



Memory Matters

Ed Nisley

The Difficult is that which can be done immediately; the Impossible that which takes a little longer.

—George Santayana

Applications programmers regard computer memory as an essentially endless line of identical bytes awaiting their data structures. Systems programmers take a more nuanced view, reserving distinct regions for the operating system, application code and data, and perhaps a few memory-mapped peripherals.

Embedded systems folks, alas, must contend with downright weird configurations. The intimate relationship between code and hardware occurs well below the usual levels of abstraction, where physics sets the speed limits, and manufacturing costs define what's available. Using a full-featured operating system just adds another layer to that complexity.

Two threads from the 2005 Linux Symposium lead back into memory matters. I'll start with the good news, then proceed to something confirming the growing awareness that system security is *really* hard.

NAND Versus NOR Versus RAM

About a year ago, I observed that the serial nature of NAND Flash memory precluded running code directly from it. As with all things technological, where there's economic pressure, there's a way to make the seemingly impossible happen. It's just a matter of trade-offs.

The Consumer Electronics Linux Forum BoFS at the Linux Symposium included a brief discussion of a Samsung NAND Flash research project that allows XIP (Execute-in-Place) access, so you can run program code directly from the chip. The paper outlining the technique illustrates just how weird embedded system memory can be. However, first you must understand why

Ed's an EE, PE, and author in Poughkeepsie, NY. Contact him at ed.nisley@ieee.org with "Dr Dobbs" in the subject to avoid spam filters.

everything you think you know about memory is wrong.

On the small end of the scale, a product's manufacturing cost can make or break it in the marketplace. That cost, in turn, depends on how many chips must be soldered to the board because the assembly cost can exceed the chip cost. Single-chip solutions reduce both board area and chip count, and may therefore reduce the overall cost even if they're more expensive than the components they replace.

Any tiny gizmo that handles music or video uses NAND Flash memory, which puts vast, cheap bulk storage behind a disk-like serial interface (ignoring, of course, those gizmos with miniature hard drives). That means a minimum of two chips: NAND Flash and a single-chip microcontroller with a CPU and on-chip program and data storage.

That's true for very small systems, but anything big enough for a real operating system requires a few megabytes of storage. Even with today's technology, that means four chips: NAND Flash, NOR Flash for the program, RAM, and the microprocessor. Plus, of course, whatever analog wiggery may be required to turn it into a phone-camera-PDA-web-pod.

In round numbers, a megabyte of NOR Flash costs five times more than NAND Flash and uses three times more power, so there's a mighty force aligned against that fourth chip. Storing less code, compressing it, and using other tricks may allow a smaller NOR Flash chip, but you want to eliminate that thing entirely.

Some of Samsung's current NAND Flash parts use their internal buffer RAM as a tiny XIP random-access memory that's automatically loaded with the contents of a specific NAND page before the CPU boots up. It's small because NAND Flash chips have only a dozen or so address lines that normally select a block within the chip, so there just isn't much address space available.

That XIP code copies the bulk of the program from NAND Flash to external

RAM, then jumps into the main routine. That's a comfortably familiar process occurring in hundreds of millions of larger systems (albeit using disk drives instead of NAND Flash), but in the embedded world it has a severe downside: The system must store two copies of the program code.

Again in round numbers, RAM costs at least 10 times more than NAND Flash and dissipates over five times more power. Program code in RAM tends to be essentially read-only after it's loaded, so you pay top dollar for a huge expanse of RAM that's used as ROM. Worst, the copy in NAND Flash is completely unused after booting.

The folks in charge of money don't like hearing that, of course.

NAND XIP

The entire recent history of CPU development revolves around the simple fact that memory-access time is much, much slower than CPU instruction cycle time. In fact, high-end CPUs now have three levels of cache in front of the main memory, each dedicated to anticipating the CPU's next request. Lower performance CPUs don't have quite the same bandwidth pressure (that's why they're lower performance, after all) and small embedded systems tend to get by with relatively poky CPUs.

The Samsung project combines the notion of NAND-as-disk with a liberal dash of Moore's Law to come up with a memory subsystem that the CPU sees as a reasonably high-speed, random-access ROM. They implemented a prototype with an FPGA and some static RAM surrounding a NAND Flash chip, but reducing all that to a single chip is just a matter of time and, perhaps, economics.

The NAND Flash chip is a read-only backing store for the much smaller SRAM cache, with the FPGA implementing the cache-control algorithms. The CPU sees the subsystem as a standard, random-access, read-only memory rather than a

serial-access NAND Flash chip. Reads from memory addresses currently in the cache proceed at SRAM speeds, while cache misses stall the system until the FPGA fetches the corresponding block from the NAND Flash.

The overall performance of any cached memory depends critically on the cache hit ratio: If it's below the mid to upper 90 percent range, you're sunk. A crude estimate says that when a cache hit costs 10 nanoseconds, and a miss costs 35 microseconds, a 99 percent hit ratio makes the average access time 360 nanoseconds. Ouch!

Unlike most cached systems, embedded applications tend to have fairly straightforward execution paths and a very limited number of programs. The Samsung designers analyzed a program trace to prioritize each address block by the number and frequency of accesses, then stored those priorities in the spare data area associated with each NAND Flash block. The FPGA controller can then decide which blocks are most likely to be required next, based on actual knowledge of the program's execution, and fetch new blocks into the lowest priority SRAM cache lines.

Their results for an MP3-player program show roughly 100 ns average access times for a 256-KB SRAM cache. It's not clear whether media data is also streaming through the cache, which could either increase or decrease the hit ratio depending on how the caching algorithm handles relentlessly sequential accesses.

In any event, the net result is random-access memory that's somewhat faster than NOR Flash and somewhat slower than SDRAM. The overall energy cost, measured in nanosecond-milliwatts, is roughly half that of NOR Flash, which may be the single most important parameter for mobile applications. However, the risk of protracted stalls on cache misses requires careful system design to ensure uninterrupted execution of those time-critical music decoding routines.

That's the sort of memory trade-off embedded-systems designers and programmers must put up with all the time. Beyond the usual requirement for correct functions, even the code's location can scuttle a project. Building a working system sometimes seems impossible, but the success of hand-held gizmos shows that it's merely difficult.

Stack Smashing

Back in the land of infinite virtual memory, even applications programmers could benefit from a little memory differentiation, as a good idea can go badly wrong with just the right design decisions. Here's a horror story from the world of big memory that extends down into the embedded world.

Back when Intel introduced the 8080 microprocessor, solid-state memory was still breathtakingly expensive. The 8080, implemented with a countably finite number of transistors, had an 8-bit ALU and 16-bit addresses. Filling that 64-KB address was more than most folks, myself included, could afford.

The 8086 microprocessor had a 16-bit ALU, but was more-or-less compatible with the 8080 at the assembly language level. In order to access more than 64 KB of memory, Intel introduced segment registers, which basically provided the upper 16 bits of a 20-bit address. Programmers became intimately familiar with the CPU's CS, DS, SS, and ES registers because large programs sprawled their code, data, and stack storage into different 64-KB segments.

Those segments were different in name only, as the hardware didn't enforce any access semantics. You could reach any type of data using any segment register with complete impunity. Needless to say, that led to some truly baffling bugs.

The 32-bit 80386 (the less said about the 80286 the better) enhanced the notion of segments to provide memory protection, while grafting paged virtual memory onto the side. You didn't have to use the VM paging hardware, but memory segmentation was mandatory.

Segment registers became pointers into tables of segment descriptors and, with the hardware now enforcing access semantics, a once-quirky architecture abruptly grew teeth. Segments contained only code or only data, selected by a single descriptor bit. Code segments could be execute-only or execute-read, while data segments could be read-only or read-write. Stack segments became a specialized data segment, with the ability to grow downward rather than upward.

Once upon a time, I actually wrote a bare-metal protected-mode program with full-throttle segmentation and can state from personal knowledge that figuring out the segmentation was somewhere beyond difficult. While my demo system worked, it became obvious that scaling it up wasn't in the cards.

I wasn't alone, as most OS designers opted for "flat model" segmentation in x86 systems. Although the hardware enforces the segment rules, there is nothing preventing you from defining all the segments to refer to the same chunk of memory. That turned addresses into 32-bit offsets from the common segment base, rather than unique values tied to a specific segment.

The fact that you could only write data in a Data segment, pop registers from a Stack segment, and execute code in the Code segment became completely irrelevant. If you could manage to write arbitrary data into the stack segment, you

could easily run it in the code segment without the hardware ever noticing.

And that, party people, explains why Windows is so vulnerable to stack-smashing attacks.

As it turns out, the Linux kernel has the same exposure. Windows just makes it easier for Other People's Code to gain access to the stack in the first place.

No Execute, No Cry?

The textbook heap and stack implementation puts the two at opposite ends of a common storage block with the heap growing up from the lowest address, and the stack growing down from the highest. All is well, so long as the two never meet.

The C language, lacking any inherent array index checking, makes buffer overruns trivially simple: Feeding a long string into a *strcpy()* function expecting, say, a username will do the trick. A sufficiently long string not only overflows the target buffer, but can extend all the way up into the stack storage area. In fact, if the string is stored on the stack (where automatic variables within C functions live), you don't even need the heap.

Strings in C, being just linear arrays of bytes, can contain nearly anything except the binary zero terminator that makes this attack possible. Attackers can therefore write both a small program and the register contents that pass control to it into the stack, ready for action when the abused *strcpy()* function executes a *RET* instruction.

The details of this process are tedious and depend on exactly what's going on in the attacked program and the OS. However, stack-smashing attack generation can be automated and, should the attacker get it wrong, the attacked program crashes and wipes out the evidence. If the attack happens in the kernel stack, it can take down the entire system.

Various Linux kernel patches have made stack-smashing attacks far more difficult, but its flat-memory layout means they can't be completely eliminated. AMD, with Intel tagging on behind, has added an NX (No-Execute) bit to the virtual page description in x86-64 mode, which obviously applies only to 64-bit programs, that does solve the problem.

All this assumes that nobody in their right mind would want to execute code from the stack. That turns out to be not quite correct, as it's often convenient to build trampolines on the stack, a subject quite outside the scope of this column. In any event, turning off the ability to run code from the stack can break innocent programs doing entirely reasonable things, so changing the OS underneath existing code may require recompiling some applications.

But the AMD NX bit should solve the problem for new code running in 64-bit mode, right? Nope, not quite.

Frankencode

Although 64-bit CPUs aren't commonly found in current embedded systems, let alone hand-held devices, Moore's Law tells us that it's only a matter of time. Let's suppose you're building a must-be-secure system, using both a CPU and an OS that can prevent code execution from the stack. Does a no-execute stack render buffer overflow attacks harmless, other than perhaps trashing the stack and crashing the program?

I found a paper by Sebastian Kraemer describing how a stack-smashing attack can execute arbitrary code, even on an x86-64 CPU with a properly NX-protected stack. The technique involves stitching together chunks of code from the ordinary library routines that are linked into essentially every compiled program.

Basically, an attacker can arrange the stack so that a *RET* instruction passes control to the last few instructions of a library function that pops the attacker's data into registers. Synthesizing system calls with the proper parameters requires finding the proper function epilogs and creating the appropriate stack contents to fill the reg-

isters. This is, of course, subject to automation.

The buffer overflow manipulates pure data on the (necessarily) writable stack, leaving code execution for already-existing functions in the code segment. The CPU's protection mechanisms have no idea anything is amiss.

The lesson to be drawn from all this resembles the lessons found in copy protection, digital-rights management, and Trusted Computing: The attackers are at least as smart as you are, they have better tools, and they will find a way around whatever technological measures you put in place. Declaring that hardware makes an attack impossible may be strictly correct, but finding an alternative vulnerability is merely difficult.

If you're building an embedded system that must be reliable and secure, getting the code working is just the first step. You must also control the environment around it, the access to it, and the data in it. Concentrating your attention on any one aspect, no matter how tempting, simply shifts the attacks to a weaker entry point.

Happy memories!

Reentry Checklist

The Linux Symposium proceedings are at <http://www.linuxsymposium.org/2005/>

and <http://www.linuxsymposium.org/proceedings.php>.

The Samsung paper on XIP NAND Flash is at <http://www.iccd-conference.org/proceedings/2003/20250474.pdf>. A summary of their existing parts, each sporting a tiny XIP boot block, is at http://www.samsung.com/Products/Semiconductor/Flash/OneNAND_TM/.

The Linux Kernel Mailing List discussion of Ingo Molnar's NX bit patch is at <http://kerneltrap.org/node/3240/>. Kraemer's explanation of the x86-64 NX exploit is at <http://www.suse.de/~krahmer/no-nx.pdf>, but the link for reference 3 should be <http://www.cs.rpi.edu/~hollingd/comporg/notes/overflow/overflow.pdf>. Find more on stack-smashing protection at <http://www.research.ibm.com/trl/projects/security/ssp/>. There is a description of GCC trampolines at http://www.delorie.com/gnu/docs/gcc/gccint_136.html.

Maybe you can't throw out your *Bartlett's Familiar Quotations* yet, but <http://www.brainyquote.com/> is in the running to replace it. "Everything You Know Is Wrong" is a vintage Firesign Theatre album I haven't heard in quite a while. Bob Marley's "No, Woman, No Cry" is a true classic.

DDJ

WE HAVE WHAT YOU NEED TO SEE BEFORE STARTING YOUR NEXT .NET PROJECT...

JUNKIE (JŪNG'KĒ)

n. *Slang pl.* junk·ies

One who has an insatiable interest or devotion



- Articles
- Blogs
- Up-to-date-news
- Whitepapers
- Forums
- And more!



www.dotnetjunkies.com

.NET developers come here to find out the latest news and discuss the latest issues in the .NET world. Expand your skills with .NET QuickStart Tutorials, post a question in the Forum, blog your latest thoughts on .NET or read the most recent articles. DotNetJunkies has what you need to see before starting your next .NET project.

The Community for DotNet Developers

www.dotnetjunkies.com