Linux Kprobe Keyboard Logger

March 2, 2009

1 Assignment

The goal of this assignment is to build a Linux AT/PS2 scancode logging module. The module will use the kprobe instrumentation mechanism to collect scancodes and export them using debugfs to be read by a user application What the application does with the scancodes is outside the scope of the assignment. Presumably it logs them to disk or may take some other action.

For this description, we will use the in-kernel data structure terms to avoid any confusion. The VFS term translation is as follows:

- an *inode* is the file on disk
- a *dentry* is the naming structure for files or directories
- a *file* is an open file descriptor.

If you need assistance via email for this assignment, please follow good Bug Reporting Guidelines [3]. It is very difficult for us to offer remote assistance without enough information about what you are doing. Requests for help not following these guidelines will receive **only minimal responses**.

2 Module

Your module will capture **unprocessed** AT and PS2 scancodes from the AT keyboard interrupt handler. Scancodes **must be** stored in a ring-buffer large enough to hold exactly 16 scancodes. It will make these scancodes available via a debugfs inode named by the dentry "`atkbd/scancodes`" from the debugfs root. Multiple programs **must** be able to open this inode and obtain scancodes without interfering with each other. Your module should not disrupt the stability of the kernel, and should therefore not allocate memory for read requests.

This is largely an interfacing problem between kprobes and debugfs. For capturing scancodes, we will be using the `kprobe`s mechanism. You will need to find the appropriate instruction(s) to probe to capture the scancodes. Interrupts are already disabled by the time interrupt handlers are invoked, so waiting on a mutex will hang the kernel. This motivates building a preemptable transaction in your inode implementation.

You will export an inode with the following properties. When the inode is first opened, the file must have available the most recently captured scancodes that are still live in the ring buffer. This is illustrated in Figure 1. When a new scancode is captured, it is logically placed at the end of the file, as in Figure 2. Figure 3 shows what happens when your buffer is full and a new scancode will evict an old one. In this case, attempts to read an evicted scancode must result in reading bytes containing zeros. This allows applications to detect missed scancodes.

At a minimum, you will need to implement the `open`, `read`, `llseek`, and `release` operations for your debugfs inode, though some may be trivial. Each of these functions is documented by an analogous Unix `man` page and in the *Understanding the Linux Kernel* book. `Reads` will need to implement a preemptable transaction, as an interrupt could come at any time and update your ring buffer.

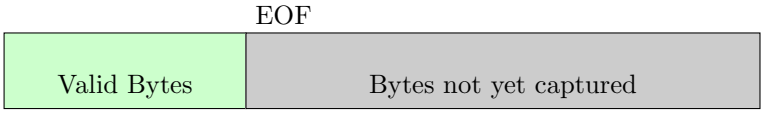

Figure 1: Freshly Opened File.

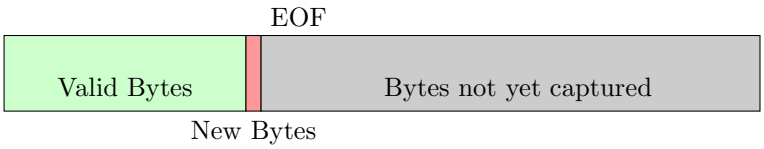

Figure 2: New Scancode.

2.1 Requirements

- Use the *kprobes* instrumentation mechanism. `kprobe`, `jprobe`, or `kretprobe` is acceptable.
- Buffer 16 scancodes in a ring buffer. You **must** overflow at this point, I don't want to type all day or run arbitrary qemu scripts. Note that this isn't practical in a production implementation, so you probably specify this as a macro.
- Export your buffer as an inode available through the dentry "`atkbd/scancodes`" within the root of the debugfs filesystem.
- When an application opens the scancodes inode, the first (index: zero) byte of the newly opened file should be the oldest byte remaining in ring buffer. Bytes which were **not** written by logging scancodes should not appear here.
- When an application attempts to read a byte which is no longer in the buffer, that byte will be read as a zero. (The zero scancode indicates a fatal keyboard error, and is a reasonable result. There are better choices, but this is a bit more legible.)
- Applications **do not** evict bytes from the scancode buffer by reading them. This is **not** a producer-consumer model.
- Any number of applications must be able to open the scancodes inode, obtaining different open files, each with its own logical mapping. This should work for the same application opening the inode multiple times.
- Applications must have a method of waiting for new scancodes to arrive and read them.
- Do **not** worry about shared open files. The Unix specification requires that applications sharing open file descriptors manage their own synchronization. However, individual open files **must** operate independently.

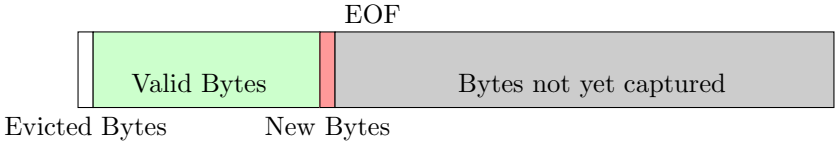

Figure 3: Evicted Scancode.

- You **must** support 64-bit file offsets.
- **Do not** worry about overflowing 64-bit indexes. Assume that less than 2^{60} characters will be typed before reboot.
- You **must** allow applications to seek in the file.
- You **must** clean up any dynamic memory you have allocated at open, if any, during close.
- You may assume you are running on a single-processor system. You may implement an SMP mechanism if so inclined.
- **Do not** add any overhead to a probed interrupt handler other than a constant-time algorithm.
- Do **not** worry about spurious keyboards. Assume that the keyboard and interrupt system are working correctly.
- Do **not** over-engineer this problem. In particular, kernel top-half and bottom-half drivers are unnecessary and **will not** be graded.

3 Build Environment

3.1 On cs418

In the following descriptions, commands run on the host machine are run as some user, and use the "`$`" prompt. Commands run on the guest are run as root, and therefore use the "`#`" prompt.

On CS418, copy the directory /home/kprobe/build to your home directory using:

```
$ cp -r /home/kprobe/build ~/
```

Then give yourself read authority by typing

```
$ chmod -R u+w ~/build
```

You may wish to commit this to a version control system at this point. We recommend mercurial

```
$ hg init && hg commit -m "initial import"
```

The file atkbd_scancodes.c is a Linux kernel module skeleton which you can edit. To create the kernel module, run make.

```
$ make
```

To run qemu with a file system using your modules drive, use:

```
$ make qemu-vnc
```

This will create a new qemu disk image for you to run, generate a modules disk image loaded with your module and some test programs, and launch qemu. You will be given a series of commands for you to use ssh to access qemu remotely using vnc. Once connected, the root password to the image is "kprobes" and your kernel modules are installed in /mnt/modules. To install your kernel module in the guest system type:

```
# insmod /mnt/modules/atkbd_scancodes.ko
```

To remove it type:

```
# rmmod atkbd_scancodes
```

The `/mnt/modules` filesystem is generated on the host as `modules.img`. You can recreate this file without relaunching qemu, but be sure to unmount `/mnt/modules` first. To rebuild only your modules drive, type:

```
$ make modules.img
```

We do NOT recommend reloading new kernel module versions without rebooting. Loading a newer module version into a running system isn't a problem. However, there is no guarantee that the old, and by definition buggy, driver has not corrupted the system in some yet unnoticed way. In this case, it is more likely that you will be simultaneously debugging two interfering modules instead of one.

Be very careful about using any networking on these systems, including SSH, as all virtual machines share private keys. If you must load new modules into your running system, we would prefer that you do not use SSH. Instead, please `unmount /mnt/modules`, rebuild the modules image, and `mount /mnt/modules`. Your new modules drive will be available now.

Do not leave any important files in the qemu guest system. Any changes to the qemu system will be removed when performing a `make clean` or `make distclean`. Your modules image will be recreated whenever running qemu. If you want to have files available in the guest, place them into the `my-modules` directory. The default modules filesystem is about 80MB and should have plenty of room for anything.

Two applications have been provided for your use. **These are the applications we will use to test your module.** The `reader` application is nearly identical to `tail -f`, insofar as it sequentially reads bytes from a file and pumps them to `stdout`. However, it ignores the reported file length and does not attempt to manage truncated files. It can be used in place of a scancode logging program. If no file is specified it uses `stdin`.

The `tester` program is identical to the reader, except it outputs human readable messages instead of raw bytes. Additionally, it waits for a character from `stdin` before reading from the file. Each call to `read` will report how many bytes were read followed by reports of each byte in character, hexadecimal, and decimal formats. It requires a file and does not fall back to `stdin`.

3.2 On your own PC

Simply copy the /home/kprobe directory to another computer. You will need to specify the full path of the local kprobe directory in the Makefile by changing the `KPROBE_HOME` variable. You will need approximately 11 Gb of storage and Linux running on x86. We will not support cross compiles. You will need qemu and most of the available Linux development tool chain.

You may run qemu locally without vnc by using the command:

```
$ make qemu
```

REMEMBER TO TEST YOUR FINAL SUBMISSION ON CS418. All submissions will be graded on cs418 and must compile and install to receive credit.

4 Hints

You will need to probe some instruction from `drivers/input/keyboard/atkbd.c`.

Since you are instrumenting an interrupt handler, you will not be able to use any mutex when writing to your buffer. You will need to implement a transaction. We recommend keeping a counter for the total number of captured scancodes. Before the kernel responds to a read request, it will copy this counter to a declared `volatile` location. It then performs the string copy, and when finished, checks to see if the counter has changed mid-copy. If so, it simply re-executes the entire transaction before returning control to the process. During a write to the buffer, increment the counter. Since interrupts are disabled, writes happen during a critical section on a uniprocessor.

To get this to work for SMP, you would need to keep a transaction boolean that indicates you are changing data. This is easily implemented by using the least significant bit of the counter, and atomically incrementing it twice: once before writing, and once after. A thread reading will be able to tell whether the buffer has changed, or is changing, and rollback the write transaction. The only difference in read logic is to mask the least significant bit after reading the counter. Spinlocks aren't strictly constant-time. However, we will argue that these events happen infrequently, so in general, we won't see any queuing or starvation.

Why don't reading threads starve? Keyboard controllers have a minimum repeat rate of 250 ms, and in general humans can't type faster than 600 characters per minute. A character every 100 ms is incredibly fast for a human input, and it's difficult to sustain. Remember that scancodes produce a down code and an up code, so there will be at least two codes for every character, but we now have a worst-case average of 50ms between events. Memory operates at speeds in tens of ns, fast enough to read many scancodes between key events. To calibrate, your average programs run in time slices in this range. Of course, this assumes that escape code sequences fit into this 50 ms boundary, or don't overflow the buffer entirely. Provided that we make the buffer large enough to amortize the reading cost over escape sequences, we will be fine.

This assignment will not require much written code, but will be heavy in development and reading. You will need to read the VFS chapter of Understanding the Linux Kernel. You will want to read a bit about using debugfs [5] [1] and you may want to lookup an AT keyboard scancode chart [4] to debug your output. You can learn more about kprobes by reading the Linux documentation [2].

References

[1] http://cs418.cs.jhu.edu:8080/source/xref/include/linux/debugfs.h.

[2] Kprobes documentation. http://cs418.cs.jhu.edu:8080/source/xref/Documentation/kprobes.txt.

[3] Mozilla bug writing guidelines. http://developer.mozilla.org/en/docs/Bug_writing_guidelines.

[4] Scan codes demystified. http://www.quadibloc.com/comp/scan.htm.

[5] Debugfs. http://lwn.net/Articles/115405/, December 2004.