Proof–Carrying Code for Linux Device Drivers

Sean C. Rhea

December 17, 1999

1 Introduction

This document is a draft of the information I have learned thus far concerning applying the ideas behind Proof–Carrying Code (PCC) [1] to Linux device driver code. The motivation for this work is as follows: when a company releases an operating system (OS), it usually does not write all the device drivers for that OS itself. In fact, most device drivers are written by the manufacturers of the devices they drive. This is a problem because the drivers generally run (at least partially) in kernel mode, so they have the ability to crash the machine. For this reason, it would be quite helpful to prove the safety of device drivers before allowing them to be installed. In one sense, this will allow for OSs to be more stable; in another, it will prevent the reputation of an otherwise stable OS from being tarnished by buggy device drivers causing it to periodically crash.

This research so far has been divided into two parts. The first part consisted of learning the Linux device driver application programming interface (API), specifically through looking over the *psaux* driver while reading [2]. This is the driver for the PS/2 mouse port; it was chosen for its relatively small amount of code and simple hardware. The second part has consisted of trying to formalize this API and devise a method of constructing proofs with respect to it.

2 Linux Device Driver API

The majority of my research so far on this project has been into the Linux device driver API. Before we can decide what constitutes "safe" behavior for a device driver, we need to understand the interaction between that driver and the OS.[1] This interface can be logically divided into several groups of functions, which I discuss below.

2.1 Initialization and Finalization

The Linux kernel has support for *loadable modules*, sections of code that can be dynamically linked into the OS as they are needed. In this section, I will assume that all drivers are loaded in this manner.

A module is loaded into the kernel by calling the function `init_module`, which is defined by the writer of the device driver. It is the responsibility of this function to register with the kernel any facility that device driver intends to provide. For instance, with the *psaux* module, the driver will register `/dev/psaux`. Registration with the kernel is accomplished by calling the `misc_register` command, which takes a pointer to a structure of type `miscdevice`, which is defined in the header file `miscdevice.h` like this:

[1] Note that we make no attempt to understand or verify the interaction between the driver and the hardware that it is driving. For instance, one thing that we might like to verify is that the driver does not cause the hardware to do a DMA into an arbitrary section of memory. However, there is no way for us to know what particular sequence of reads and writes causes the hardware to begin such a DMA, and without trusting the driver writer, we have no way of figuring this information out. This is unfortunate, and should be a subject of later research.

```
struct miscdevice
{
    int minor;
    const char *name;
    struct file_operations *fops;
    struct miscdevice * next, * prev;
};
```

minor takes a device identifier as an integer, while name provides a text string which describes the device. The next and prev pointers reveal that the kernel will store these devices in a list. The most important member of this structure, though, is fops, which is defined as follows:

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, const char *, int);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

As the reader might guess, this structure allows the device driver to register all of the callback functions necessary for users of the device to access it. For instance, the open member provides a pointer to the function to be called when a user makes an open system call on this device.

In any case, the device may do other things in init_module besides register itself. One common thing is for a device to allocate memory for any persistent data structures it needs. This memory is allocated through kmalloc, which will be further discussed later in this document.

The complement function to init_module is cleanup_module. In it, a device driver should undo the actions it performed in init_module. For instance, it should unregister itself with the kernel through the function misc_deregister and release any memory it has obtained through kmalloc with kfree.

2.2 Opening and Closing the Device

Once a module has been loaded into the kernel, the device it registers is ready to be operated on by users. In general, Unix devices must be opened before they can be read from or written to and must be closed when they are finished being used. The system calls for opening and closing a device are open and close; with a device driver, they will cause the kernel to call the open and close functions registered with the file_operations structure in init_module.

The open function should ready the device for being read from or written to. This is generally accomplished by either enabling interrupts for the device or setting up some form of periodic polling of the device; interrupts are the method used by the *psaux* device and will be discussed later. Also, once the open function has determined that it can successfully open the device (it is present and not busy, for example), it should increment the module use count and return 0. The module use count is used to determine when the kernel can automatically unload the module. A return value of 0 from `open` indicates success; a negative return value indicates failure.

The complement to `open` is `close`; it is responsible for decrementing the module use count and possibly unregistering any interrupts set up by `open`. For example, if the given call to `close` is due to the last reader of the device closing it, any interrupts the device has enabled should probably be disabled, since there is no one left to need data from the device.

2.3 Reading and Writing to the Device

Reading from the device is done through the `read` function pointer, writing through the `write` function pointer. The OS ensures that the device is open before either of these is called (it will generate a runtime error if this occurs). There really is not much to check with these functions.

2.4 Interrupts

Interrupts are registered through the kernel function `request_irq`, which is defined as follows:

```
int request_irq(unsigned int irq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *device,
                void *dev_id);
```

`irq` specifies the interrupt request number, `handler` supplies a function to call when the interrupt occurs, `device` is a character string describing the device registering the interrupt, and `dev_id` is a unique pointer identifying this particular device. If this is the only device that ever uses this IRQ (as in the case of the *psaux* device, `dev_id` may be `NULL`. Finally, `flags` is the binary OR combination of various integers which specify qualities of the interrupt function to be registered. The most important of these is `SA_INTERRUPT`, which indicates that the `handler` function is fast, and should be called with interrupts disabled. If this flag is not specified the kernel goes through some additional overhead, including re–enabling interrupts, before calling the `handler` function.

The complement to `request_irq` is `free_irq`, which is defined like this:

```
extern void free_irq(unsigned int irq, void *dev_id);
```

The `irq` and `dev_id` passed to `free_irq` should be the same ones that were passed to the corresponding `request_irq` call. Otherwise, the function is unremarkable.

2.5 Obtaining an Releasing Memory

The next set of interesting kernel functions I have studied are `kmalloc` and `kfree`, which are used for obtaining and releasing kernel memory for device drivers.[2] Unlike the standard C function `malloc`, `kmalloc` takes a flag argument in addition to a size argument. This flag can take on different values, but the two most important are `GFP_KERNEL`, which allows the caller to be put to sleep if the memory is not yet available, and `GFP_ATOMIC` which causes `kmalloc` to return `NULL` if it cannot satisfy the request. Clearly, `kmalloc` should not be called with `GFP_KERNEL` from within functions where interrupts are disabled.

[2]Actually, there are several other interesting functions involved with kernel memory (especially `put_user` and `get_user`, which copy memory to and from the user space), but I have not thought about them in detail yet, so I will not discuss them here.

2.6 Going to Sleep

The final set of interesting functions in the Linux kernel allow a driver to go to sleep. In other words, they allow the driver to put itself back on the run queue and transfer control to another process. This is useful if the driver is polling for some external event that may take some time to occur. For example, the *psaux* driver's `read` function contains the following code:

```
while ((inb(AUX_STATUS)&0x03) && retries < MAX_RETRIES) {

  /* AUX_OBUF_FULL checks bits 5 and 0 of the status byte,
     which represent */

  if ((inb_p(AUX_STATUS) & AUX_OBUF_FULL) == AUX_OBUF_FULL)
    inb_p(AUX_INPUT_PORT);
  current->state = TASK_INTERRUPTIBLE;
  current->timeout = jiffies + (5*HZ + 99) / 100;
  schedule();
  retries++;
}
```

It is not necessary to fully understand this code to recognize the technique. Essentially, the `while` loop is polling the status flag of the device for input bytes. It knows (approximately) how far apart the various bytes are coming in, so after each byte it goes to sleep for some time before polling again.

3 Proofs

Given the above descriptions of the various kernel functions, let me now define some of the things we might want to verify about them and in manner this verification might be done. Note that I will only discuss several of the functions, since this portion of the research is less well understood and I am still formulating much of it currently.

3.1 Logs

In order to generate proofs about device drivers, we need to formulate a logic in which to specify the proofs. Essentially, we will use first order logic with one extension—the concept of a *log.* A log is a sequence of events known to have occurred at any part in the code. They will be used to check kernel function pre–conditions and device driver function post–conditions. Logs will also be used to guarantee certain pre–conditions on driver functions. Additionally, functions with side effects will enter events into the log to indicate what changes they have made to the global state. I will try and clarify each of this instances with an example; in later works I hope to formalize these notions.

3.2 Log Events

First, allow me to give several examples of the types of events functions with side effects will enter into the log. Consider, for example, the `open` function. We can imagine that it would enter a simple log event such as

$$\text{open}$$

On the other hand, a function with more arguments, such as `request_irq`, might enter a more complicated event such as

$$\text{request\_irq}(N, \text{fp}, \text{flags}, \text{nm}, I)$$

Here, the log event is parameterized by the various arguments that the function was called with.

3.3 Log Predicates

Given some set of log events, we can define log predicates over them. These predicates will be used to check kernel function pre–conditions and device driver function post–conditions, and to guarantee certain pre–conditions on driver functions. For example, let us specify the pre–condition to `free_irq`—that the given interrupt was previously requested by a corresponding call to `request_irq` and has yet to be freed—as follows (note that the dot ($\cdot$) operator indicates log concatenation):

$$\frac{}{irqNotFreed(req\_irq(N,\_,\_,\_,I)\cdot l, N, I)}$$

$$\frac{E \neq req\_irq(N,\_,\_,\_,I), free\_irq(N,I) \quad irqNotFreed(l)}{irqNotFreed(E\cdot l, N, I)}$$

In other words, if the head item in the log is a request event at the point at which `free_irq` is called, then the IRQ is not yet freed, and the function's precondition is met. Otherwise, if the head element of the log is not a request or free event for this IRQ, and if the IRQ had yet to be freed in the remainder of the log, then the IRQ is not yet freed, and the function's precondition is met.

As an example of guaranteeing a certain precondition on a user function, consider the `read` function. As mentioned above, the kernel guarantees that the `read` function will not be called on a device that has not yet been opened. This fact may be necessary to prove that the `read` function satisfies the safety policy, so it should be listed as one of the function's pre–conditions.

Finally, there are post–conditions for driver functions that we will want to check to guarantee their safety. An example of such a condition for the `close` function is as follows: if this is the last open instance of this driver, any interrupts requested between now and the initial opening of this driver should be freed by the time this function exists. In other words, a device driver should free all the IRQs it requests. This precondition would require a predicate *last_close*, which I do not define here, but which would be defined in a similar manner to the logic for balancing parenthesis in a mathematical expression.

4 Conclusion

We have done some preliminary research on the topic of applying the ideas of Proof–Carrying Code to device drivers under Linux. We have documented the Linux kernel API and identified promising areas to apply PCC. Additionally, we have formulated the beginnings of a logic in which to specify proofs about device drivers. In the remaining portions of this research, we hope to develop a theorem prover for this logic and verify a simple safety policy for several device drivers under Linux.

5 Acknowledgments

George Necula has been an integral part of this research throughout the project. He proposed the initial idea for the research, as well as the concept of a log. Additionally, I have frequently consulted with him about other areas of the work.

References

[1] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[2] Alessandro Rubini. *Linux Device Drivers.* O'Reilly, Cambridge, Massachusetts, 1998.