Zarek Lazowski

CPE 453

Assignment 4 - /dev/Secret

Driver Architecture:

This driver is expected to act more or less as file with a lock. The “file”, henceforth known as secret, can be set to any size by setting the environment variable SECRET_SIZE, but the default is 8192 bytes long.

It expects an open request, in which it will determine whether or not to open. The driver opens for only read or only write access, other types of access result in EACCES being returned. Write file descriptors will be granted to requestors when there is no current owner. Otherwise ENOSPC is returned as it is assumed that if there is an owner, the device has been written to. Read file descriptors will be granted to requestors with UIDs that match the current owner stored in the device. Otherwise EACCES is returned.

The owner of the secret can give ownership to a new user by calling the ioctl() function with an open file descriptor, the flag SSGRANT and the UID of the new user.

Upon closing a file descriptor, the driver can either preserve the secret or reset it. A reset will occur when a read file descriptor has been granted and all file descriptors have been closed. If no read file descriptor has been opened and there are no open file descriptors then the secret is preserved.

Additionally, the device preserves its state when it goes through a live update event.

Driver Implementation:

a. Development Environment
   - I use the provided Minix 3.1.8 image running on a VirtualBox VM.
b. Files Modified
   - system.conf:
     i. At the end of the file, I copy and pasted the hello service and changed the name of the service to ‘secret’.
     ii. Modifying the system config file was necessary to let the kernel know what permissions the secret device is allowed.
   - ioctl.h:
     i. At the end of ‘ioc_XXX’ includes, I added the line

```
#include <sys/ioc_secret.h> /* 'K' */
```

     ii. Modifying ioctl.h was necessary because SSGRANT is not native to the kernel. This was an addition that is specific to the secret device and therefore needed to be defined.

c. Driver Source

```
#include "secret.h"

/*
 * Function prototypes for the secret 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( int secret_ioctl,       (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,
    nop_ioctl,
    do_nop,
};

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

/*Current owner of the secret*/
PRIVATE uid_t secOwner;

/*Variable to count the number of times the device has
 *been opened for read/write*/
PRIVATE int open_counter;

/*Boolean to determine if it has been attempted to be read yet.*/
PRIVATE int secRead;
```

```
/*Keeps track of the earliest position in the secret.*/
PRIVATE int secPos;

/*Keeps track of end of secret buffer*/
PRIVATE int secEnd;

/*Buffer to hold the message*/
PRIVATE char secBuffer[SECRET_SIZE];

PRIVATE char *secret_name(void)
{
  return "secret";
}

/*Open the secret keeper device*/
PRIVATE int secret_open(d, m)
    struct driver *d;
    message *m;
{
  struct ucred caller;

  /*Check access type, grab only read and write bits*/
  switch (m->COUNT & (R_BIT | W_BIT))
  {
    /*Read access, here we care that the same UID is reading as written*/
    case R_BIT:
      /*Determine owner of caller*/
      getnucred(m->IO_ENDPT, &caller);

      /*If there is no owner*/
      if(secOwner == INVAL_UID)
      {
     secOwner = caller.uid;
      }

      /*If owner does not match, return error*/
      if(caller.uid != secOwner)
      {
     return EACCES;
      }

      /*Increase FD count*/
      open_counter++;

      /*Mark the message as has been read*/
      secRead = 1;
      break;

    /*Write access, here the first person is made the owner.*/
    case W_BIT:
      /*If there is an owner*/
      if(secOwner != INVAL_UID)
      {
     /*Return error, as it has been written to*/
     return ENOSPC;
      }
```

```
      /*Determine owner of caller*/
      getnucred(m->IO_ENDPT, &caller);

      /*Set caller as the owner*/
      secOwner = caller.uid;

      /*Increase FD count*/
      open_counter++;

      break;

    /*All other access requests*/
    default:
      return EACCES;
  }

  return OK;
}

PRIVATE int secret_close(d, m)
    struct driver *d;
    message *m;
{
  /*Decrement the FD counter*/
  /*Theoretically this should be safe, because they lose access to the
   *device when they close their file descriptor. It shouldn't be
possible
   *to close more file descriptors than there are.*/
  open_counter--;

  /*If read was set and FD is now 0, reset secret*/
  if(secRead && open_counter == 0)
  {
    /*Mark secret as unread*/
    secRead = 0;

    /*Reset the positions in the buffer*/
    secPos = 0;
    secEnd = 0;

    /*Mark as unowned*/
    secOwner = INVAL_UID;
  }

  return OK;
}

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

  int res;

  /*If this isn't a grant request, error out*/
```

```
  if(m->REQUEST != SSGRANT)
  {
    return ENOTTY;
  }

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

  /*If we were able to copy the UID*/
  if(res == OK)
  {
    /*Set the current owner as the the grantee given to us*/
    secOwner = grantee;
  }

  /*Return result of the copy*/
  return res;
}

/*I have been told that this part isn't used for anything*/
PRIVATE struct device * secret_prepare(dev)
    int dev;
{
  secret_device.dv_base.lo = 0;
  secret_device.dv_base.hi = 0;
  /*I figure have it return something that makes a little sense*/
  secret_device.dv_size.lo = SECRET_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)
  {
    /*Copies data to other process (acts as a read)*/
    case DEV_GATHER_S:
      /*If there is nothing left to read from the secret, return 0*/
      if(secPos == secEnd)
     return 0;

      /*Calculate how much to read*/
      /*If the current position + requested size is beyond the end,
output
       *only the rest of the message*/
      if(secPos + iov->iov_size > secEnd)
     bytes = secEnd - secPos;
      /*Otherwise output the requested amount of bytes*/
      else
```

```
    bytes = iov->iov_size;

      /*Copy the contents of the buffer, starting from the current
position
       *and ending at position + io_size, to wherever the process wants*/
      ret = sys_safecopyto(proc_nr, iov->iov_addr, 0,
                      (vir_bytes) (secBuffer + secPos),
                      bytes, D);

      iov->iov_size -= bytes;

      /*Update position tracker*/
      secPos += bytes;

      /*If theres nothing else in the secret, reset to the beginning*/
      if(secPos == secEnd)
      {
     secPos = 0;
     secEnd = 0;
      }
      break;

    /*Copies data into this device (acts as a write)*/
    case DEV_SCATTER_S:
      /*If we are at the end of the buffer*/
      if(secEnd == SECRET_SIZE)
     return ENOSPC;

      /*If we have space, fill up the buffer*/
      if( (iov->iov_size + secEnd) > SECRET_SIZE )
     bytes = SECRET_SIZE - secEnd;
      /*Otherwise save the requested size as the secret length*/
      else
     bytes = iov->iov_size;

      /*Copy whatever the process wants into the buffer*/
      ret = sys_safecopyfrom(proc_nr, iov->iov_addr, 0,
                        (vir_bytes) (secBuffer + secEnd),
                       bytes, D);

      /*As per Prof Nico "You always subtract", got it*/
      iov->iov_size -= bytes;

      /*Update end of secret tracker*/
      secEnd += bytes;
      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 sef_cb_lu_state_save(int state) {
/* Save the state. */
  ds_publish_u32("open", open_counter, DSF_OVERWRITE);
  ds_publish_u32("read", secRead, DSF_OVERWRITE);
  ds_publish_u32("position", secPos, DSF_OVERWRITE);
  ds_publish_u32("end", secEnd, DSF_OVERWRITE);

  ds_publish_mem("owner", &secOwner, sizeof(uid_t), DSF_OVERWRITE);
  ds_publish_mem("buffer", &secBuffer, (size_t) SECRET_SIZE,
DSF_OVERWRITE);

  return OK;
}

PRIVATE int lu_state_restore() {
/* Restore the state. */
  u32_t open, readBoolean, fullBoolean, position, end;

  size_t idLen = sizeof(uid_t);
  size_t bufLen = (size_t) SECRET_SIZE;

  /*Retrieve old data*/
  ds_retrieve_u32("open", &open);
  ds_retrieve_u32("read", &readBoolean);
  ds_retrieve_u32("position", &position);
  ds_retrieve_u32("end", &end);

  ds_retrieve_mem("owner", (void *) &secOwner, &idLen);
  ds_retrieve_mem("buffer", (void *) &secBuffer, &bufLen);

  /*Delete backups*/
  ds_delete_u32("open");
  ds_delete_u32("read");
  ds_delete_u32("position");
  ds_delete_u32("end");

  ds_delete_mem("owner");
  ds_delete_mem("buffer");

  /*Restore old data*/
  open_counter = (int) open;
  secRead = (int) readBoolean;
  secPos = (int) position;
  secEnd = (int) end;

  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 secret driver. */
  int do_announce_driver = TRUE;

  /*Set buffer to nul byte*/
  memset(secBuffer, '\0', (size_t) SECRET_SIZE);

  /*Initialize owner of the secret to something that won't be used*/
  secOwner = INVAL_UID;

  /*Initialize the counter for file descriptors*/
  open_counter = 0;

  /*Init the read boolean*/
  secRead = 0;

  /*Init the position variables*/
  secPos = 0;
  secEnd = 0;

  switch(type) {
    case SEF_INIT_FRESH:
      printf("%s", SECRET_MESSAGE);
      break;

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

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

    case SEF_INIT_RESTART:
      printf("%sHey, I've just been restarted!\n", SECRET_MESSAGE);
      break;
```

```
  }

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

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

PUBLIC int main(int argc, char **argv)
{
  /*
   * Perform initialization.
   */
  sef_local_startup();

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

Driver Behaviour:

The driver performs as expected, next are snippets of the driver performing in the Minix 3 environment with explanations of what is happening below:

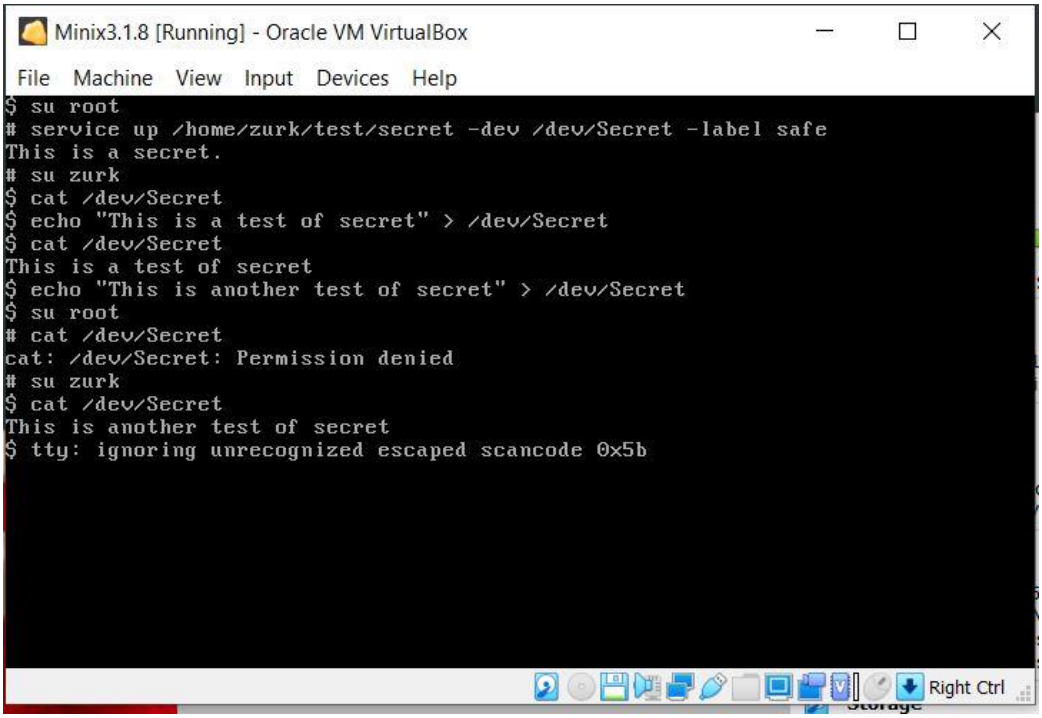

Figure 1: Here we see boot up of the device, and the first test, reading then writing

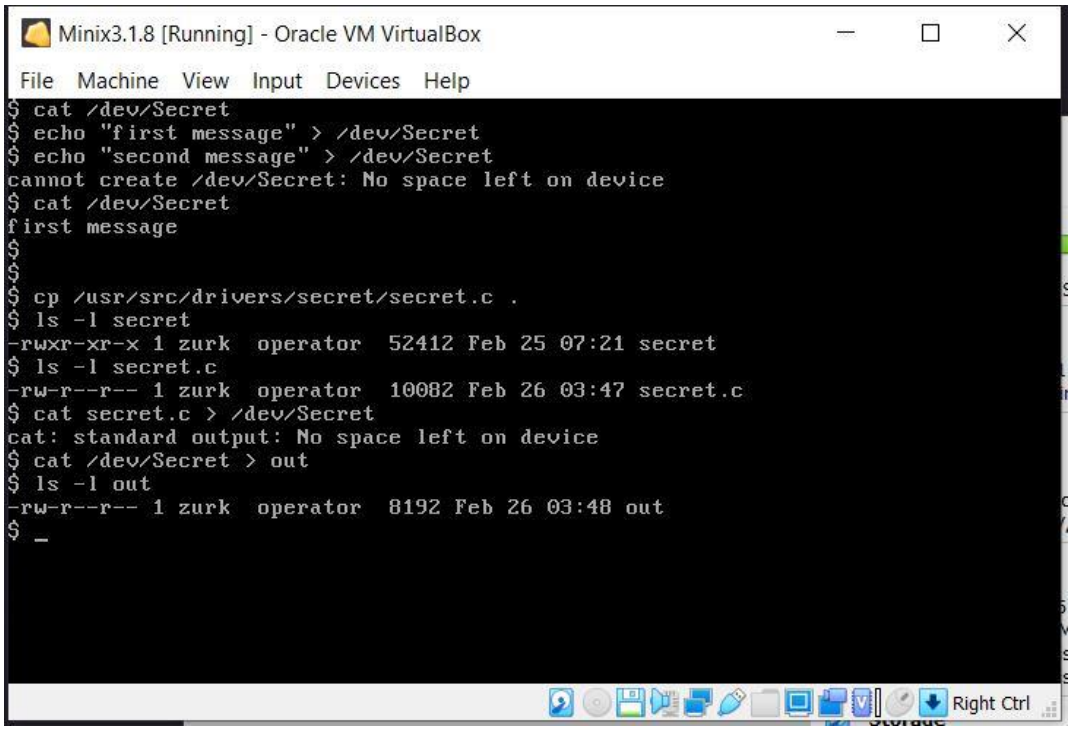

Figure 2: The upper test shows that we can only write once. The lower test shows that it will only accept 8192 bytes of data

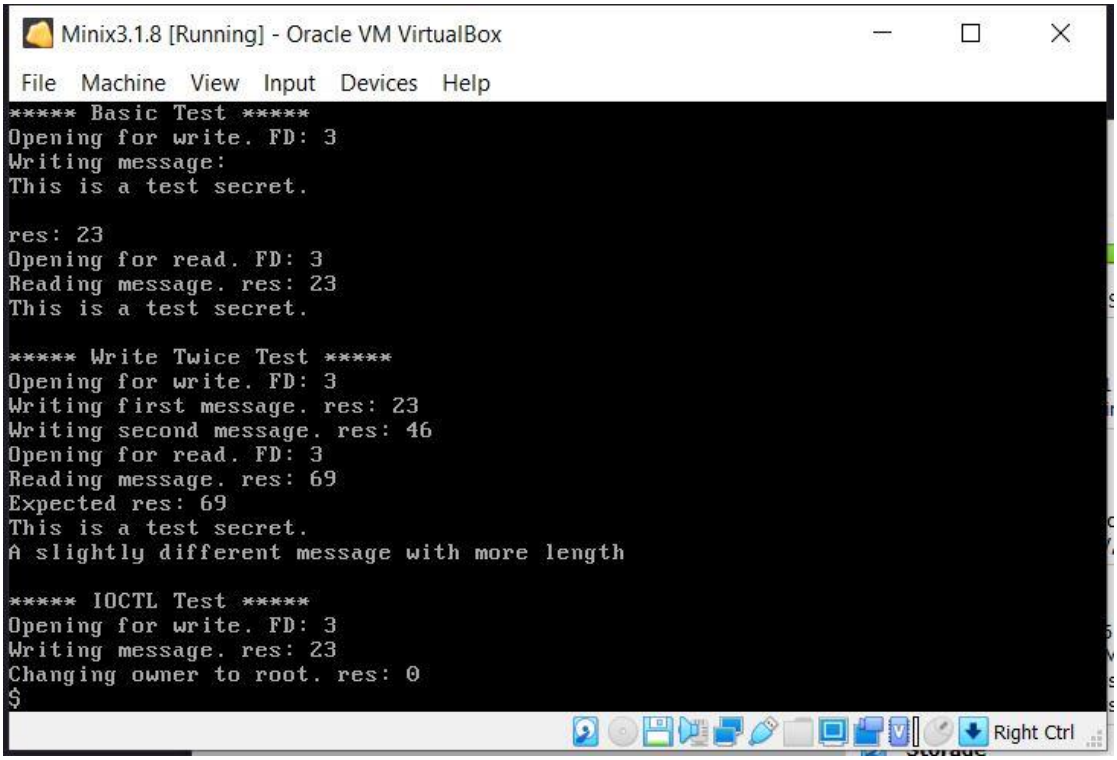

Figure 3: This is the result of a test program I made that tests reading then writing, writing to the same file descriptor, and transferring ownership via IOCTL

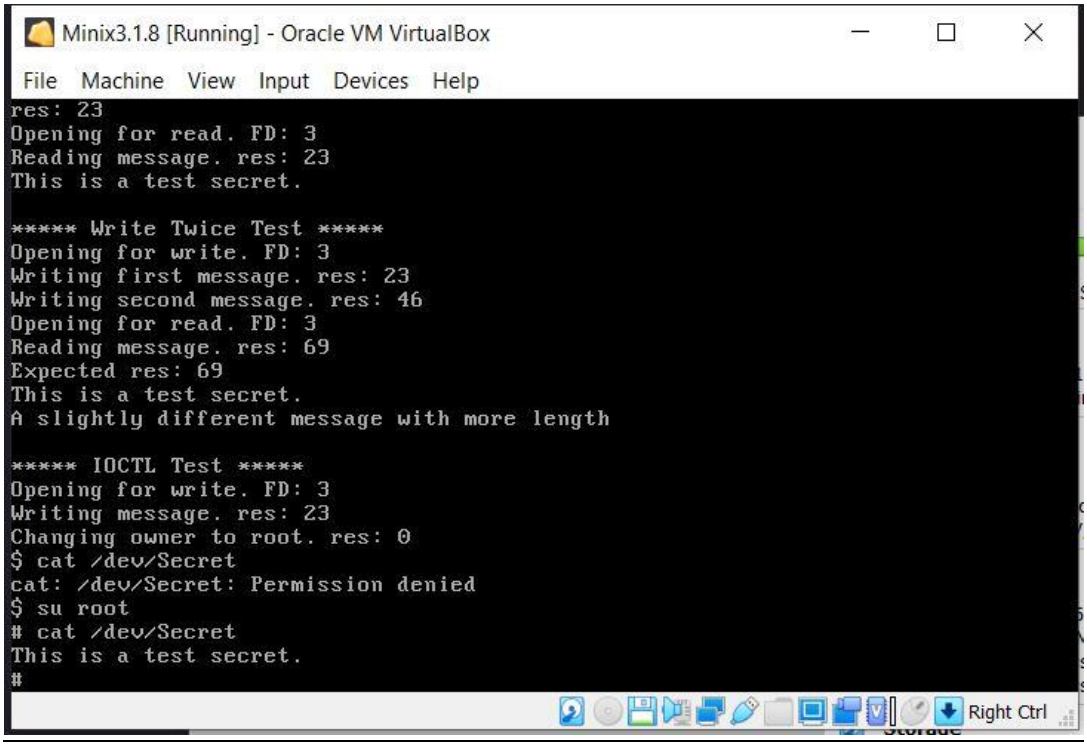

Figure 4: This is a continuation of the IOCTL test, where we see only root can access the secret

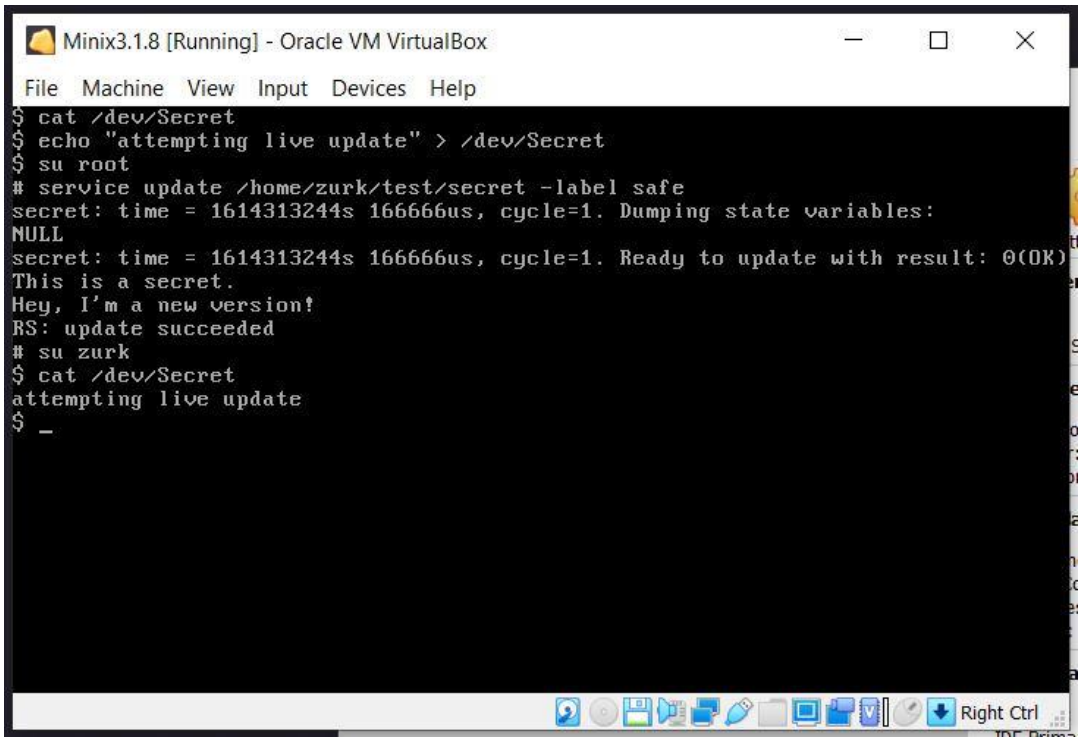

Figure 5: This shows that the message is preserved across a live update

Problems, Solutions, Results, and Lessons:

1. Problem: Minix is a terrible development environment.

   Solution: Because using elle to write code is absolutely awful, mostly due to the keyboard commands not being the same as actual emacs, some files were ported to my main development environment for writing. If files only required a couple lines of modification, that was done in elle. But if files required much more extensive modifications then they were copied to the "floppy disk" and ported to my ubuntu development environment so I could change it in emacs.

   Results: This was an alright solution, as once I had the files in one environment it was fairly simple to make the necessary modifications. The real issue was the time spent transferring the files from one environment to the other. Too much time was spent moving files back and forth and hindered the actual debugging process.

   Lessons: There wasn't really a lesson here, other than developing for a system, outside of that system is very very time consuming.

2. Problem: I was not familiar with the function calls in the minix environment.

   Solution 1: The first solution was to close down my Ubuntu VM, open up my Minix VM, look up what I needed, take a screen grab of it, and boot back into my Ubuntu VM for development.

   Results 1: This also added to the development time of this project and wasn't always reliable, as the function either didn't have a man page or wouldn't show up when searching.

   Solution 2: Eventually I did find a website that had all the man pages for various versions of Minix.

   Results 2: This was extremely helpful and much faster than the original solution. Because I use a second computer to search online, I was able to keep one environment open with the file I was working on and work off the second PC to do research.

   Lessons: The main lesson was to avoid switching environments as much as possible because it wastes time that can be used on much more important things.

3. Problem: The device wouldn't build because it said I had replaced the _send function from <sys/socket.h>.

   Solution: After searching through my device for any mention of send function, and finding nothing, I decided to remove <sys/socket.h> from my included files.

   Results: This actually worked. After removing this line it built just fine, which was super confusing because in the man page for getnucred() it says to include this file. So I don't get why it would say this if including that file would break the program.

Lessons: Even Operating System writers are not infallible, and they will make mistakes when it comes to their manuals.