Aubrey Russell – Chris Voncina OS-453 Project 4 Device Driver 2/26/2016

System Specifications:

- Windows 10 x64
- core i7 2.6ghz
- 16gb DDR3 1333Mhz
- 180gb SSD
- VirtualBox with Minix operating system version 3.1.8

Approach :

The first step we took when beginning this assignment was to read, in detail, the assignment four specification as well as the “hello driver” wiki page--we focused on this, rather than just coding immediately, to gain an initial understanding of how device drivers work with respect to Minix. Covering this material was essential for the successful completion of the project and so we spent the first day just reading over the details of the specification and figuring out how the “hello” device driver example was supposed able to interface with the operating system; in particular, we looked at core concepts and functionality like the callback functions, the system event framework, the various terminal commands we would use, the iov vector struct, “sys_safecopyto”, “sys_safecopy from”, and of course other structures we needed to utilize.

After completing our review of the spec and finishing our conceptual design for how to go about implementing the secretkeeper, we then created of the Device file in “/dev/Secret” by switching to super user and typing “mknod /dev/Secret c 20 0 ”. Then we added a secretkeeper to the system.conf and modified the ioctl.h folder to define the SSGRANT. In system.conf we simply copied the code related to the hello driver and used it for the secret keeper since both drivers require similar operating system resources. Finally, we completed the required modifications in “usr/include/sys” called secret_ioctl.h which included the ioctl.h file in it. This file, along with adding the SSGRANT definition, would allow us to use secret_ioctl.

Once we configured the system.conf file, we then essentially copied the device driver code to a new directory, replaced all instances of “hello” in functions and comments with “secret”, changed the makefile so it would create a secretkeeper instead of the hello driver, and started up the service in order to verify that we successfully setup a new driver. Once we worked out some bugs and mistakes in our code, we then produced a new convenience bash file called cleanup.sh to save time and reduce typing mistkes, which essentially downs the current runnings service, removes the old secretkeeper file, rebuilds the modified code, and then ups the new service.

With our convenience script in hand we could finally begin coding in secret.c. The first step in coding was to create the global variables that we might need including the secret, the current read position, the current size, and more. Then we had to modify the save and load state functions so that if the driver crashed it would be able to restore its state. We utilized the “ds_publish” functions that enabled us to save and load anything from the datastore. After this we moved onto the secret_open function and added error checks including the following: verifying permissions using the “getnucred” system call, checking if the secret is currently full, checking that it's not in read/write made, and other error checks like making sure the same process isn’t trying to open the device driver multiple times.

After finishing “secret_open”, we moved onto “secret_transfer” since we needed to write “secret_transfer” in order to see if we could just read and write into and from the secret. The first step here was to remove all the unnecessary code from the hello driver and create a simple function containing the switch statement that tested if the opcode is either GATHER or SCATTER. We knew that the GATHER case was sufficient since the hello driver worked, but we had to modify the code in the GATHER case so that the “sys_safecopyto” would accurately use the correct number of bytes and adjust the return value and the iov->iov_size value accordingly depending on the size of the secret. The GATHER case was working well after these modifications and we could always “cat /dev/Secret” such that it was able to output a default secret each time; we needed to disable our user permission check in secret_open for testing purposes here. Then we had to create the SCATTER case, which was relatively similar to the GATHER case except that it used sys_safecopyfrom in order to get a secret from a program and write it into the empty secret. The caveat for the SCATTER case was that we needed to handle the case where the new iov->iov_size plus current size was greater than the total buffsize. We accomplished this by subtracting the current size from the buffer size and using that to determine that number of bits that were written to the secret.

The last main function we had to write was the “secret_ioctl”, which transfers ownership to another user. This was actually an easy part because the specification listed the necessary sys_safecopyfrom code snippet in order to grab the new grantee uid. The only other part was checking to make sure to current user was the only one trying to change ownership. Testing this portion was also easy due to the provided C code in the spec that worked flawlessly to demonstrate the proper operation of our “secret_ioctl” function.

Finally, the last step in our approach was to add all the error handling for the system calls and operating system functions and then test our code using the tests listed specified in our document.

Detailed Description of Modifications:

1.) ioctl.h, in /usr/src/include/sys.
2.) “system.conf” adding the “secretkeeper”
3.) In folder “sys” added “ioc_secret.h.”
4.) <sys/ioctl.h>
5.) Modified hello.c

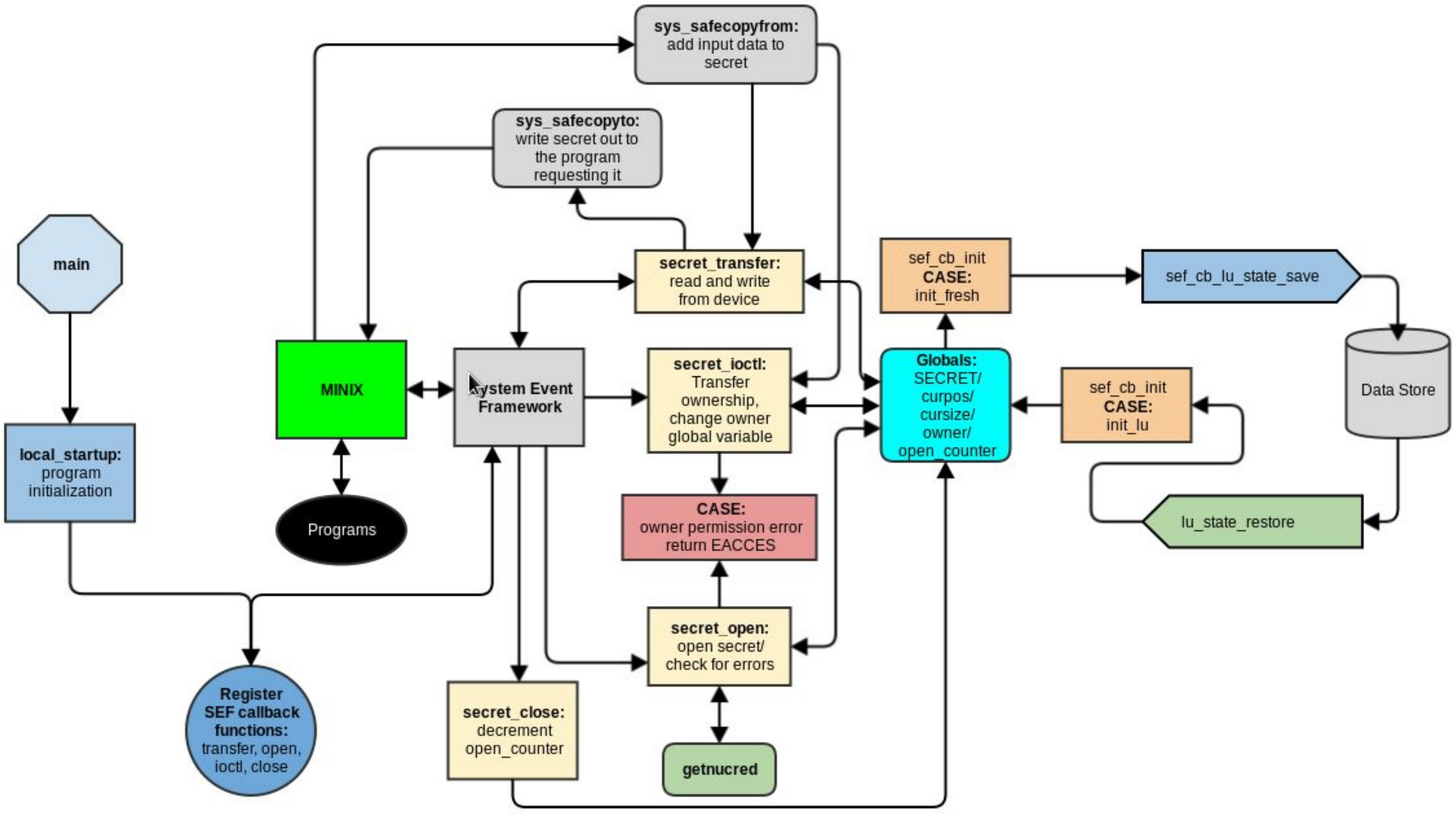
The 1st , 3rd, and 4th: all of these modifications are linked because they are used to help secret driver use the ioctl function. All of these modifications were detailed inside the specification so all we had to do was insert the includes or defines in the right file and folder in Minix. This was basically a matter of following the specifications directions. These modifications allow us to transfer ownership between any user as long as the owner of the secret is making the ioctl request. Furthermore, the 3rd modification includes the new definition in ioctl.h and defined as “SSGRANT _IOW(‘k’, 1, uid_t)”.

2nd: This modification was simple since we could copy the hello driver system.conf and then just rename it to secretkeeper. The hello driver system.conf has all the necessary functionality and mapping.

5th: We copied and changed the hello.c to secret.c. We also updated the transfer function to add the SCATTER opcode case and call the function “sys_safecopyfrom”. This used a “sys_safecopyfrom” to read the necessary data from the program and write to the secret buffer. This allows for communication between the program and device which is necessary. Third, we added in the “secret_ioctl” function to transfer ownership of the secret and deny people who do not have permission. Fourth we added to the open function to is in read or write mode, also checking if file was full or empty. Open function also checks for if user has permission to read/write this includes error handling. Finally, we modified the “lu_state_restore” function so that it grabs a memory block for each global variable the program needs for proper operation in the event of a driver failure; we don’t want to lose our potentially valuable secret just because of a driver crash.

Driver Architecture:

Architecture flowchart description:


sys_safecopyfrom: add input data to secret
sys_safecopyto: write secret out to the program requesting it
main
secret_transfer: read and write from device
sef_cb_init CASE: init_fresh
sef_cb_lu_state_save
MINIX
System Event Framework
secret_ioctl: Transfer ownership, change owner global variable
Globals: SECRET/ curpos/ cursize/ owner/ open_counter
sef_cb_init CASE: init_lu
Data Store
local_startup: program initialization
Programs
CASE: owner permission error return EACCES
lu_state_restore
secret_open: open secret/ check for errors
Register SEF callback functions: transfer, open, ioctl, close
secret_close: decrement open_counter
getnucred

The driver service begins when the “service up” command is executed; at this point the program begins its initialization sequence in main and immediately goes to the “local_startup” function. This function establishes all the call back functions with the system event framework(SEF), which is essential for programs to be able communicate with the secretkeeper. By registering the callback functions, the SEF can now trigger the appropriate functions when a program tries to do something with the device driver; programs go through the minix operating system to affect the SEF and thus trigger the functions. The local startup function is able to officially register the callback functions with the SEF by calling the “sef_startup()”. Finally, the initialization process also includes general variable initialization.

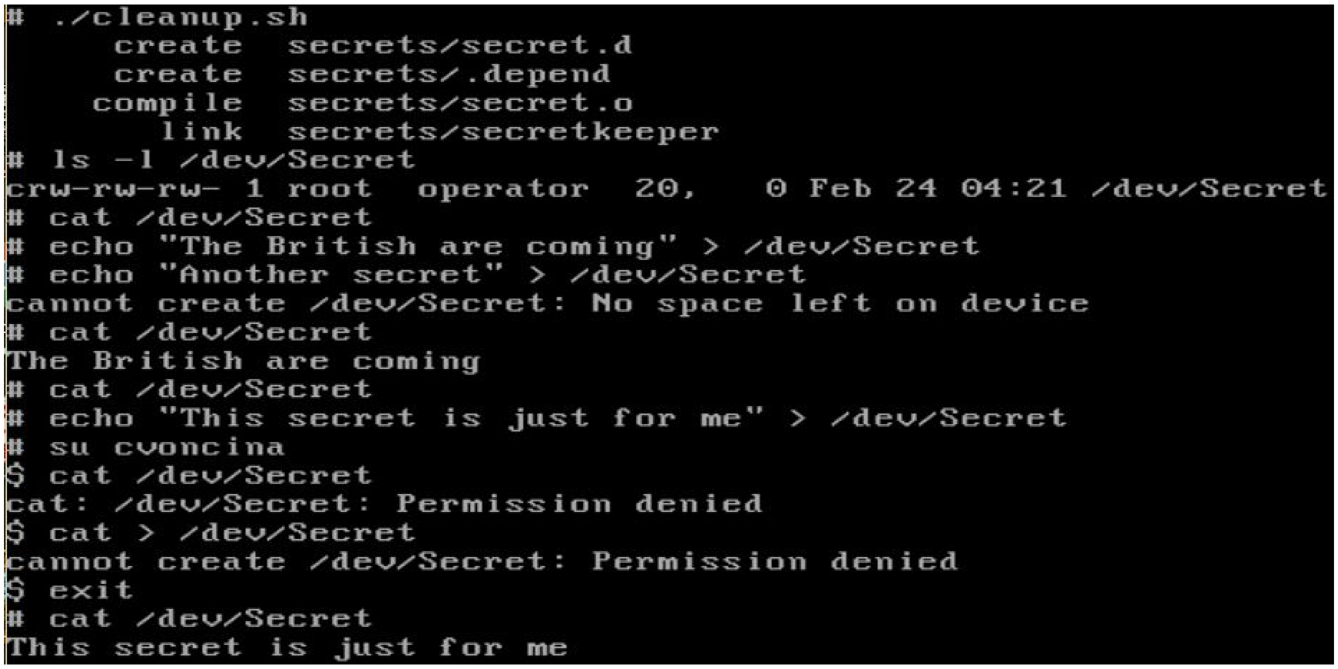
After initialization is complete, the device driver waits for the SEF to trigger one of its callback functions. The first function that is usually triggered is the “secret_open” function. This function handles error logic including the following: it verifies that user permissions are valid, checks if the device isn’t full, certifies that the secret isn’t already owned by another user or program, validates that the request is not for read/write simultaneously, and confirms a program isn’t running the open system call multiple times. “secret_open” also utilizes the “getnucred” system call in order to determine permissions. If there is an error when opening, the “secret_open” function returns the appropriate value consisting of an integer representing either an EACCES or ENOSPC error.

Once the program has opened the driver successfully, it can do several things such as reading, writing, or taking ownership. If the program tries to write to the device driver, the “secret_transfer” function is triggered; the SCATTER opcode is passed to the opcode parameter of the “secret_transfer” function. The SCATTER case will then use “sys_safecopyfrom” to read the necessary data from the program through Minix and then write it to the secret buffer. In addition, if a program tries to read from the device driver, “secret_transfer” receives the GATHER opcode and attempts to use “sys_safecopyto” to write data out to another program through Minix by using the data from the secret buffer. A program, if it owns the secret, also has the ability to change the owner permissions over to another user via the “ioctl” command. “secret_ioctl” modifies the owner global variable and allows another user to read a secret. The last step of the program is utilizing the “secret_close” in order to decrement the open_counter and then clear the secret in the case where the program is reading and the open_counter is zero.

The final thing to note about the driver architecture is how it works with the respect to the reincarnation server. Not only does the reincarnation server handle the logic for the “service” commands, but the reincarnation server also tries to restore the state of the code if it notices the driver has crashed. Thus, the flow chart demonstrates the save and restore functions by including ds_publish functions. Furthermore, if the reincarnation server detects a driver failure, it will attempt to restore the current state of the driver based on the values from the data store. As a result, a secret will not be lost due to a driver crashing, failing, or detecting a problem from one of the system calls that could occur when we do the error checking of any of the system calls.

Device Behavior:

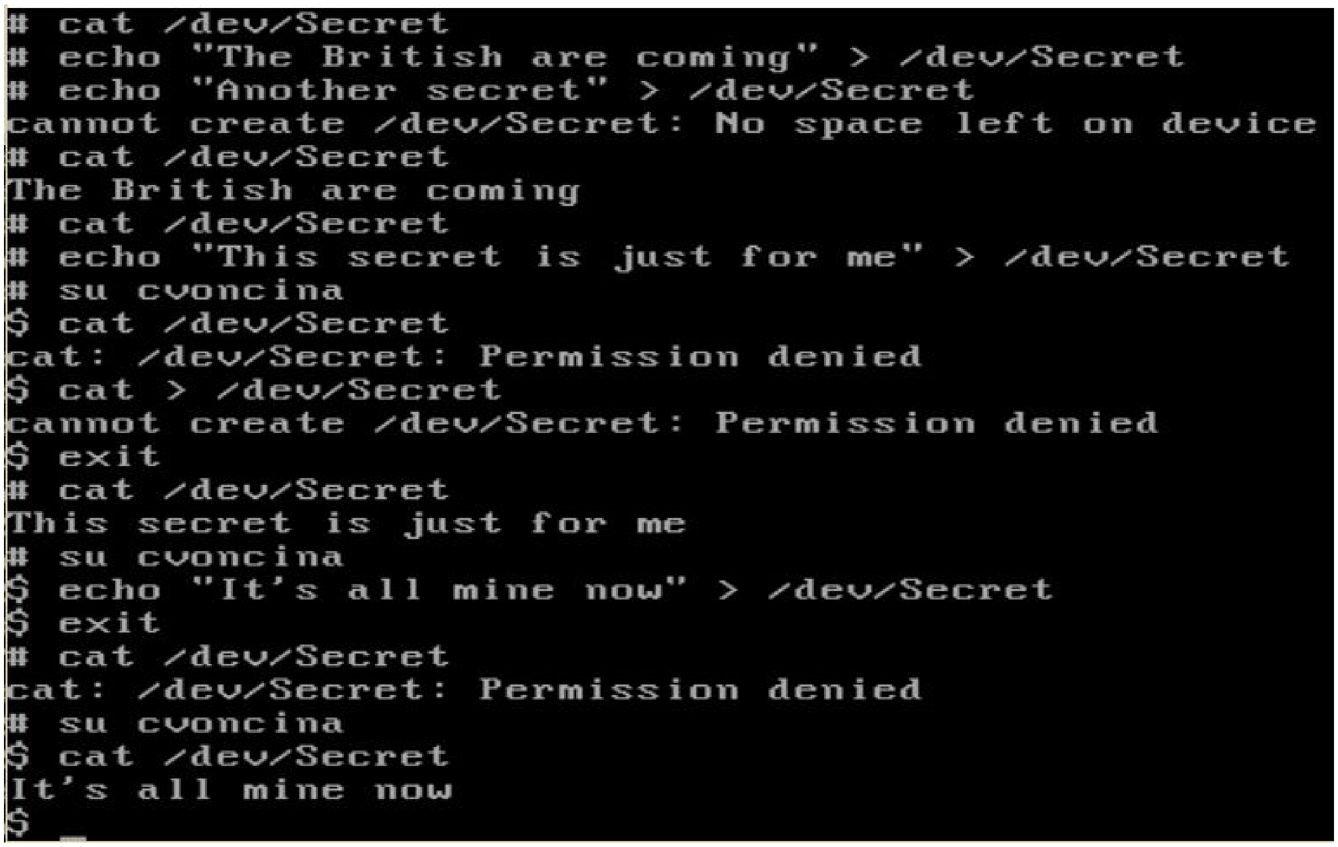
The tests below begin with a shell script called cleanup.sh which disables the current service with the “service down secretkeeper” command, deletes the current secretkeeper compilation, uses make to rebuild the secret.c file, and then finally executes a “service up” in order to launch the device. Most of these tests involve using cat, echo, and c programs in order to read, write, or transfer ownership respectively. Error tests are also included in order to verify that a user has proper permissions.



```
# ./cleanup.sh
     create  secrets/secret.d
     create  secrets/.depend
    compile  secrets/secret.o
       link  secrets/secretkeeper
# ls -l /dev/Secret
crw-rw-rw- 1 root  operator  20,   0 Feb 24 04:21 /dev/Secret
# cat /dev/Secret
# echo "The British are coming" > /dev/Secret
# echo "Another secret" > /dev/Secret
cannot create /dev/Secret: No space left on device
# cat /dev/Secret
The British are coming
# cat /dev/Secret
# echo "This secret is just for me" > /dev/Secret
# su cvoncina
$ cat /dev/Secret
cat: /dev/Secret: Permission denied
$ cat > /dev/Secret
cannot create /dev/Secret: Permission denied
$ exit
# cat /dev/Secret
This secret is just for me
```

Test # 1:

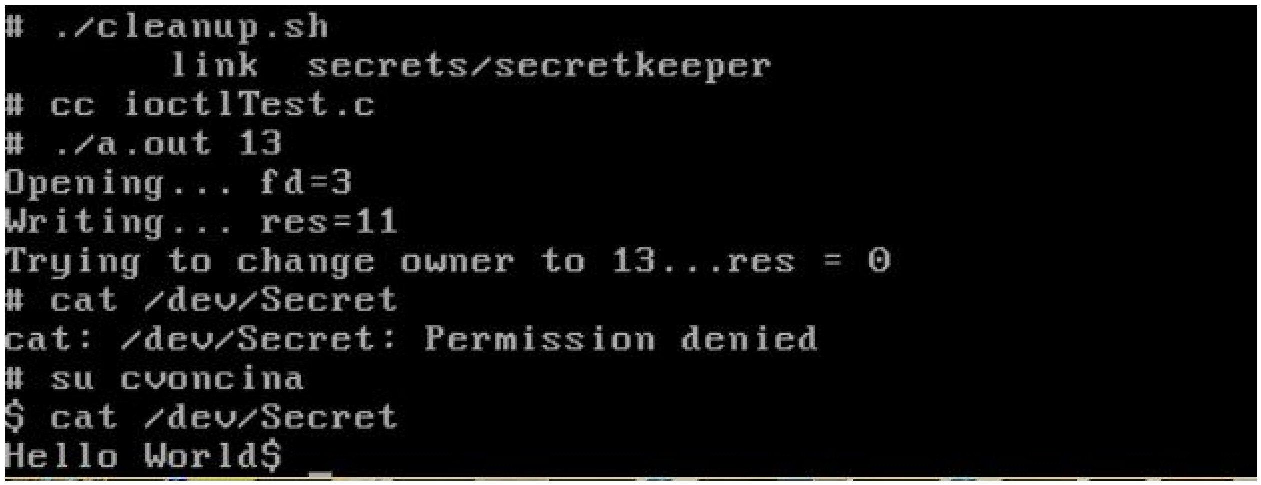
This test uses similar terminal commands compared to those provided at the bottom of the assignment four spec. First, the cleanup script runs which rebuilds secret.c and restarts the secretkeeper service. Then, the empty secret is read and two other secrets are written. However, the second secret fails to be written since it is trying to add another secret by opening a new file descriptor as shown in the above image. An error message is produced to reflect that no space is left because the file descriptor used to add the initial secret was closed before more text was written to the secret. Next, the driver displays the contents of its secret with a cat command, reflecting that “Another secret” wasn’t added improperly, thus implying that the error handling prevented incorrect behavior. Furthermore, another cat /dev/Secret is done in order to show that reading a secret and closing the . Finally, an additional secret is written, but now the user is switched to cvoncina, and a permission denied error occurs which indicates the permissions were correctly checked. Lastly, we switch back to the other user and use cat again to show the secret was not read or affected by a user that didn’t have permission.



```
# cat /dev/Secret
# echo "The British are coming" > /dev/Secret
# echo "Another secret" > /dev/Secret
cannot create /dev/Secret: No space left on device
# cat /dev/Secret
The British are coming
# cat /dev/Secret
# echo "This secret is just for me" > /dev/Secret
# su cvoncina
$ cat /dev/Secret
cat: /dev/Secret: Permission denied
$ cat > /dev/Secret
cannot create /dev/Secret: Permission denied
$ exit
# cat /dev/Secret
This secret is just for me
# su cvoncina
$ echo "It's all mine now" > /dev/Secret
$ exit
# cat /dev/Secret
cat: /dev/Secret: Permission denied
# su cvoncina
$ cat /dev/Secret
It's all mine now
$ _
```

Test #2:

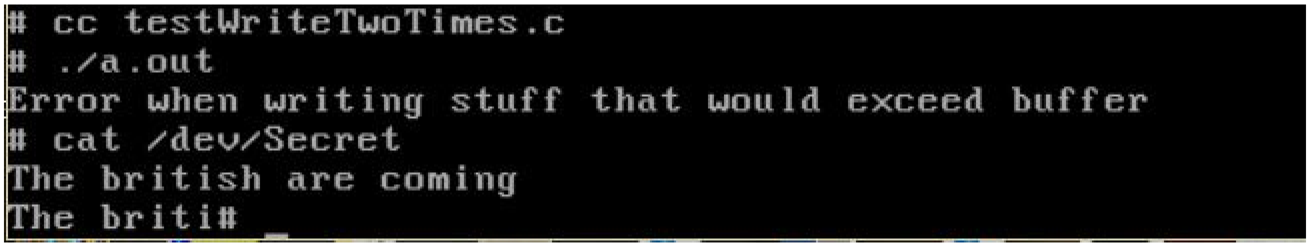
This second test is a continuation of test #1 which we took from the sample run at the end of assignment four. Test #2 starts with the command "su cvoncina". Once again we are verifying permissions by emptying a secret and then letting the cvoncina user write to the secret. When we switch back to superuser, as expected we are not allowed the read the secret, but when we switch to cvoncina we are able to read the secret. This test also includes trying to write into the buffer with cat when the user does not have permission. In addition, when the secret is emptied by the superuser, permissions are changed when cvoncina writes a new secret and effectively takes over the secret. This demonstrates permissions are valid again since super user receives a permission denied for a secret owned by cvoncina. Finally, cvoncina reads the secret and verifies that the secret wasn't affected.



```
# ./cleanup.sh
        link  secrets/secretkeeper
# cc ioctlTest.c
# ./a.out 13
Opening... fd=3
Writing... res=11
Trying to change owner to 13...res = 0
# cat /dev/Secret
cat: /dev/Secret: Permission denied
# su cvoncina
$ cat /dev/Secret
Hello World$ _
```

Test #3:

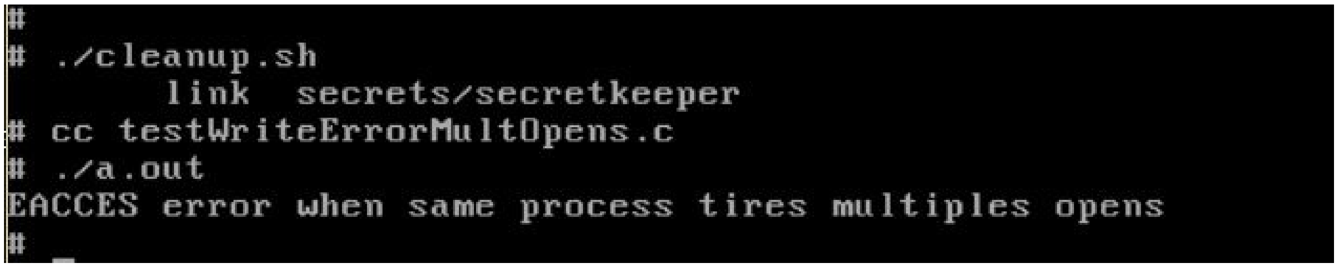
During this test we use the ioctl code given in assignment four which is used to switch permissions. We copy the input from the assignment and run the program. This program writes to secret in super user, does a cat to verify it can no longer read the secret that was just written, and switches to the cvoncina user. The cvoncina user is then able to cat the driver, which returns the proper secret in the right user context. Other simple tests were done that are not included in the screenshot here: we checked to make sure a user who didn't already own a secret could not use secret_ioctl to transfer ownership back to itself. Furthermore,



```
# cc testWriteTwoTimes.c
# ./a.out
Error when writing stuff that would exceed buffer
# cat /dev/Secret
The british are coming
The briti# _
```

Test #4:

To run this output we used one of our programs called testWriteTwoTimes.c, which opens the file and then writes two times in such a way that will exceed the buffer buffer size. As demonstrated by the output, the write system calls wrote as much as possible to the secret. Lastly, the driver produces an error ENOSPC which is due to the fact that more was written to the secret than the secret could actually store.



```
#
# ./cleanup.sh
      link  secrets/secretkeeper
# cc testWriteErrorMultOpens.c
# ./a.out
EACCES error when same process tires multiples opens
# _
```

Test #5:

This final test simply verifies that opening the device driver multiple times in the same program results in an error since it makes no sense for the same process ID to call multiple opens simultaneously without closing first before opening again.

Problems Encountered:

The first primary issue we had was simply parsing and reading through the assignment specification as well as the wiki page. These documents, along with the textbook, are quite dense and contained several new concepts such as the system event framework and reincarnation server, which made the reading challenging since there were many new things to learn. Essentially, there was a large learning curve in order to even get started on the coding in any meaningful way.

The other main issue we had was determining what the various structs, vectors, and defined constants were and understand how to take advantage of them. Furthermore, we also had some problems determining which parts of the code were unnecessary like the position.io variable and the preparation function. In addition, we faced problems typing in many terminal commands again and again such as when we accidentally redirected an echo output into our secret.c file, thus erasing the entire thing.

A related problem we had was when we made large code changes which then broke the entire program; it was then very difficult to go back through the code and deduce exactly what we did that caused the problem. In several instances, we had to recopy the original hello.c file over from scratch and then make our changes line by line, combined with testing, in order to figure out what went wrong. This took up a substantial amount of our time.

Another issue we encountered was how to use the transfer function correctly in the Secret.c file. More specifically, in the "DEV_SCATTER_S" function, we were confused on how to update the parameter that required the size of IOV. This caused an infinite loop and would not print out the stored data in "/dev/Secret " properly. We fiddled around with the parameters and were stuck on errors such as "Math Argument" and "Bad argument" after using the command "cat /dev/Secret". We had to determine how to return the correct value in secret transfer.

Error Handling was another huge part of this assignment, and determining where all the error handling checks had to go was quite difficult. After looking through the sample runs we realized we had not checked for a lot of error handling. For example, we Forgot to add the case of if same owner wants to open multiple times. The way we handled this initially was to not allow the secret to be changed as opposed to throw a ENOSPC error. Other error handling issues we ran into involved printing what the return value was and how differed from output of "cat/dev/Secret".

Solutions:

First, in order to understand all the material, we spent a massive amount of time reading documentation about minix in order to figure out what was in all of the different structs. We were able to use google and man pages in order to create a word document cheat sheet that allowed us to quickly look up or remind ourselves of how a struct or constant was defined. Second, we were able to fix the issue of mistyping terminal commands by using shell scripts.

In addition, since we broke our code many times while coding ,we created a backup each time we modified code that tested correctly.. These backups were stored in a different folder so that even if we accidentally overrode our code we would be able to recover

Furthermore, the way we solved the Math argument and Bad argument issue in the secret transfer was that we decided to use a current size variable and to increment it by the IOV->IOV_SIZE, while decrementing the IOV->IOV_SIZE to reflect the number of bits that were written. This was so that the driver knew how much was left in the BUFFER of allocated space for secret.

The way we went about solving the error handling was by creating a bunch of test cases to purposefully find the error. For instance on the case with two opens not throwing an error we wrote a class called testWriteErrorMultOpens. This would basically just call two opens inside of it and use our secret driver to find out what happens. Seeing we had issue we added more error handling in the "secret_open" function in secret.c.

Lessons Learned:

This project was a truly powerful learning experience. We learned fascinating new concepts like the system event framework and how Minix would theoretically work with actual hardware devices. We also gained an idea of the complexity of Minix and an appreciation for Minix by understanding how the operating system handles device driver crashes with the reincarnation server, how it translates virtual memory addresses for us, how it's able to use the system event framework so that any program can interact with any device driver via a few simple system calls, and much more. We hadn't truly realized the amount of logic and complexity contained within Minix alone, which is a relatively simple operating system.

We also learned the hard way about the importance of doing constant backups that are stored in multiple locations. We actually spent half a day just trying to deduce which part of the code we broke after doing some significant coding. Thus, we quickly learned that making backups after any kind of code modification was essential so that we could revert to a previous state. This is also reflective of the problems with vim and the Minix environment; had we been developing in sublime on windows, we would have simply been able to use control Z to undo all of our changes. With hindsight, we also realized we could have used github in order to save the progression of our work, especially since Minix has a git package that could be installed.. Once we started the backups however, it saved us from losing our work at least three or four times.

Next, we also learned a few smaller lessons like how to modify vim to allow for easier readability. We modified the ".vimrc" file which allowed us to change the size of tab indents and color for comments and variable types using "highlight & settab". Another important modification we did to vim was adding the command "set nu", which basically allows for permanent line numbers. We also learned how to export a virtualbox image to transfer virtual machine images to other computers.

Finally, we realized that using shell scripts to do repetitive terminal tasks was absolutely essential to get things done on time. This allowed for easy and quick compilation of our driver without having to retype the commands " service up, down, and make". Using shell scripts and test files also improved our reliability since we made fewer mistakes, some of them potentially deadly. Ultimately, however, this project gave us the opportunity to go on a sophisticated scavenger hunt through Minix and improve our understanding of operating system fundamentals.

Makefile

```
# Makefile for the Secret driver.
PROG=   secretkeeper
SRCS=  secret.c

DPADD+=        ${LIBDRIVER} ${LIBSYS}
LDADD+=        -ldriver -lsys

MAN=

BINDIR?= /usr/sbin

.include <bsd.prog.mk>
```

Cleanup.sh

```
service down secretkeeper
rm secretkeeper
make
service up /dev/secrets/secretkeeper -dev /dev/Secret
```

IOCTLTest.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/drivers.h>
#include <minix/driver.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include <minix/syslib.h>
#include "secret.h"
#include <unistd.h>
```

```
#include <sys/ucred.h>
#include <sys/ioc_secret.h>
#include <minix/const.h>

int main(int argc, char **argv)
{
   int fd, res, uid;
   char *msg = "Hello World";
   fd = open("/dev/Secret", O_WRONLY);
   printf("Opening... fd=%d\n", fd);
   res = write(fd,msg, strlen(msg));
   printf("Writing... res=%d\n", res);
   if (argc > 1 && 0 != (uid=atoi(argv[1])))
   {

          if(res = ioctl(fd, SSGRANT, &uid) )
             perror("ioctl");

         printf("Trying to change owner to %d...res = %d\n",
           uid, res);
   }
   res = close(fd);

   return 0;
}
```

Explanation: the point of this test was to see if we correctly implemented IOCTl in our secret.c file.

testWriteErrorMultOpens.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/drivers.h>
#include <minix/driver.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include <minix/syslib.h>
#include "secret.h"
#include <unistd.h>
```

```
#include <sys/ucred.h>
#include <sys/ioc_secret.h>
#include <minix/const.h>
/*this program is going to test that an open file
can continue to append to the secret and fills
up the buffer such that it truncates the
string that would exceed the buffer length
its a buffer size of 32 in this case*/
int main(int argc, char **argv)
{
   int fd1, fd2, res, uid;
   char *msg = "The british are coming\n";
   fd1 = open("/dev/Secret", O_WRONLY);
   fd2 = open("/dev/Secret", O_WRONLY);

   if (fd2 == -1)
   {
     printf("EACCES error when same process tries multiples opens\n");
   }

   res = close(fd1);
   res = close(fd2);
   return 0;
}
```

Explanation: The above class was used to test if owner tries to open file more than once and the output should throw an EACCESS error.

testWriteTwoTimes.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/drivers.h>
#include <minix/driver.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include <minix/syslib.h>
#include "secret.h"
#include <unistd.h>
#include <sys/ucred.h>
#include <sys/ioc_secret.h>
#include <minix/const.h>
/*this program is going to test that an open file
can continue to append to the secret and fills
up the buffer such that it truncates the
```

```
string that would exceed the buffer length
its a buffer size of 32 in this case*/
int main(int argc, char **argv)
{
   int fd1, fd2, res, uid;
   char *msg = "The british are coming\n";
   fd1 = open("/dev/Secret", O_WRONLY);
   res = write(fd1,msg, strlen(msg));
   res = write(fd1,msg, strlen(msg));
   if (res < 0)
   {
     printf("Error when writing stuff that would exceed buffer\n");
   }
   res = close(fd1);
   res = close(fd2);
   return 0;
}
```

Secret.c

```
/*Author: Aubrey Russell, Chris Voncina
Program: Assignment 4 driver
Date:  2/27/2016
Description: This is the MINIX driver for
the secret keeper and it is used by processes
to store a secret that can only be read by the user
who owns the secret */

#include <minix/drivers.h>
#include <minix/driver.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include <minix/syslib.h>
#include "secret.h"
#include <unistd.h>
#include <sys/ucred.h>
#include <sys/ioc_secret.h>
#include <minix/const.h>
#define SECRET_SIZE 32 /*Buffer size*/
#define O_RDWR 0x6 /*read and write constant*/
#define O_WRONLY 0x2 /*write consant*/
#define O_RDONLY 0x4 /*read constant*/
/*
 * Function prototypes for the secret driver.
 */
FORWARD _PROTOTYPE( int secret_ioctl,    (struct driver *d, message *m) );
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) );

/* 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) );

/* Entry points to the secret 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;

PRIVATE int open_counter; /*the number of fds*/
PRIVATE int curread = 0; /*current read position*/
PRIVATE char secret[SECRET_SIZE]; /*secret pointer */
PRIVATE size_t cursize = 0; /*current size of secret*/
PRIVATE size_t curpos = 0; /*current write position*/
PRIVATE struct ucred owner; /*owner of the secret*/
PRIVATE struct ucred tempowner; /*compare to owner to determine permission*/

/*function to get the name of this device, similar to hello_name*/
PRIVATE char * secret_name(void)
```

```
{
   return "secret";
}

/*transfer ownership between different user ids*/
PRIVATE int secret_ioctl(d, m)
   struct driver *d;
   message *m;
{
   int res;
   uid_t grantee;

   if (m->REQUEST != SSGRANT)
   { /*verify the SSGRANT request corresponding to secret*/
            return ENOTTY;
   }

   res = getnucred(m->IO_ENDPT, &tempowner);
   /*if the getnucred system calls fails, report it*/
   if (res < 0)
   {
     perror("getnucred failure in ioctl");
     return res;
   }

   if (owner.uid != tempowner.uid)
   { /*verify user permissions*/
            return EACCES;
   }

   res = sys_safecopyfrom(m ->IO_ENDPT, (vir_bytes)m->IO_GRANT, 0,
                    (vir_bytes) &grantee, sizeof(grantee), D);/*
                    get the grantee as specified in spec*/
   if (res < 0)
   { /*check that sys safe copy from worked*/
     perror("sys_safecopyfrom failed in ioctl");
     return res;
   }
   owner.uid = grantee; /*set the new grantee*/
   return res;
}

/*open the driver and check permissions and sizes to make sure able
to open*/
PRIVATE int secret_open(d, m)
   struct driver *d;
   message *m;
```

```
{
   int error;
   if (cursize == 0)
   { /*if the secret is empty, set a new owner of the secret*/
      error = getnucred(m->IO_ENDPT, &owner);
      if (error < 0)
      {
         perror("getnucred in secret_open failed: ");
         return error;
      }
   }
   error = getnucred(m->IO_ENDPT, &tempowner); /*check that
   the owner is theone who is granting the permission*/

   if (error < 0) /*verify that getnucred worked successfuly*/
   {
      perror("getnucred in secret_open failed: ");
      return error;
   }

   if (owner.uid != tempowner.uid)
   {
            return EACCES;        /*access error*/
   }
   /*check to make sure not read write*/
   if (m->COUNT & O_RDWR == O_RDWR)
   {
             return EACCES;
   }
   /*check if no open files and if there aren't
   and there is stuff in the buffer, then
   return ENOSPC error in anything
   but read mode--this is for making
   sure nothing can write when there are
   no extra open calls and another
   program tries to write into the buffer*/
   if (open_counter == 0 && cursize > 0)
   {
     if (m->COUNT != O_RDONLY)
     {
        return ENOSPC;
     }
   }
   /*the file has already been opened
   by the same process, prevent
   opening multiple times by same process*/
   if (open_counter > 0 && owner.pid == tempowner.pid)
```

```
    {
        return EACCES;
    }
    open_counter++;
    return OK;
}

/*called to let the driver know a file descriptor is closing*/
PRIVATE int secret_close(d, m)
    struct driver *d;
    message *m;
{
    open_counter--;
    if (curread > 0 && open_counter == 0)
    { /*checks if a read has occured and the last file descriptor
    has been closed, and then clears the secret*/
      cursize = 0;
      curpos = 0;
      curread = 0;
    }
    return OK;
}

/*not important here but used to prepare the driver base and sizes*/
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 = 0;
    secret_device.dv_size.hi = 0;
    return &secret_device;
}

/*primary logic, read and write to the secret if there is room*/
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;
    struct ucred tempowner;
   switch (opcode)
    {
      case DEV_GATHER_S:/*get secret read data from secret*/
```

```
        if (curpos > 0 && curread < cursize)
        {

  ret = sys_safecopyto(proc_nr, iov->iov_addr, 0,
                    (vir_bytes) (secret + curread),
                    curpos, D);
  if (ret < 0)
  {
    perror("GATHER sys_safecopyto failed: ");
    return ret;
  }
          curpos -= cursize; /*move the position back*/
  curread = cursize; /*make so read doesn't cover what
  it just read*/
          iov->iov_size -= iov->iov_size; /*change iov size so
  program using driver knows how many bytes were read*/
}
        else
        {
                    ret = 0;
  curpos = cursize;
        }

        break;
case DEV_SCATTER_S:
        if ((iov->iov_size + cursize) <= SECRET_SIZE)
        {/*if the size is still in the limit*/
  ret = sys_safecopyfrom(proc_nr, iov->iov_addr, 0,
                         (vir_bytes) (secret + cursize),
                          iov->iov_size, D);
  if (ret < 0)
  {  /*error handling for safecopyfrom system call*/
    perror("SCATTER sys_safecopyfrom failure: ");
  }
          cursize += iov->iov_size; /*indicate bytes written to secret*/
      curpos += iov->iov_size; /*adjust the current position
      to the end so writing can append if multiple file descript
      ors*/
          iov->iov_size -= iov->iov_size;/*tell calling program how
  many bytes were written*/
        }
else
        {/*if there are file descriptors open and the buffer still isn't
full, then write in the difference to fill the gap*/
  ret = sys_safecopyfrom(proc_nr, iov->iov_addr, 0,
      (vir_bytes) (secret + cursize),
      SECRET_SIZE - cursize, D);
```

```
        if (ret < 0)
        {
          perror("SCATTER sys_safecopyfrom failure: ");
        }
              cursize = curpos = SECRET_SIZE;
              iov->iov_size = 0;
              ret = SECRET_SIZE - cursize;
              return ENOSPC; /*return ENOSPC since couldn't fill
        buffer entirely*/
            }
            break;
      default:
        return EINVAL;
    }
    return ret;
}

/*geometry initialization*/
PRIVATE void secret_geometry(entry)
    struct partition *entry;
{
    entry->cylinders = 0;
    entry->heads     = 0;
    entry->sectors   = 0;
}

/*used to save the state of driver in case of crash or restart--
used to restore important variables*/
PRIVATE int sef_cb_lu_state_save(int state) {
/* Save the state. */
    int error;
    error = ds_publish_mem("open_counter", (void *) &open_counter,
    sizeof(int),DSF_OVERWRITE);/*save fd counter*/
    if (error < 0)
    {
      perror("ds_publish_mem open_counter error: ");
      return error;
    }

    error = ds_publish_mem("curread", (void *) &curread, sizeof(int),
    DSF_OVERWRITE); /*save current read position*/
    if (error < 0)
    {
      perror("ds_publish_mem curread error: ");
      return error;
    }
    error = ds_publish_mem("secret",(void *) secret, cursize *sizeof(char),
    DSF_OVERWRITE); /*save secret block*/
    if (error < 0)
    {
      perror("ds_publish_mem secret error: ");
      return error;
    }

    error = ds_publish_mem("owner", (void *) &owner,sizeof(struct ucred),
    DSF_OVERWRITE); /*save ownership permission struct*/
    if (error < 0)
    {
      perror("ds_publish_mem owner error: ");
      return error;
    }

    error = ds_publish_mem("cursize",(void *) &cursize, sizeof(int),
    DSF_OVERWRITE); /*save current secret size*/
    if (error < 0)
    {
      perror("ds_publish_mem cursize error: ");
      return error;
    }

    error = ds_publish_mem("curpos", (void *)&curpos, sizeof(int),
    DSF_OVERWRITE); /*save current writting position*/
    if (error < 0)
    {
      perror("ds_publish_mem curpos error: ");
      return error;
    }
    return OK;
}

/*restores all the variables listed in the above function
such that the driver may continue running normally if the driver
needs to be restarted by the reincarnation server for whatever
reason*/
PRIVATE int lu_state_restore() {
/* Restore the state. */
    size_t length;
    struct ucred credpoint;
    int ret, del, error;

    ret = ds_retrieve_mem("curpos", (char *)&curpos, &length);
    del = ds_delete_mem("curpos");
    if(ret < 0 || del < 0)
```

```
{
  error = ret < del ? ret : del;
  perror("problem with retrieve or delete curpos: ");
  return error;
}

ret = ds_retrieve_mem("open_counter",  (char *)&open_counter, &length);
del = ds_delete_mem("open_counter");
if (ret < 0 || del < 0)
{
  error = ret < del ? ret : del;
  perror("problem with retrieve or delete open_counter: ");
  return error;
}

ret = ds_retrieve_mem("curread", (char *)&curread, &length);
del = ds_delete_mem("curread");
if (ret < 0 || del < 0)
{
  error = ret < del ? ret : del;
  perror("problem with retrieve or delete curread: ");
  return error;
}

ret = ds_retrieve_mem("cursize", (char *)&cursize, &length);
del = ds_delete_mem("cursize");
if (ret < 0 || del < 0)
{
  error = ret < del ? ret : del;
  perror("problem with retireve or delete cursize: ");
  return error;
}

ret = ds_retrieve_mem("secret", secret, &length);
del = ds_delete_mem("secret");
if (ret < 0 || del < 0)
{
  error = ret < del ? ret : del;
  perror("problem with retrieve or delete secret: ");
  return error;
}

ret = ds_retrieve_mem("owner", (char *)&owner, &length);
del = ds_delete_mem("owner");
if (ret < 0 || del < 0)
{
  error = ret < del ? ret : del;
```

```
      perror("problem with retrieve or delete owner: ");
      return error;
   }

   return OK;
}

/*used to initialize the driver and callback functions*/
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();
}

/*handles new states of the driver and its corresponding logic*/
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:
         cursize = curpos = 0;
         owner.pid = owner.uid = owner.gid = -1;
      break;

      case SEF_INIT_LU:
         /* Restore the state. */
         lu_state_restore();
```

```
            do_announce_driver = FALSE;

        break;

        case SEF_INIT_RESTART:
            lu_state_restore();
        break;
    }

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

    /* Initialization completed successfully. */
    return OK;
}
/*startup the driver*/
PUBLIC int main(int argc, char **argv)
{
    /*
     * Perform initialization.
     */
    sef_local_startup();

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