pr->ps_pid, ni->ni_vp);
#endif
            goto done;
        }
        if (!unveil_flagmatch(ni, uv->uv_flags)) {
#ifdef DEBUG_UNVEIL
            printf("unveil: %s(%d) flag mismatch for directory"
                " vnode %p\n",
                pr->ps_comm, pr->ps_pid, ni->ni_vp);
#endif
            pr->ps_acflag |= AUNVEIL;
            if (uv->uv_flags & UNVEIL_USERSET)
                return EACCES;
            else
                return ENOENT;
        }
        /* directory and flags match, success */
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d): matched directory \"%s\" at vnode %p\n",
            pr->ps_comm, pr->ps_pid, ni->ni_cnd.cn_nameptr,
            uv->uv_vp);
#endif
        return (0);
    }

    /* Otherwise, we are matching a non-terminal component */
    uv = unveil_lookup(ni->ni_dvp, pr, NULL);
    if (uv == NULL) {
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d) no match for directory vnode %p\n",
            pr->ps_comm, pr->ps_pid, ni->ni_dvp);
#endif
        goto done;
    }
    if ((tname = unveil_namelookup(uv, ni->ni_cnd.cn_nameptr)) == NULL) {
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d) no match for terminal '%s' in "
            "directory vnode %p\n",
            pr->ps_comm, pr->ps_pid,
            ni->ni_cnd.cn_nameptr, ni->ni_dvp);
#endif
        /* no specific name, so check unveil directory flags */
        if (!unveil_flagmatch(ni, uv->uv_flags)) {
#ifdef DEBUG_UNVEIL
            printf("unveil: %s(%d) terminal "
                "'%s' flags mismatch in directory "
                "vnode %p\n",
                pr->ps_comm, pr->ps_pid,
                ni->ni_cnd.cn_nameptr, ni->ni_dvp);
#endif
            /*
             * If dir has user set restrictions fail with
             * EACCES. Otherwise, use any covering match
             * that we found above this dir.
             */
            if (uv->uv_flags & UNVEIL_USERSET) {
                pr->ps_acflag |= AUNVEIL;
                return EACCES;
            }
            /* start backtrack from this node */
            ni->ni_unveil_match = uv;
            goto done;
        }
        /* directory flags match, success */
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d): matched \"%s\" underneath vnode %p\n",
```

```c
            pr->ps_comm, pr->ps_pid, ni->ni_cnd.cn_nameptr,
            uv->uv_vp);
#endif
        return (0);
    }
    if (!unveil_flagmatch(ni, tname->un_flags)) {
        /* do flags match for matched name */
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d) flag mismatch for terminal '%s'\n",
            pr->ps_comm, pr->ps_pid, tname->un_name);
#endif
        pr->ps_acflag |= AUNVEIL;
        return EACCES;
    }
    /* name and flags match. success */
#ifdef DEBUG_UNVEIL
    printf("unveil: %s(%d) matched terminal '%s'\n",
        pr->ps_comm, pr->ps_pid, tname->un_name);
#endif
    return (0);

done:
    /*
     * last component did not match, check previous matches if
     * access is allowed or not.
     */
    for (uv = ni->ni_unveil_match; uv != NULL; uv = nuv) {
        if (unveil_flagmatch(ni, uv->uv_flags)) {
#ifdef DEBUG_UNVEIL
            printf("unveil: %s(%d): matched \"%s\" underneath/at "
                "vnode %p\n", pr->ps_comm, pr->ps_pid,
                ni->ni_cnd.cn_nameptr, uv->uv_vp);
#endif
            return (0);
        }
        /* if node has any flags set then this is an access violation */
        if (uv->uv_flags & UNVEIL_USERSET) {
#ifdef DEBUG_UNVEIL
            printf("unveil: %s(%d) flag mismatch for vnode %p\n",
                pr->ps_comm, pr->ps_pid, uv->uv_vp);
#endif
            pr->ps_acflag |= AUNVEIL;
            return EACCES;
        }
#ifdef DEBUG_UNVEIL
        printf("unveil: %s(%d) check cover for vnode %p, uv_cover %zd\n",
            pr->ps_comm, pr->ps_pid, uv->uv_vp, uv->uv_cover);
#endif
        nuv = unveil_covered(uv, uv->uv_vp, p);
        if (nuv == uv)
            break;
    }
    pr->ps_acflag |= AUNVEIL;
    return ENOENT;
}

/*
 * Scan all active processes to see if any of them have a unveil
 * to this vnode. If so, NULL the vnode in their unveil list,
 * vrele, drop the reference, and mark their unveil list
 * as needing to have the hole shrunk the next time the process
 * uses it for lookup.
 */
void
unveil_removevnode(struct vnode *vp)
{
    struct process *pr;

    if (vp->v_uvcount == 0)
        return;

#ifdef DEBUG_UNVEIL
    printf("unveil_removevnode found vnode %p with count %d\n",
        vp, vp->v_uvcount);
#endif
    vref(vp); /* make sure it is held till we are done */

    LIST_FOREACH(pr, &allprocess, ps_list) {
        struct unveil * uv;

        if ((uv = unveil_lookup(vp, pr, NULL)) != NULL &&
            uv->uv_vp != NULL) {
            uv->uv_vp = NULL;
            uv->uv_flags = 0;
#ifdef DEBUG_UNVEIL
            printf("unveil_removevnode vnode %p now count %d\n",
                vp, vp->v_uvcount);
#endif
            if (vp->v_uvcount > 0) {
                vrele(vp);
                vp->v_uvcount--;
            } else
                panic("vp %p, v_uvcount of %d should be 0",
                    vp, vp->v_uvcount);
        }
    }
    KASSERT(vp->v_uvcount == 0);

    vrele(vp); /* release our ref */
}
```