Ahlquist, Ethan
Delaplaine, Victor
CPE 453, Section 01
Lab 2
February 29, 2020

Asgn 4: Minix Secret Driver

What to Turn In:

- An overall architecture of the driver: How is it expected to function?
- A detailed description of the driver implementation including:
  - Your development environment: version of the Minix kernel and type of installation(platform, native vs. bochs, qemu, or VMware)
  - A list of all files modified, and the modifications (along with why such modification was necessary),
  - The complete code of your driver.
- A description of the drivers behavior when running in the system. If possible, include typescript or screenshot of the driver in action. (The screenshot is easy if you are running simulator, not easy if you are running a native installation.)
- A section listing problems encountered, solutions attempted, results obtained, and lessons learned as previous labs.
- Other things as necessary: Remember, the meta-goal here is to convince the reader of the report that you successfully implemented a Secret Keeper device, or, failing that, to convey what you did do and that you learned something useful.

Architecture

This driver is supposed to create restricted file, in the form of a drive driver. The reason, being that the user restriction is based on user-ID rather than the group-level. This makes it so that the number of people that are owners of the file are 0 or 1.

This makes sense why we label this device drive (secretKeeper), because it restricts access of the file to only the user who last wrote to the empty device, and they alone are able to read from it.

For example:

```
service up /usr/src/drivers/secrets/secret -dev /dev/secret
```

This should start our driver as a service connected to `/dev/secret`

```
echo "Hello Person" > /dev/secret
```

This gives the user who ran this command exclusive access to read from `/dev/secret`

```
su victor
cat /dev/secret
```

This should not work, because you are no longer the user that wrote to `/dev/secret`

```
su root
cat /dev/secret
Hello Person
```

This time it worked because you are the right user

Implementation

Development Environment

We had only one Minix machine able to connect to the internet. So we could only work from one computer or through ssh.

On both of our computers used the `Virtual Box` virtualizer.

Files Modified:

/usr/src/drivers/hello/hello.c

- This code was used as the basis for our secret driver

/etc/system.conf

- Tells the kernel of a new valid service type (secret)
- Tells the OS knows what the service should have access to, and what permissions it needs

/usr/src/includes/sys/ioctl.h

- Changing ioctl includes allows it to have acess to the secret dirver that are needed to use secret driver with other programs.

/usr/src/includes/sys/ioc_secret.h

- Gives the secret driver a unique key ID for programs, being 'K'

Secret.c Source Code:

```
#include <minix/drivers.h>
#include <minix/driver.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include <sys/ioctl.h>
#include <sys/ucred.h>
#include <unistd.h>
#include <minix/com.h>
#include <minix/const.h>
#include <string.h>


#define NOT_OWNED -1
#define SECRET_SIZE 6000
#define TRUE 1
#define FALSE 0
#define O_RDONLY 04
#define O_WRONLY 02
#define O_RDWR 06


char secret[SECRET_SIZE] = {'\0'};
int is_reading=FALSE;

PRIVATE uid_t owner;

PRIVATE int open_counter;
PRIVATE struct device s_device;

/* helper function*/
FORWARD _PROTOTYPE( int isFlag, (int flag ));
FORWARD _PROTOTYPE( void clear_secret, (char *message, int size) );
FORWARD _PROTOTYPE( int is_not_owned, (uid_t owner) ) ;
FORWARD _PROTOTYPE( void rst_owner, (uid_t * owner) );
FORWARD _PROTOTYPE( int is_cred_same, (uid_t own, uid_t curr) );


FORWARD _PROTOTYPE( char * s_name, (void) );
FORWARD _PROTOTYPE( int s_do_ioctl, (struct driver *dp, message *
    m_ptr) );
FORWARD _PROTOTYPE( int s_do_open, (struct driver *dp, message *m_ptr
    ) );
```

```
FORWARD _PROTOTYPE( int s_do_close, (struct driver *dp, message *
    m_ptr) );
FORWARD _PROTOTYPE( struct device * s_prepare, (int dev) );
FORWARD _PROTOTYPE( int s_transfer, (
  int proc_nr,
  int opcode,
  u64_t position,
  iovec_t *iov,
  unsigned nr_req)
  );

FORWARD _PROTOTYPE( int sef_cb_lu_state_save, (int state) );
FORWARD _PROTOTYPE( int sef_cb_init_fresh, (int type, sef_init_info_t
     *info) );
FORWARD _PROTOTYPE( int lu_state_restore, (void) );
FORWARD _PROTOTYPE( void sef_local_startup, (void) );
```

```
/* entry point for this driver*/
PRIVATE struct driver s_dtab = {
  s_name,     /* current device's name */
  s_do_open,     /* open request, init device */
  s_do_close,    /* release device */
  s_do_ioctl,    /* ioctl */
  s_prepare,     /* prepare for I/O */
  s_transfer,      /* do I/O read or write*/
  nop_cleanup,     /* nothing to clean up */
  NULL,          /* no geometry*/
  nop_alarm,     /* no alarm */
  nop_cancel,    /* nop cancel */
  nop_select,      /* not really sure */
  NULL,
  do_nop         /* not sure for dr_hw_int*/
};


PUBLIC int main(){
  printf("hello world\n");
  rst_owner(&owner);
  /* SEF local startup */
  sef_local_startup();


  /* Call the generic recieve loop */
  driver_task(&s_dtab, DRIVER_STD);
```

```
  return OK;
}


PRIVATE char *s_name(){
  return "Secret";
}

PRIVATE int s_do_ioctl(dp, m)
  struct driver *dp;
  message *m;
{
  int res;
  uid_t grantee;
  if( m->REQUEST== SSGRANT ){
    res=sys_safecopyfrom(
      m->IO_ENDPT,
      (vir_bytes)m->IO_GRANT,
      0,
      (vir_bytes)&grantee,
      sizeof(grantee),
      D);

    owner = grantee;
    return OK;
  }
  printf("command for ioctl doesnt exist");
  return ENOTTY;

}


PRIVATE int s_do_open(struct driver *dp, message *m_ptr){

  struct ucred curr;
  int flag, is_empty=FALSE;

  flag=(m_ptr->COUNT & (R_BIT|W_BIT));


  /* Step 1 - check to see if the flag is only R or W*/
  if(!isFlag(flag)){
    printf("open() was opened not in read or write mode\n");
    return EACCES;
  }
  /*Step 2 - get the processes endpoint(uid, gid, pid) */
  if(getnucred(m_ptr->IO_ENDPT, &curr)){
```

```
    printf("getnucred failed\n");
    return errno;
  }

  printf("owner uid: %d\n", owner);
  printf("curr uid: %d\n", curr.uid);
  /*Step 3 - first check to see if owner is enmpty;*/
  if(is_not_owned(owner)){
    is_empty=TRUE;
    owner=curr.uid;
  /*Step 4 - this means that owner isnt full*/
  }else if(!is_cred_same(owner, curr.uid)){
    return EACCES;
  }
  /* step 5 - see if there request is R/W
    Assumption only if we make it here the
    Owner is empty or the curr is owner
  */
  switch(flag){
    case O_RDONLY:
      /*ask about this*/
      is_reading=TRUE;
      /* only owner can read the message(assumption ony)*/
      if (is_empty){
        printf("Cant read it because this file doesnt contain
            anything\n");
      }else{
        /* if the secret is full*/
        printf("Opened in read mode where the secret is full\n");
      }
      break;
    case O_WRONLY:
      /*only if empty you can write*/
      if(is_empty){
        /*then one can write*/
        printf("Empty secret in WRITE only mode\n");
      }else{
        /* cant write because its full*/
        printf("Secret is full cannot write anything\n");
        return ENOSPC;
      }
  }
  open_counter++;
  return OK;
}


PRIVATE int s_do_close(struct driver *dp, message *m_ptr){
```

```
  if(open_counter < 1){
    panic("closed too often\n");
  }
  /*if this is the last open file descriptor then entry the message*/
  if(open_counter==1 && is_reading){
    printf("Closed last file descriptor, clearing the secret\n");
    clear_secret(secret, SECRET_SIZE);
    is_reading=FALSE;
    owner=NOT_OWNED;
  }

  printf("open fds: %d\n", --open_counter);
  return OK;
}


PRIVATE struct device *s_prepare(int dev){
  s_device.dv_base.lo=0;
  s_device.dv_base.hi=0;
  s_device.dv_size.lo=strlen(secret);
  s_device.dv_size.hi=0;
  return &s_device;
}


/* Assumption: that file is open(that is secret is empty or owner
   reading)*/
PRIVATE int s_transfer(
  int proc_nr,
  int opcode,
  u64_t position,
  iovec_t *iov,
  unsigned nr_req
){
  int ret, bytes;
  struct ucred curr;



  /* step 3 - is it a read or write request? */
  switch (opcode)
  {
  /* READ*/
  case DEV_GATHER_S:
      /*position - where we are in the /dev/Secret file*/

      bytes = strlen(secret) - position.lo < iov->iov_size ?
```

```
            strlen(secret) - position.lo : iov->iov_size;

        /* step 2 - if less than 0 bytes give a EOF */
        if(bytes <=0 ) {
          return OK;
        }


        /* step 3.1 - send the data to user at iov_add */
        ret=sys_safecopyto(
          proc_nr,
          (vir_bytes)iov->iov_addr,
          (vir_bytes)0,
          (vir_bytes)(secret+position.lo),
          bytes,
          D);
      /* step 3.2 - if didnt didnt read all of it*/
      iov->iov_size-=bytes;
      /* step 3.3 - shift, then check to see if read all of it */
      break;
    case DEV_SCATTER_S: /* WRITE*/
      /* step 3.1 - write data from iov_add to secret*/
      /* step 1 - if the size asked for is greater than our size*/
      /*position - where we are in the /dev/Secret file*/
      bytes = SECRET_SIZE - position.lo < iov->iov_size ?
        SECRET_SIZE - position.lo : iov->iov_size;

      if(iov->iov_size > SECRET_SIZE){
        return ENOSPC;
      }
        ret=sys_safecopyfrom(
             proc_nr,
             iov->iov_addr,
             (vir_bytes)0,
             (vir_bytes)(secret+position.lo),
             bytes,
             D);
      iov->iov_size-=bytes;
      break;
    default:
      printf("not read/write\n");
      return EINVAL;
    }
    return ret;
}


PRIVATE void sef_local_startup(){
```

```
  /* Register init callbacks to be called by RS*/
  sef_setcb_init_fresh(sef_cb_init_fresh);
  sef_setcb_init_lu(sef_cb_init_fresh);
  sef_setcb_init_restart(sef_cb_init_fresh);

  /* Register live callback (for diff events)*/
  sef_setcb_lu_state_save(sef_cb_lu_state_save);

  sef_startup();

}

PRIVATE int sef_cb_lu_state_save(int state){

  /* save the state of open_counter*/
  if(ds_publish_u32("open_counter",open_counter, DSF_OVERWRITE)){
    return EAGAIN;
  }
  return OK;
}


PRIVATE int sef_cb_init_fresh(int type, sef_init_info_t *info){
  /* Initalize the secret driver. */
  int do_announce_driver = TRUE;

  open_counter = 0;
  switch(type){
    /* init fresh */
    case SEF_INIT_FRESH:
      printf("fresh start\n");
      break;
    /* init after live update */
    case SEF_INIT_LU:
      /* restores the state */
      if(lu_state_restore()==ESRCH){
        return ESRCH;
      }
      do_announce_driver=FALSE;
      printf("Hey, I'm a new version!\n");
      printf("open fd's restored at %d= !\n", open_counter);
      break;
    case SEF_INIT_RESTART:
      printf("Hey, I've just been restarted\n");
      break;

  }
  if (do_announce_driver){
```

```
    driver_announce();
  }

  /* init went sucessfully*/
  return OK;
}

PRIVATE int lu_state_restore(){
  /* restore the state */
  u32_t value;
  /* Retrieve an unsigned int */
  if(ds_retrieve_u32("open_counter", &value) == ESRCH){
    return ESRCH;
  }
  /* delete the variable stored in ds*/
  if(ds_delete_u32("open_counter")==ESRCH){
    return ESRCH;
  }
  open_counter = (int)value;

  return OK;
}


PRIVATE int isFlag(int flags){
  switch (flags)
  {
  case O_WRONLY:
    return TRUE;
    break;
  case O_RDONLY:
    return TRUE;
    break;
  default:
    return FALSE;
    break;
  }
  return FALSE;
}

PRIVATE void clear_secret(char *message, int size){
  int i;
  for(i=0; i < size; i++){
    message[i] = '\0';
  }
}
PRIVATE void rst_owner(uid_t *owner){
  *owner=NOT_OWNED;
```

```
}

PRIVATE int is_not_owned(uid_t owner){
     return owner==NOT_OWNED;
}
PRIVATE int is_cred_same(uid_t owner, uid_t curr){
     return owner==curr;
}
```

Makefile:

```
PROG= secret
SRCS= secret.c

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

MAN=

BINDIR?= /usr/sbin

.include <bsd.prog.mk>

up: secret
  service up /usr/src/drivers/secrets/secret -dev /dev/secret

down: secret
  service down secret
```

Behavior

In order to display functionality under different test cases, we placed print statements in out code, to show what was being done behind the scenes. These are some such outputs:

Nico's Sample Run:

- Provided by professor
- Tests the driver's ability to tell user possession of driver.
- Tests driver's ability to tell wither it is full or empty, and respond appropriately.

```
bash-4.1# ls -l /dev/secret
crw-rw-rw- 1 root  operator  20,   0 Feb 28 13:47 /dev/secret
bash-4.1# cat /dev/secret
bash-4.1# echo "The British are coming" > /dev/secret
bash-4.1# echo "Another secret" > /dev/secret
bash: /dev/secret: No space left on device
bash-4.1# cat /dev/secret
The British are coming
bash-4.1# cat /dev/secret
bash-4.1# echo "This secret is just for me" > /dev/secret
bash-4.1# su victor
$ cat /dev/secret
cat: /dev/secret: Permission denied
$ cat > /dev/secret
cannot create /dev/secret: Permission denied
$ exit
bash-4.1# cat /dev/secret
This secret is just for me
bash-4.1# su victor
$ echo "It's all mine now" > /dev/secret
$ exit
bash-4.1# cat /dev/secret
cat: /dev/secret: Permission denied
bash-4.1# su victor
$ cat /dev/secret
It's all mine now
$ exit
bash-4.1# ls -l test1.c
-rw-r--r-- 1 root  operator  892 Feb 27 04:01 test1.c
bash-4.1# cat test1.c > /dev/secret
bash-4.1# cat /dev/secret > a
```

Figure 1: Nico's Sample Run

BigMessage:

- Provided by professor
- Tests edge case where the driver is given a message larger than it can store

```
bash-4.1# ls -l test1.c
-rw-r--r-- 1 root  operator  892 Feb 27 04:01 test1.c
bash-4.1# cat test1.c > /dev/secret
bash-4.1# cat /dev/secret > a
bash-4.1# diff a test1.c
bash-4.1# ls -l BigFile
-rw-r--r-- 1 root  operator  2210443 Feb 28 13:40 BigFile
bash-4.1# cat BigFile > /dev/secret
cat: standard output: No space left on device
bash-4.1# cat /dev/secret > out
bash-4.1# ls -l out
-rw-r--r-- 1 root  operator  0 Feb 28 15:32 out
bash-4.1#
```

Figure 2: Service Update Test Output

Owner Behavior:

- One of our tests
- Again, tests the driver's correct response to user IDs
- Tests using print-statements inside the driver source code
- Code directly changes the uid of the process, to fool the driver
- In the end, changing the uid will allow the driver to be read from

```
bash-4.1# ls
.depend   Makefile   secret   secret.c   secret.d   secret.o   tests
bash-4.1# cd tests/
bash-4.1# ls
BigFile   a   a.out   out   test1.c   test2.c   test2.c~   test3.sh
bash-4.1# ./a.out 13
opening... fd1=3
writing... res=13
Trying to change owner to 13... res=0
bash-4.1# su victor
$ cat /dev/secret
Hello, World
$ exit
bash-4.1#
```

Figure 3: Service Update Test Output

Refresh Secret:

- One of our tests
- Tests `service refresh secret`
- Tests the ability of the driver to Refresh
- The `"Hello World"` shows that driver `open()` protocol is being run
- The `"Hey, I've just been restarted"` shows that the reincarnation code is being run

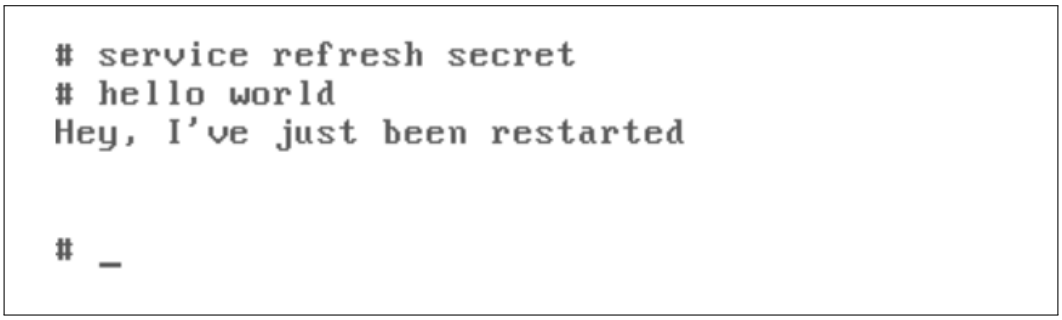

Figure 4: Service Update Test Output

NOTE: this image looks different, because this was captured via the Virtual Box screen Capture, whereas the other Screenshots are taken from an ssh terminal.

Problems Encountered

- Sharing Code Bases, since Minix is harder to work with than a normal machine.

  We also had the addeed difficulty that one of our machine's Minix could not connect to the internet.
- understanding `service(8)`
- Understanding how Minix drivers worked
- Broken Makefile
- Debugging issues

Solutions Developed

- Sharing code required us to ssh into a VirtualBox Minix machine. This was complicated with the fact that ssh-ing directly into Minix from my Linux caused problems.

  To solve this, we found that first ssh-ing into the partner's Linux, and from there ssh-ing into Minix worked the best.

  Lastly, for sharing all the code, it was fortunate that the code could be pushed to github from Minix.

- `service(8)` took very long to understand. The man page was not very specific about what paths should be used where, and what the commands where supposed to do.

  We spent time reading man pages, and trying `service(8)` in different ways with the hello driver.

  Eventually, by bruit force we found out how to use it.

  Later we saw how it was used in the provided test script, realizing that we could have learned from the script.

- Understanding how to start was the biggest issue, this project had not much code to write; however, required a lot of reading to understand what was going on.

  To deal with this, we would meet together, and ask questions we wanted the answer to, and look them up in the Textbook.

  This had us learning more than was needed to complete the assignment, but it gave us a good idea on how some of the more complex drivers in Minix work.

- In one of our Makefiles, we had incorrect spacing, and that was causing us issues.

  We only found out the cause of the problem by posting a question to Piazza

- Lastly, with debugging we learned that often placing print-statements in your code, can display a lot of important debugging information. For example, to find out which

functions we were using at different times, we placed print statements displaying this information

Simply placing print-statements will likely not ruin the functionality of your code, but display important information.

Results

In the end we created a working driver. The did what was required by the specifications. The driver exhibited this in our macro tests, as well as a low-level debugging of specific functions. Our driver worked with more than just the basic functionality but also the reincarnation and resetting. I believe this was a productive assignment teaching us the fundamentals of Minix drivers and drivers as a whole.

Lessons Learned

- In sharing code, we found that sometimes sharing code isn't actually the most beneficial use of time. Towards the end of the project, we were just debugging, so instead of working separately, we used google-hangouts to share our screen. This made it so we were both looking at the same code; therefore, thinking about the same problems.
- Often times with school projects if you have a question, the answer is provided somewhere in the instructions. When we were trying to figure out how `service(8)` worked, we could have looked at the provided test code to see how to use it instead of brute forcing it.
- When working with the kernel, it might not be a bad idea to use print-statements to debug. In fact, it might be the only way to debug on the Minix system.
- Reading documentation can be a pain. Sometimes you might fall into the trap of reading too much and wasting time. You can also read too little and waste time as well. In this project, we found that it was better to over-read documentation, especially when dealing with a completely new framework. The time saved by coding with intelligence out weigh the coding quickly.

functions

s_do_open

check if read Or write only if(not read or Write) return No access get endpoint == process idd

if no owner set current to owner is_empty = true

Else if(credibility matches) //usr id not pid return error

If here -¿ (No message —— current reader is owner)

/*Case Statement is for debugging */ set is_reading because you can only change ownership once the message has been completely read and the last file descriptor has been closed

s_do_close()

decrement s the open count - meaning num file descriptors open

s_do_transfer()

bytes number want to send or receive

If bytes ¡ 0 then return OK //EOF

If bytes ¿ secret Size

Return error

ssafecopy()

(SOURCE) iov_addr = the address of the buffer we are going to put stuff into

(DESTINATION) secret+bytes == starting address for where we are going to read from

D === No idea

restore

ds = data storage

File descriptor open service update

Print out the open_count, because we want to see if driver is publishing and receiving correctly from ds, when reincarnation/refresh occurs.

rest are helpers

s_do_ioctl