CPE 453
Assignment 4 - /dev/Secret

Lance Boettcher (lboettch)
Christina Sardo (csardo)
11/16/15

Introduction

In this lab the group created a character-special Secret Keeper device driver for MINIX that is able to store secret information and pass a secret securely to another process.

Architecture

Our device driver is a single source file based on the hello.c driver that was already part of the MINIX3 source tree. It follows the modular MINIX 3 design principle and functions in a similar fashion to other drivers that interface with parts of the MINIX 3 system, through request messages sent to drivers. The messages contain a variety of fields used to hold the opcode (READ/WRITE) and its parameters, and our driver attempts to fulfill a request and return a reply message. Our group retained the function structure of the original hello.c driver and added six global variables to keep track of a secret's information:

- `struct device secret_device` : Represents the /dev/Secret device
- `uid_t secretOwner` : Uid of the caller that created the secret
- `char secretMessage[]` : Content of secret
- `int secretEmpty` : Boolean representing if secret is empty
- `int secretRead` : Boolean representing if the secret has been opened for reading
- `int open_counter` : State variable to count the number of times the device has been opened

Implementation

Environment

Mac OS X El Capitan and VMware Fusion 7 running MINIX 3.1.8 (Both partners)

Files Modified

- Added secretkeeper to /etc/system.conf
- Created device file /dev/Secret
- Created driver source directory in /usr/src/drivers/Secret
- Added SSGRANT to /usr/src/include/sys/ioctl.h
- The base of our secretkeeper.c file was copied from the hello.c driver that was already installed on both of our MINIX machines. We then modified three functions and added one:
    - `secret_open()` : The majority of the logic for our driver lies in the hello_open() function. This function needed to be modified in order to keep track of what user was attempting to create/read a secret and deny access to those trying to read a secret that they don't own.
    - `secret_close()` : Minimal modification was necessary to the hello_close() function. We simply added a conditional that if the secret has

content and has been read already, clear the secret by calling `clearSecret` (see below).

- `secret_transfer()` : Modified in order to transfer data both in and out of the device (the previous `hello` driver only included data transfer out of device). We also needed to edit the function to include updating our global variables secretEmpty and secretRead.
- `secret_ioctl()` : Added function that does the ioctl call SSGRANT which allows the owner of a secret to change the ownership to another user.
- `clearSecret()` : The team quickly found that we had to repeatedly write code to reset the secret. To minimize duplicate code, we created a helper function called clearSecret() that simply sets our `secretEmpty` flag, sets `secretOwner` to NO_OWNER, and initializes the message to be empty.

Code

```
#include <minix/drivers.h>
#include <minix/driver.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include "secretkeeper.h"
#include <minix/const.h>
#include <sys/ucred.h>
#include <minix/endpoint.h>
#include <sys/select.h>
#include <minix/const.h>

#include <sys/ioc_secret.h>
#include <unistd.h>

#define O_WRONLY 2
#define O_RDONLY 4
#define O_RDWR 6

#define MESSAGE_SIZE 4096
#define NO_OWNER -1

/*
 * Function prototypes for the hello driver.
 */
FORWARD _PROTOTYPE( char * secret_name,   (void) );
FORWARD _PROTOTYPE( int secret_open,      (struct driver *d, message *m) );
FORWARD _PROTOTYPE( int secret_close,     (struct driver *d, message *m) );
FORWARD _PROTOTYPE( struct device * secret_prepare, (int device) );
FORWARD _PROTOTYPE( int secret_transfer,  (int procnr, int opcode,
                                          u64_t position, iovec_t *iov,
                                          unsigned nr_req) );
FORWARD _PROTOTYPE( void secret_geometry, (struct partition *entry) );

FORWARD _PROTOTYPE( int secret_ioctl,      (struct driver *d, message *m) );

/* SEF functions and variables. */
FORWARD _PROTOTYPE( void sef_local_startup, (void) );
FORWARD _PROTOTYPE( int sef_cb_init, (int type, sef_init_info_t *info) );
FORWARD _PROTOTYPE( int sef_cb_lu_state_save, (int) );
FORWARD _PROTOTYPE( int lu_state_restore, (void) );

/* Secret Prototypes */
void clearSecret();
```

```
/* Entry points to the hello driver. */
PRIVATE struct driver secret_tab =
{
    secret_name,
    secret_open,
    secret_close,
    secret_ioctl,
    secret_prepare,
    secret_transfer,
    nop_cleanup,
    secret_geometry,
    nop_alarm,
    nop_cancel,
    nop_select,
    do_nop,
};

/** Represents the /dev/Secret device. */
PRIVATE struct device secret_device;

/* Secret global variables */
static uid_t secretOwner;                /* The UID of the owner process */
static char secretMessage[MESSAGE_SIZE]; /* The message string */
static int secretEmpty;                  /* 1 if empty. 0 if full */
static int secretRead = 1;               /* 1 if read. 0 if not read */

/** State variable to count the number of times the device has been opened. */
PRIVATE int open_counter;

PRIVATE char * secret_name(void)
{
    printf("secret_name()\n");
    return "hello";
}

PRIVATE int secret_open(d, m)
    struct driver *d;
    message *m;
{
    struct ucred callerCreds;
    int openFlags;

    printf("secret_open() called\n");

    /* Get the caller's credentials */
    if (getnucred(m->IO_ENDPT, &callerCreds)) {
        fprintf(stderr, "Open: getnucred error \n");
        exit(-1);
```

```
    }

    /* Get the flags given to open() */
    openFlags = m->COUNT;
    printf("Open Flags: %d\n", openFlags);

    /* TODO: Figure out why random flags are being used.
     * e.g. 'echo "message" > /dev/Secret' gives 578 as an open flag.
     * For now, Just consider anything but RD_ONLY as WR_ONLY
     * Commenting out check for rd_only and wr_only
     */
/*   if (openFlags != O_WRONLY && openFlags != O_RDONLY) {
        printf("Unknown or unsuppported open() flags. Got '%d'\n",
                openFlags);
        return EACCES;
    }
 */
    if (secretEmpty) {
        /* Secret empty - Owner changes to caller */

        secretOwner = callerCreds.uid;

        if (openFlags == O_RDONLY) {
            /* Empty secret opened to read. Writing not allowed */
            printf("Empty secret opened in RD_ONLY\n");

        }
        else {
            /* WR_ONLY. */
            printf("Opening an empty secret for writing \n");
        }
    }
    else {
        /* Secret Full. No writing. Owner process can read */

        if (openFlags == O_RDONLY) {
            /* Full secret opened in Read only */

            if (secretOwner == callerCreds.uid) {
                /* The owner of the secret is trying to read it - Allowed */
                printf("Full secret opened in rd_ONLY\n");
            }
            else {
                /* Someone else trying to read the secret - Denied */
                printf("%d is not the secret owner.Permission denied\n",
                       callerCreds.uid);
                return EACCES;
            }
```

```
        }
        else {
            /* WR_ONLY --> error */
            if (secretOwner != callerCreds.uid) {
                /* Someone else is trying to read the secret - Denied */
                printf("%d is not the secret owner. Permission denied \n",
                        callerCreds.uid);
                return EACCES;
            }
            else {
                /* Even though you own the secret, its full - Denied */
                printf("Cannot open a full secret for writing\n");
                return ENOSPC;
            }
        }
    }

    return OK;
}

PRIVATE int secret_close(d, m)
    struct driver *d;
    message *m;
{
    printf("secret_close()\n");

    /* If a full secret has been read - clear it */
    if (secretRead && !secretEmpty) {
        clearSecret();
    }

    printf("End of close. Owner: %d \n", secretOwner);

    return OK;
}

PRIVATE struct device * secret_prepare(dev)
    int dev;
{
    secret_device.dv_base.lo = 0;
    secret_device.dv_base.hi = 0;
    secret_device.dv_size.lo = MESSAGE_SIZE;
    secret_device.dv_size.hi = 0;
    return &secret_device;
}

PRIVATE int secret_transfer(proc_nr, opcode, position, iov, nr_req)
    int proc_nr;
```

```
    int opcode;
    u64_t position;
    iovec_t *iov;
    unsigned nr_req;
{
    int bytes, ret;

    switch (opcode)
    {
        case DEV_GATHER_S:
            /* Reading */

            printf("Hello transfer reading. Secret Message: '%s'\n",
secretMessage);

            /* Figure out the number of bytes to copy */
            bytes = strlen(secretMessage) - position.lo < iov->iov_size ?
                    strlen(secretMessage) - position.lo : iov->iov_size;

            /* If theres nothing to read, just return */
            if (bytes <= 0) {
                return OK;
            }

            /* Copy the secret to the recieving process */
            ret = sys_safecopyto(proc_nr, iov->iov_addr, 0,
                                 (vir_bytes) secretMessage,
                                  bytes, D);

            if (ret != 0) {
                printf("Transfer: Problem with safecopyto \n");
                exit(-1);
            }

            iov->iov_size -= bytes;

            /* We have read the secret - set the flag */
            secretRead = 1;

            break;

        case DEV_SCATTER_S:
            /* Writing */

            printf("Hello transfer writing: %d \n", iov->iov_size);

            /* If IO buffer size and current message bigger than max size */
            if (iov->iov_size > MESSAGE_SIZE) {
```

```
                return ENOSPC;
            }
            else { /* Message fits */

                /* Number of bytes to transfer = IO buffer size */
                bytes = iov->iov_size;

                printf("Transfer bytes: %d\n", bytes);

                /* Copy the message from the IO Buffer to our string */
                ret = sys_safecopyfrom(proc_nr, iov->iov_addr, 0,
                                    (vir_bytes) secretMessage,
                                     bytes, D);
                if (ret != 0) {
                    printf("Transfer: Problem with safecopyfrom\n");
                    exit(-1);
                }

                /* Mark flags - secret hasnt been read and is full */
                secretEmpty = 0;
                secretRead = 0;
            }
            break;

        default:
            return EINVAL;
    }

    return ret;
}

PRIVATE void secret_geometry(entry)
    struct partition *entry;
{
    printf("secret_geometry()\n");
    entry->cylinders = 0;
    entry->heads     = 0;
    entry->sectors   = 0;
}

PRIVATE int secret_ioctl(d, m)
    struct driver *d;
    message *m;
{
    struct ucred callerCreds;
    uid_t grantee;
    int res;
```

```
    printf("secret_ioctl()\n");

    /* SSGRANT is the only supported ioctl call */
    if (m->REQUEST != SSGRANT) {
        return ENOTTY;
    }

    /* Get the UID  and store in callerCreds */
    if (getnucred(m->IO_ENDPT, &callerCreds)) {
        fprintf(stderr, "Open: getnucred error \n");
        exit(-1);
    }

    /* Get the grantee */
    res = sys_safecopyfrom(m->IO_ENDPT, (vir_bytes)m->IO_GRANT, 0,
            (vir_bytes)&grantee, sizeof(grantee), D);

    if (res != 0) {
        fprintf(stderr, "Secret IOCTL: Error with safecopyfrom\n");
        exit(-1);
    }

    /* Make the grantee the owner if caller is owner*/
    if (callerCreds.uid == secretOwner) {
        secretOwner = grantee;
    }

    return OK;
}

PRIVATE int sef_cb_lu_state_save(int state) {
    /* Save the state. */
    ds_publish_u32("open_counter", open_counter, DSF_OVERWRITE);

    /* Save the secret state */
    ds_publish_str("secret_message", secretMessage, DSF_OVERWRITE);
    ds_publish_u32("secret_owner", secretOwner, DSF_OVERWRITE);
    ds_publish_u32("secret_empty", secretEmpty, DSF_OVERWRITE);
    ds_publish_u32("secret_read", secretRead, DSF_OVERWRITE);

    return OK;
}

PRIVATE int lu_state_restore() {
    /* Restore the state. */
    u32_t value;
    char oldSecretRead;
```

```
    ds_retrieve_u32("open_counter", &value);
    ds_delete_u32("open_counter");
    open_counter = (int) value;

    /* Restore the Secret state */
    ds_retrieve_str("secret_message", secretMessage, MESSAGE_SIZE);
    ds_delete_str("secret_message");

    ds_retrieve_u32("secret_owner", &value);
    ds_delete_u32("secret_owner");
    secretOwner = (int) value;

    ds_retrieve_u32("secret_empty", &value);
    ds_delete_u32("secret_empty");
    secretEmpty = (int) value;

    ds_retrieve_u32("secret_read", &value);
    ds_delete_u32("secret_read");
    secretRead = (int) value;

    return OK;
}

PRIVATE void sef_local_startup()
{
    /*
     * Register init callbacks. Use the same function for all event types
     */
    sef_setcb_init_fresh(sef_cb_init);
    sef_setcb_init_lu(sef_cb_init);
    sef_setcb_init_restart(sef_cb_init);

    /*
     * Register live update callbacks.
     */
    /* - Agree to update immediately when LU is requested in a valid state. */
    sef_setcb_lu_prepare(sef_cb_lu_prepare_always_ready);
    /* - Support live update starting from any standard state. */
    sef_setcb_lu_state_isvalid(sef_cb_lu_state_isvalid_standard);
    /* - Register a custom routine to save the state. */
    sef_setcb_lu_state_save(sef_cb_lu_state_save);

    /* Let SEF perform startup. */
    sef_startup();
}

PRIVATE int sef_cb_init(int type, sef_init_info_t *info)
{
```

```
/* Initialize the hello driver. */
    int do_announce_driver = TRUE;

    open_counter = 0;
    switch(type) {
        case SEF_INIT_FRESH:
            printf("Hey, Im initing FRESH\n");
        break;

        case SEF_INIT_LU:
            /* Restore the state. */
            lu_state_restore();
            do_announce_driver = FALSE;

            printf("Hey, I'm a new version!\n");
        break;

        case SEF_INIT_RESTART:
            printf("Hey, I've just been restarted!\n");
        break;
    }

    /* Announce we are up when necessary. */
    if (do_announce_driver) {
        driver_announce();
    }

    /* Initialization completed successfully. */
    return OK;
}

/*
 * Function to clear the secret and owner and
 * reset the secret flags
 */
void clearSecret() {
    int i;

    secretEmpty = 1;
    secretOwner = NO_OWNER;

    /* Initialize the message to be empty */
    for (i = 0; i < MESSAGE_SIZE; i++) {
        secretMessage[i] = '\0';
    }
}

PUBLIC int main(int argc, char **argv)
```

```
{
    clearSecret();

    /*
     * Perform initialization.
     */
    sef_local_startup();

    /*
     * Run the main loop.
     */
    driver_task(&secret_tab, DRIVER_STD);
    return OK;
}
```

Behavior

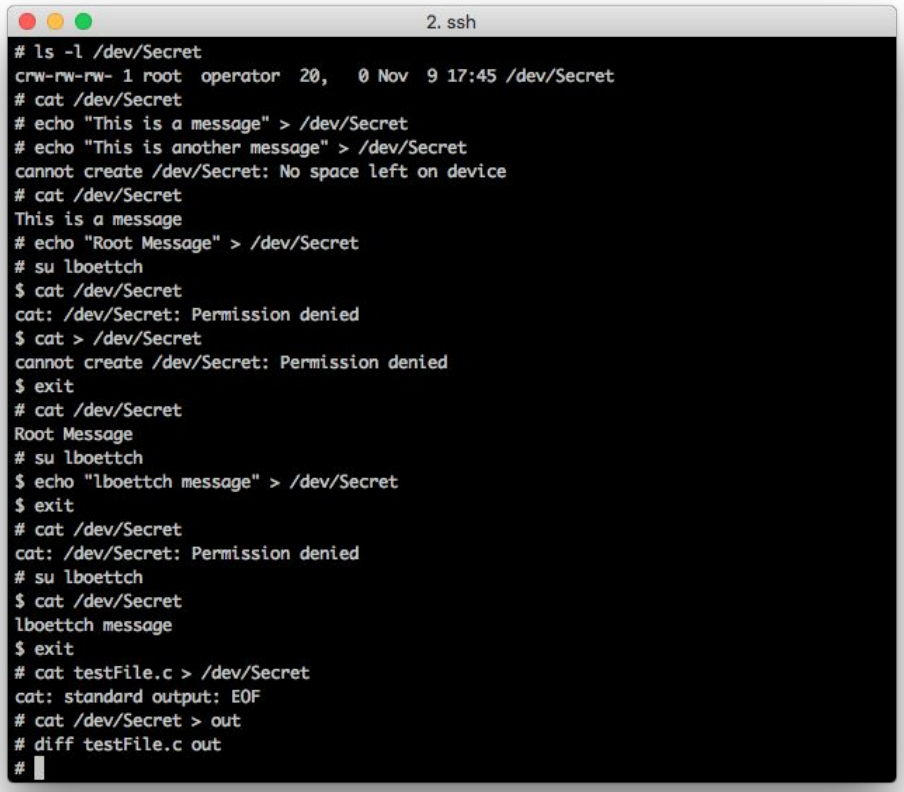

Figure 1: /dev/Secret standard functionality tests

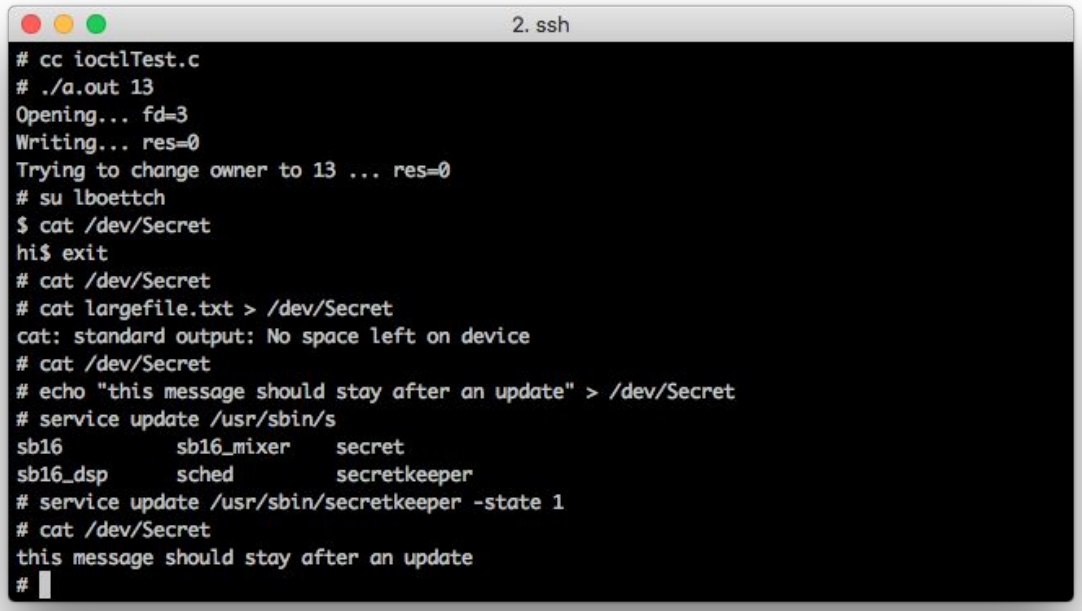

Figure 2: /dev/Secret ioctl, large file, and update tests

Problems Encountered

Problem:

The major problem that we were not able to solve involved incorrect flags being passed along with `open()` calls. We take the flags from the `COUNT` field of the message parameter as instructed, but we found that they were not always as we expected. For example, `echo "message" > /dev/Secret` would give the open flag `578`. Also, when testing the ioctl functions and calling `open()` with `O_WRONLY`, the flags were received as `O_RDWR`.

Solution:

We could not figure out if this was a problem with how we were calling `open()` or how we were receiving the flags in secretkeeper.c. The flags did always work when reading, so our solution was to treat everything but `O_RDONLY` as a `O_WRONLY`. This works for our testing but does not check for unknown or `O_RDWR` flags which should be invalid and return `EACCES`.

Lessons Learned:

Since we didn't figure out a real solution to this problem, we didn't learn any notable lessons.

Problem:
After writing the `secret_ioctl` function, we could not figure out why it was not being called with the provided ioctl driver code. We always just received a bad return code and our print statements never executed.

Solution:
We eventually figured out that we added the `secret_ioctl` function into the wrong place in the driver struct. We eventually switched `nop_ioctl` for our `secret_ioctl` function and we saw our code being executed.

Lessons Learned:
Before encountering this problem, we were blindly adding code into the existing hello driver code. We were not worrying or caring about how these functions were being called. After fixing the issue, though, we were able to see that the driver struct expects the driver functions in a specific order.

Conclusion

This lab taught us how to interact with the hardware components of our system through programming a device driver. This knowledge can be incredibly useful in the future if a user wants to customize his/her system to behave in a particular way.