The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2012

COMP3301/COMP7308
Assignment 2 — Encrypting device driver

Due: 8pm Friday ~~21st~~ 28th September 2012
Weighting: 25% for COMP3301 students (100 marks total)

Version 1.1 — 19 September 2012
See Changelog at bottom for details

Introduction

The goal of this assignment is to give you practical experience writing a device driver for Linux, and to learn how kernel modules are a useful and powerful way of implementing operating systems concepts.

As part of the assessment you must also answer some short–response questions that will test your understanding of the concepts that you have implemented.

You will complete this assignment on the virtual machine image provided on the course website. More information on this image is available under the Resources page.

This assignment may be completed **individually** or in **self-selected groups of two**. Please read the section on group work if you decide to work in groups of two. Although you may share code with your team member (if working in groups of two), it is still considered cheating to look at another student's code or allow your code to be seen or copied by anyone but your partner. You should be aware that all code submitted may be subject to automated checks by plagiarism-detection software. You should read and understand the school's policy on student misconduct, which is available in the course profile.

Overview

The assignment is to create a Linux device driver interface to an encryption coprocessor. You will be provided with a pseudo-hardware device in the form of a kernel module that you will use to perform the cryptography computations. The encryption scheme that the device implements is RC4.

It is highly recommended you complete the two practicals surrounding this assignment (Hello kernel world and character device drivers) before attempting this assignment.

The architecture of the driver that you must implement is as follows:

- The device node is located at `/dev/crypto`.
- After opening the device, processes use `ioctl()` calls to create or attach to one or more encryption buffers, and to initialise the encryption coprocessor.
- Once a file descriptor is attached to a buffer, `read()` and `write()` operations are used to send and receive plaintext/ciphertext to and from the device. You will perform crypto operations on the encryption coprocessor via its API.
- The `mmap()` operation is supported so that processes may inspect a buffer's contents directly.
- After completing the cryptography operations, the process detaches from and/or destroys the buffers using `ioctl()` calls, before finally closing the file descriptor.

Device registration

When loaded, your device driver must register itself with major number 250 using the dynamic allocation method described in the Linux Device Drivers, 3rd edition book, chapter 3 (available from `http://lwn.net/images/pdf/LDD3/ch03.pdf`). The minor number should be automatically assigned by the kernel. **Do not** use the "older way" (`register_chrdev()`) — you will lose marks if you do.

After registration, your driver must print out its assigned major and minor number to the kernel's ring buffer in the following format (`KERN_INFO` is sufficient):

```
crypto: major=MAJOR, minor=MINOR
```

(where `MAJOR` is the assigned major number and `MINOR` is the assigned minor number.) For example:

```
crypto: major=250, minor=0
```

You should ensure when the driver is unloaded it unregisters itself and cleans up any memory it had allocated during operation. Your driver should not leak any memory.

File operations

A skeleton *fops* struct has been defined for you in *a2-fops.c* and *a2-fops.h* (both available on the course website). You should use this as a basis for writing your module. All of the callbacks in the structure need to be implemented.

Buffers

The device uses buffers internally to store the data received through a `write(2)` system call. Initially, the device has no buffers allocated; to use the device a process must make an ioctl call to create at least one buffer (see below for details on all of the ioctl calls).

Each buffer has a unique identifier represented by an unsigned integer. Identifiers are to start at 1 and be incremented every time a new buffer is created. Identifiers may be reused, however it is up to you to guarantee uniqueness across all buffers that exist in the driver at any given time.

Before reading and writing to a buffer, a file descriptor must attach to it through an ioctl call.

A buffer must be explicitly created but can be deleted in one of two ways: either an explicit ioctl call to destroy the buffer, or when there are no file descriptors remaining attached to the buffer. This means that you will need to implement simple reference counting to ensure buffers are removed when the last file descriptor detaches from it. This automatic deletion procedure should be run whenever a buffer is detached from.

Due to this behaviour, when creating a buffer the driver should transparently attach the file descriptor to that buffer, to prevent it from being removed at the next removal point.

Each buffer has a fixed-size of 8,192 bytes (8K) and can be read and written to independently of each other. This means that each buffer has two variables associated with it: a read offset and a write offset, and has first-in, first-out semantics. These 'pointers' determine where the next read and write calls (respectively) will operate from. Writing past the end of the buffer should wrap around to the start, but attempting to write past the current read pointer should ~~fail with -ENOBUFS~~ cause an incomplete write to occur (that is, as many bytes as possible are copied up to the read pointer, and this number of bytes is returned to the user-space `write(2)` call).

A buffer may have at most one reader and one writer attached to it at any given time, which may be from the same file descriptor opened for both reading and writing, or two file descriptors, one opened for reading and the other for writing. Whether a file descriptor is a reader or writer (or both) is specified by its *user-space* access mode to `open(2)` (there are complimentary definitions inside the kernel that match these modes, see the `linux/fs.h` header file):

***User-space* access mode**	**Meaning**
`O_RDONLY`	reader
`O_WRONLY`	writer
`O_RDWR`	reader and writer

This restriction must be enforced when attempting to attach to a buffer, and an error returned if there is already a reader or writer attached (depending on the mode). More on this behaviour is described in the `ioctl.h` file.

Encryption coprocessor

You have been provided with an encryption coprocessor in the form of a kernel module (`cryptodev`). It provides an API that you are able to call into to perform cryptographic functions as needed. More information on this device is available in the `cyrptodev-1.0.tar.gz` archive that you may download off the course website.

Encryption

Each open file descriptor must support a settable encryption mode. This mode specifies to the driver when it should encrypt (and decrypt) data being passed through it, and on which call (read/write). The mode can be specified through the `CRYPTO_IOCSMODE` ioctl call (specified below). This call also allows the key to be set. The key has a maximum size of 255 bytes (plus terminating null character) and can be made up of any ASCII characters.

For instance a process may set its write calls to encrypt, and its read calls to decrypt. This behaviour would assert that the driver is successfully encrypting and decrypting the data given. Alternatively, the process may just set the mode on one of the read/write operations, and leave the other unset (this is called *passthrough*).

Using the device

You must implement the following file operations in the device driver:

open

Opening the device will set up any data structures required to store the state for this file descriptor. ~~Reader/writer restrictions should be enforced in this operation.~~

close

Closing the device will detach from a buffer (if one is attached to) and clean up any memory used by that file descriptor. If the file descriptor was attached to a buffer, and this was the final reference to that buffer, it should be destroyed.

read

A read call can only be made if the file descriptor is already attached to a buffer. If it is, the driver must attempt to copy the requested bytes from the buffer into the user-space process, applying whichever encryption mode is set on the file descriptor. For instance, if the file is set to decrypt on read, then all data being copied from the buffer should be decrypted, and the plaintext is transferred to the process.

If the read mode has not been set at the time of the call, then the driver should act as if it were set to passthrough, and return the raw data from the buffer (regardless of whether or not it is encrypted).

You must implement blocking I/O for your read calls.

write

A write call can only be made if the file descriptor is already attached to a buffer. If it is, the driver must attempt to copy the data given into the attached buffer, applying whichever encryption mode is set on the file descriptor. For instance, if the file is set to encrypt on write, then all data being written to the buffer should be encrypted, and the ciphertext stored in the buffer.

If the write mode has not been set at the time of the call, then the driver should act is it it were set to passthrough, and place the raw data in the buffer (regardless of whether or not it is encrypted).

`-ENOBUFS` should never be returned from a write call.

mmap

You must also implement an `mmap()` file operation in the device driver that allows a user-space process to request a pointer into the buffer it is attached to. If the file descriptor passed through the `mmap()` call is not attached to a buffer, the driver should return `-EOPNOTSUPP`. Attempting to map any size that is not page-aligned or a size greater than the maximum buffer length is an error and the driver should return `-EIO`. This means that the only valid mapping sizes that a process can request are 4K and 8K.

Otherwise, the driver must attempt to map the buffer into the user-space process' virtual address space. If the user-space process requests a specific offset this must be obeyed (providing it is page-aligned and does not exceed the size of a buffer, or `-EIO` should be returned). If `PROT_WRITE` is **not** specified by the user-space program in the protocol field, then a read-only mapping should be created. In this case, the virtual memory area being mapped should be protected from writes so the user-space process faults if it attempts to do so.

For the purposes of this assignment you may ignore the flags field to `mmap(2)` and your device driver does not have to implement this.

ioctl calls

The device driver is to be controlled through various ioctl calls. These are listed below for reference. A more detailed description and examples on each ioctl is provided in the `ioctl.h` file, available on the course website.

ioctl	Description	Arguments	Returns
`CRYPTO_IOCCREATE`	Creates a new buffer	None	Buffer id or error
`CRYPTO_IOCTDELETE`	Deletes an existing buffer	Buffer id	0 on success or error
`CRYPTO_IOCTATTACH`	Attaches to an existing buffer	Buffer id	0 on success or error
`CRYPTO_IOCDETACH`	Detaches from the already attached buffer	None	0 on success or error
`CRYPTO_IOCSMODE`	Set the mode of subsequent read/write calls	`struct crypto_smode *`	0 on success or error

Note that the ioctl definitions have been defined for you in `ioctl.h`. You should use this header file when creating your device driver and test programs. Please do not modify this file — a clean copy will be used during marking and any changes you have made will be discarded. If your implementation relies on any changes made then you may be penalised.

Return codes

Each of the file operations and ioctl calls listed above must return one of the following status codes (unless otherwise described):

Value	Meaning
0	Success
`-ENOMEM`	Not enough memory to satisfy the request
`-EINVAL`	An argument given was invalid
`-EALREADY`	A file descriptor is already attached as a reader or writer
`-EOPNOTSUPP`	The operation cannot occur when not attached to a buffer
`-EIO`	`mmap()` size is not page-aligned or has an invalid size, or the offset is invalid

Test program

As part of the assignment, you must create a test program (written in C) that demonstrates your implementation of the device driver. The test program should be placed in a *a2/test* sub-directory in your repository with a Makefile that can produce the binary specified below. Your test program will be tested against both your implementation of the device driver, and a reference implementation. Note that the test program does not test the entire driver specification.

The test program shall be called `echat` and is a simple chat program that uses an encrypted IPC pipe (via the encrypted device created in the assignment). Since it is modelled on a pipe, it can only have two chat users. Both will open the test program on the same machine, but with different arguments. The first chat user will open the program with one argument (this will create the buffers required for chatting) and the second chat user will open the program with three arguments (which will attach to the buffers specified). Your program should output a sensible usage message and exit with status 1 if the number of arguments given is less or more than what is specified above.

You should ensure that you only open the device with the mode required for each file descriptor. No file descriptor in the test program should have the device open for both reading and writing at once.

You may assume that the device exists at `/dev/crypto` and you may hardcode this into your test program.

First chat user

```
Usage: echat encryption_key
```

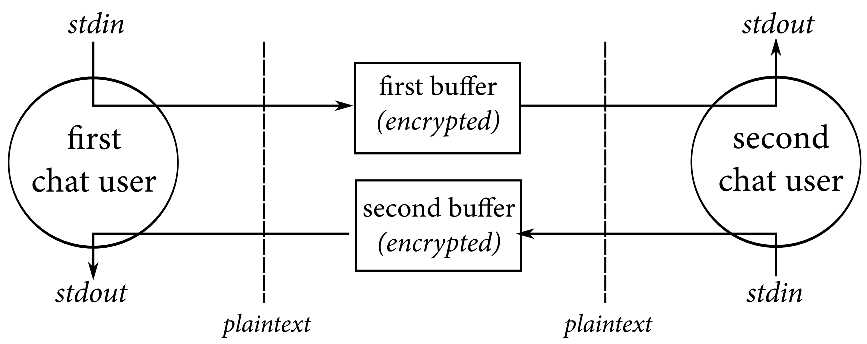
When run with a single argument, the program should open the device twice (once for reading, once for writing) and create two buffers, one for each file descriptor (we will call these the first buffer and the second buffer). The buffer identifiers should be printed on standard error in the following format:

```
first_buffer_id: X, second_buffer_id: Y
```

(where `X` and `Y` are the respective identifiers.)

The first buffer is used for text being sent from the first user to the second user, and the second buffer is used for text being sent from the second user to the first user.

The encryption mode should be set accordingly as per the argument given and the read/write direction required. After setting up the buffers, the program should wait for inputs as per the *Chat behaviour* section, below.



Second chat user

`Usage: echat encryption_key first_buffer_id second_buffer_id`

When run with three arguments, the program acts as the second chat user and attaches to the buffers given. After attaching to the buffers, the program should wait for inputs as per the *Chat behaviour* section, below.

Chat behaviour

The program should spawn two threads (using pthreads) and wait for input lines (terminated by a newline) on both standard in and the user's respective directional buffer (read from the first buffer for the second chat user; read from the second buffer for the first chat user).

When an input line is read on standard in, it should be written to the other buffer (write to the second buffer for the second chat user; write to the first buffer for the first chat user).

When an input line is read from the respective directional buffer (see above), it should be printed to standard out.

You should be able to type text in the first invocation of the program and have it appear on standard out of the second invocation, and vice versa.

End-of-file handling

If an end-of-file (EOF) condition is detected, your program should close the device and exit with status 0. An end-of-file condition is detectable by the read system call returning `0` (which will in turn cause `fgets(3)` to return `NULL`, if you are using it). This occurs when there is no more input available from the given input stream (for instance, standard in). You can test this in `echat` by typing `^D` in a terminal (Control + D). This will cause your program to read EOF from standard in and it should exit as specified. If standard in is redirected from a file, the same behaviour should occur.

Tips

- The driver must not leak memory while running, and must clean up all memory used when unloaded.
- It is recommended you split your implementation across multiple C source files.
- Be aware of allocating large local variables or buffers. You are operating in the kernel and have very limited stack space available to you.
- A linked-list may be useful in representing the buffers in the kernel. The kernel provides a linked-list implementation you may wish to use. See the Resources page on the course website for details.

Short-response questions

1. Why are ioctl calls required as opposed to implementing their functionality in read/write functions?

2. What is the difference between using `kmalloc` and `vmalloc` in kernel land? How would this effect your device driver? Justify your answer with regards to your implementation and how it would differ if you changed from `kmalloc` to `vmalloc` (or vice versa).

3. Discuss the effects of `fork` and the `dup` family of system calls on your device driver. Some things to consider are what happens to buffer reference counting, whether or not the two processes share the same attached buffer, what happens when one closes the device, *etc.* You may wish to write a program that does this and use its behaviour to justify your answer (you do not need to submit any program written for this question).

Code compilation

Your implementation must compile as a Linux kernel module (with a `.ko` extension). It must compile and be loadable on the version of the virtual image provided on the website. See the kernel module practical for information on Makefiles for kernel modules.

Your module will be built by running the following in the `a2` repository directory:

```
make
```

The test program will be built by entering into the *a2/test* subdirectory and running:

```
make
```

You should ensure that the `echat` binary is created with the `-Wall -std=gnu99` compiler flags.

Coding style

Your solution must conform to the Linux kernel coding style document available at `http://www.kernel.org/doc/Documentation/CodingStyle` (or `Documentation/CodingStyle` in the kernel source tree). You may wish to run your code through the `checkpatch` tool to validate that your solution adheres to the coding style. The following arguments may be useful to you (where `FILE` is the C source file you wish to check):

```
checkpatch --no-tree --no-signoff -f --no-summary FILE
```

The tool is available to download from the Resources page of the course website. If you download from another source, make sure you use version 0.32 as this is the version we will use when marking.

You may also find the `indent(1)` tool useful with the following arguments:

```
indent -kr -i8 FILE
```

Please note that these are tools only — they should not be considered perfect. Always double-check the results by hand to be sure.

Group work

If you decide to work in groups of two, **one** member of the group should be chosen to host the repository and that student should e-mail one of the tutors (or if unavailable the lecturer) **no later than seven days before the due date**. After this cutoff it will be assumed that you are working individually and you will be marked as such. Both members of a group will receive the same mark, and any complaints or problems should be directed to the lecturer who will treat each case confidentially.

The tutors and lecturer's e-mail addresses can be found on the course website.

Submission and Version Control

The due date for this assignment is **8pm Friday ~~21st~~ 28th September 2012**. Submission made after this time will incur a 10% penalty per day late (weekends are counted as 1 day). Any submissions more than 4 days late will receive 0 marks. No extensions will be given without supporting documentation (i.e. medical certificate or family emergency)—should such a situation occur you should e-mail the course coordinator as soon as possible.

This assignment must be submitted through ITEE's Subversion system. The repository URL for this assignment is (where *s4123456* is your student number):

`https://svn.itee.uq.edu.au/repo/comp3301-s4123456/a2`

If you are working in groups of two, only one student should use their repository. When submitting group membership to the tutor, you should nominate the student that will host the repository for the group. Permissions will be then set accordingly so you can use your own account. This means both students will checkout and commit to the same repository.

Submissions will be retrieved from your repository when the final cutoff date has been reached. Your submission time will be taken as the most recent revision in the above repository directory.

You are required to make regular commits to your repository as a demonstration of your work. Subversion history will be considered in marking. Do not submit your assignment with a single, large commit. You will be penalised for doing so.

For information regarding ITEE's Subversion system, see `http://student.eait.uq.edu.au/software/subversion/`.

Answers to the short-response questions should be provided in a `responses.txt` file in the specified repository directory.

Assessment Criteria

Grade band	Kernel module and fops (40 marks)		Buffer handling and encryption (30 marks)		Short-response answers (15 marks)		Coding style and comments (10 marks)		Version control (5 marks)	
Excellent	Correct implementation of kernel module and fops with no errors. Clear understanding of device drivers and kernel concepts.	40	Correct implementation of buffers with no errors. Encryption/decryption is handled correctly with no errors.	30	Clear explanations and descriptions of answers given. No incorrect or misleading explanations. Student clearly grasps the assignment concepts.	15	Coding style applied consistently and without any error. Code has meaningful comments where appropriate, with no complex sections left uncommented.	10	Evidence of continual progress through version control history.	5
Very good	Correct implementation of kernel module and fops with very few to no errors. Clear understanding of device drivers and kernel concepts, with only minor issues.	36 32	Correct implementation of buffers with very few to no errors. Encryption/decryption is handled correctly with only minor issues.	28 24	Clear explanations and descriptions of answers given. Very few incorrect or misleading explanations. Student clearly grasps the assignment concepts.	13 11	Coding style applied consistently with few errors. Code has meaningful comments where appropriate, with some complex sections left uncommented.	9 8	Evidence of good progress through version control history.	4
Good	Correct implementation of kernel module and fops with few errors. Good understanding of device drivers and kernel concepts, with some issues present.	28 24	Correct implementation of buffers with few errors. Encryption/decryption is handled with few errors.	22 18	Explanations and descriptions of answers given. Some incorrect or misleading answers. Student has a fair grasp of the assignment concepts.	10 9	Coding style applied with some consistency and with some errors. Code has a fair amount of meaningful comments where appropriate, with some complex sections left uncommented.	7 6	Evidence of some progress through version control history.	3
Satisfactory	Partial implementation of kernel module and fops with some errors. Basic understanding of device drivers and kernel concepts.	20	Implementation of buffers is incomplete with some errors. Encryption/decryption has some errors.	15	Some answers given with some incorrect or misleading answers. Basic explanations given. Student has a basic understanding of assignment concepts.	8	Fair attempt to apply coding style, but some errors or not much consistency. A basic attempt to place meaningful comments throughout code.	5	Little evidence of progress through version control history.	2
Poor	Basic attempt to implement kernel module and fops but has significant errors. Little evidence of an understanding of device drivers and concepts.	16 12 8 4	Basic attempt to implement buffers with significant errors. Encryption/decryption has major errors and/or is mostly incomplete.	13 10 7 4	Very few answers given. Many incorrect or misleading answers. Student has a poor grasp of assignment concepts.	6 4 2	Basic attempt to apply coding style, with many errors or no consistency. Very few meaningful comments in code.	4 3 2 1	Very little to no evidence of progress through version control history.	1
Very poor	No attempt made to implement kernel module or fops.	0	No attempt made to implement buffers or encryption/decryption.	0	No attempt made at short-response questions.	0	No attempt made to apply coding style standards or comment code.	0	No evidence of progress through version control history.	0

Penalty:	**Comments:**
Total mark: /100	

Changelog

v1.0 — 27 August 2012

1. Initial version

v1.1 — 19 September 2012

`ioctl.h` changes:

1. Updated to `ioctl-1.1.h` — please download new version from course website
2. Added missing include so it can be used in user-space without modification
3. Fixed typo in comments of how to use the key
4. Return `-EOPNOTSUPP` if already attached to buffer when issuing a `CRYPTO_IOCCREATE` call

Specification changes:

1. Extended due date to 8pm Friday 28th September 2012
2. Clarified that reader/writer restrictions are to be enforced during `CRYPTO_IOCATTACH`, not `device_open()`
3. You must implement blocking I/O for read calls
4. Clarified that incomplete writes are possible, and that `-ENOBUFS` should never be returned from a write call