Why complexity is bad and open source is good

ATTACK OF THE KILLER FTRACE

When a test kernel starts wrecking network cards, the community gets busy. **BY KURT SEIFRIED**

Complexity is an amazing and terrible thing. One of the first Intel CPUs was called the 4004 (released in 1971) and contained 2,300 transistors. The Pentium 4 (released in 2000) had 48 million transistors, and a modern Quad core CPU has around 2 billion transistors. In 2008 the Linux 2.6.27 kernel surpassed 10 million lines of code. Personally, my experience has been that more transistors mean a better computer. On the opposite side, despite faster hardware, Linux still seems to take the same amount of time to boot up and present a working desktop.

Network Card Killings

In late August 2008, a number of reports regarding e1000e network cards became public [1]. Not just the typical “I upgraded my driver and my network stopped working,” but actual “my network card has stopped working and is not being enumerated at boot time and it’s completely dead now” reports.

The good news is that hardware is cheap. The bad news is that hardware is cheap.

In an effort to extend hardware capabilities and cut down on costs, many vendors have given hardware devices (network cards, etc.) upgradeable firmware. Firmware allows vendors to fix devices that have already been sold to customers and deployed (a heck of a lot cheaper than a recall); firmware also allows vendors to upgrade hardware and add new capabilities on the fly. Of course, this means that a device with corrupted firmware is unlikely to work properly, if at all.

Although it’s pretty obvious that upgrading to a test kernel shouldn’t kill your network card, what do you do when it does? If you’re a Linux kernel developer, you normally just fire up your favorite kernel debugger and simply recreate the circumstances of the problem, watch it crash in detail, and figure out what happened. Unfortunately, when testing a bug that involves killing network cards, it can become difficult to test the problem (especially when trying to convince others to reproduce the test on their own system since your system now has a dead network card that won’t respond at all).

Because of the subtle nature of the bug and the difficulty in testing it, the first few attempts to narrow the bug down and fix it were not successful.

If you’re Intel and you have a major operating system killing your network cards, you can simply take a pile of the network cards, a slightly larger pile of engineers, and lock them into a room. The engineers then start trying different versions and patches of Linux to see where the problem occurs. Once you’ve narrowed down which patch or patches are responsible, you can analyze the code and determine what is going on.

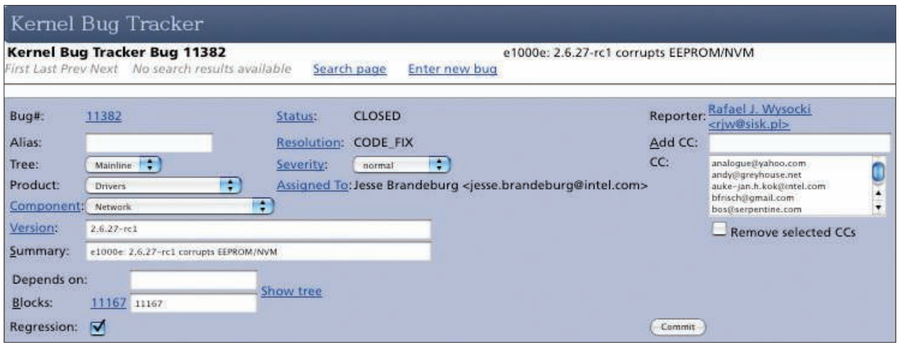
What Went Wrong?

One of the things *ftrace* was designed to do was provide simple function call tracing. In this mode of operation, a line is output for every called function and the caller of this function. ftrace uses the profiling mechanism built into GCC, which in turn adds an *mcount()* to every function so that, when the function is called, data is logged. Unfortunately this creates a performance penalty (especially because most systems don’t use ftrace functionality). To avoid this, kernel developers used memory patching to replace the function calls with noops (no operations), something most CPUs handle very quickly.

This solution has one major drawback, however: You must patch kernel code while the kernel is running, something that can result in Very Bad Things if it doesn't go completely right. To accomplish this safely and quickly, ftrace makes a log of every incoming *mcount()* call and fixes them. However, doing this one at a time is very slow, so the kernel developers took the clever approach of batching the *mcount()* replacements. As a result, there is a window of time in which the *mcount()* call no longer exists in the same memory space; thus, if the kernel patches it, the kernel will in fact be patching a location in memory that can contain virtually anything. The Linux kernel developers considered this and decided to use *cmpxchg* (compare and exchange) [2], which looks at memory and compares it. If it matches what was expected, it then patches memory; if the contents do not match what was expected (i.e., no call to *mcount()* is present), it does not patch memory. In theory this is completely safe.

Here, things become complex (as if they weren't already). On 32-bit architectures, the memory returned from *vmalloc()* and *ioremap()* share the same address space. The kernel uses *vmalloc()* for loadable modules, but the e1000e driver uses *ioremap()* to map its memory space. Once the module initialization of ftrace is finished, the *__init* functions are removed from memory and that space is freed. This free space is then used by the e1000e driver; unfortunately, the cmpxchg behavior is undefined when patching driver memory. As you can probably guess, it allows writes, even if memory did not match.

Now normally, writing to a driver's memory address space should simply cause the driver to fail catastrophically, not the hardware. But in the case of the e1000e driver, it left the hardware open to writing, thus allowing the kernel to write random data to its firmware, wrecking the hardware (in some cases it was recoverable by rewriting the firmware). The good news is that Intel has fixed the driver. At boot-up the driver prevents any writes to the card's firmware, and ftrace has been fixed so that it doesn't overwrite the wrong memory.


Kernel Bug Tracker

Kernel Bug Tracker Bug 11382 e1000e: 2.6.27-rc1 corrupts EEPROM/NVM

First Last Prev Next No search results available Search page Enter new bug

Bug#:	11382	Status:	CLOSED	Reporter:	Rafael J. Wysocki <rjw@sisk.pl>
Alias:		Resolution:	CODE_FIX	Add CC:	
Tree:	Mainline	Severity:	normal	CC:	analogue@yahoo.com andy@greyhouse.net auke-jan.h.kok@intel.com bfrisch@gmail.com bos@serpentine.com
Product:	Drivers	Assigned To:	Jesse Brandeburg <jesse.brandeburg@intel.com>		
Component:	Network				☐ Remove selected CCs
Version:	2.6.27-rc1				
Summary:	e1000e: 2.6.27-rc1 corrupts EEPROM/NVM				
Depends on:					
Blocks:	11167 11167	Show tree			
Regression:	✓			Commit	

Figure 1: The e1000e bug makes news.

The good news is that for most of us, the Linux kernel development process worked. New features were placed in a test kernel, then it was run by developers and other bleeding edge people (Ubuntu Intrepid) who found a serious problem, which was fixed. This kernel was never released by a major vendor as a default option (they tend to be conservative about kernel updates for just this reason), and very few people were affected. Additionally, tools such as ethtool can be used to recover broken cards by writing a new firmware image to it. Because so many people had access to the source code, they were able to apply or remove various patches easily to localize exactly which pieces of code were causing problems. Doing this in a closed source environment would be impossible (everyone would be at the mercy of the vendors with the source code. In this case, with two vendors (OS and hardware), they would have had to cooperate to trace down the exact problem. With GPL-licensed code, anyone who wants can test all the code and track the problem down.

Conclusion

This problem was (potentially) one of the worst bugs in Linux in a long time. Despite being an incredibly complex issue – relying on two separate flaws and specific hardware – it was caught relatively quickly and was largely confined to a small subset of kernel versions. Also, because it is a firmware issue, recovery was possible with the ethtool program. Intel has reportedly made images of the firmware available that can be used to recover broken cards [3].

Because we all have access to the source, we can not only make an informed decision about fixing our systems, we have the ability to do so. ■

But the standard says 32 max!

Another example of complexity throwing a wrench into the works is the 802.11 wireless standard(s). The standard says that the service set identifier (SSID or ESSID) can contain between 0 to 32 characters (ASCII, values from 0 to 255 are allowed), allowing the network name to be long enough to be human readable ("Free Coffee Shop Wi-Fi" vs. "1ad834d7"). Simple enough, right?

Wrong. The frame header for an 802.11 packet uses an 8-bit unsigned value to specify the actual space available for the data, meaning that the SSID parameter can be between 0 and 255 characters. Unless you dug deeply into the specification and were looking for things that could go wrong, you wouldn't know this fact. Because the SSID usually is assumed to be a maximum of 32 characters, it is often copied straight into fixed-length buffers without any checking of the actual length. This recently bit Linux (CVE-2008-4395) when the ndiswrapper – software that allows Windows wireless device drivers to be used on Linux – was found vulnerable to a buffer overflow that could be exploited remotely by a single wireless packet.

INFO

[1] State of e1000e bug: *http://lwn.net/Articles/301251/*

[2] Source of e1000e bug: *http://lwn.net/Articles/304105/*

[3] Cause of e1000e bug: *http://ostatic.com/174457-blog/likely-cause-of-intel-e1000e-bug-discovered*

THE AUTHOR

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.