Views on software from Bryan Cantrill's deck chair

- Home
- About

The Observation Deck

Search Find

DTrace Safety

DTrace is a big piece of technology, and it can be easy to lose the principles in the details. But understanding these principles is key to understanding the design decisions that we have made — and to understanding the design decisions that we will make in the future. Of these principles, the most fundamental is the principle of safety: DTrace must not be able to accidentally induce system failure. It is our strict adherence to this principle that allows DTrace to be used with confidence on production systems — and it is its use on production systems that most fundamentally separates DTrace from what has come before it.

Of course, it's easy to say that this should be true, but what does the safety constraint mean? First and foremost, given that DTrace allows for dynamic instrumentation, this means that the user must not be allowed to instrument code and contexts that are unsafe to instrument. In *any* sufficiently dynamic instrumentation framework, such code and contexts exist (if nothing else, the framework itself cannot be instrumented without inducing recursion), and this must be dealt with architecturally to assure safety. We have designed DTrace such that **probes are provided by instrumentation providers that guarantee their safety**. That is, instead of the user picking some random point to instrument, instrumentation providers make available only the points that can be safely instrumented — and the user is restricted to selecting among these published probes. This puts the responsibility for instrumentation safety where it belongs: in the provider. The specific techniques that the providers use to assure safety are a bit too arcane to discuss here,[1] but suffice it to say that the providers are very conservative in their instrumentation.

This addresses one aspect of instrumentation safety — instrumenting wholly unsafe contexts — but it doesn't address the recursion issue, where the code required to process a probe (a context that we call *probe context*) ends up itself encountering an enabled probe. This kind of recursion can be dealt with in one of two ways: lazily (that is, the recursion can be detected when it happens, and processing of the probe that induced the recursion can be aborted) or proactively (the system can be designed such that recursion is impossible). For a myriad of reasons, we elected for the second approach: to make recursion architecturally impossible. We achieve this by mandating that **while in probe context, DTrace itself must not call into *any* facilities in the kernel at-large.** This means both implicit *and* explicit transfers of control into the kernel-at-large — so just as DTrace must avoid (for example) allocating memory in probe context, it must also avoid inducing scheduler activity by blocking.[2]

Once the fundamental safety issues of instrumentation are addressed, focus turns to the safety of user-specified actions and predicates. Very early in our thinking on DTrace, we knew that we wanted actions and predicates to be completely programmable, giving rise to a natural question: how are they executed? For us, the answer was so clear that it was almost unspoken: we knew that we needed to develop a virtual machine that could act as a target instruction set for a custom compiler. Why was this the clear choice? Because the alternative — to execute user-specified code natively in the kernel — is untenable from a safety perspective. Executing user-specified
code natively in the kernel is untenable for many reasons:

- Explicit stores to memory would have to be forbidden. To allow for user-defined variables (as we knew we wanted to do), one must clearly allow data to be stored. But if one executes natively, one has no way of differentiating a store to legal variable memory from a stray store to arbitrary kernel state. One is reduced to either forbidding stores completely (destroying a critical architectural component in the process), rewriting the binary to add checks around the store, or to emulating stores and manually checking the target address in the emulation code. The first option is unacceptable from a feature perspective, the second option is a non-trivial undertaking rife with new failure modes, and the third option — emulating stores — shouldn't be thought of as native execution but rather executing a a virtual machine that happens to match the underlying instruction set architecture.
- Loops would have to be dynamically detected. One cannot allow user-defined code to spin on the CPU in the kernel, so loops must be dynamically detected and halted. Static analysis might be tempting, but in a Turing-complete system such analysis will always be heuristic — one cannot solve the Halting Problem. While heuristics are fine when trying to achieve performance, they are not acceptable when correctness is the constraint. The problem must be therefore solved dynamically, and dynamic detection of loops isn't simple — one must detect loops using *any* control transfer mechanism, including procedure calls. As with stores, one is reduced to either forbidding calls and backwards branches, rewriting the binary to add dynamic checks before all control transfers, or emulating them and manually checking the target address in the emulation code. And again, emulating these instructions negates the putative advantages of native execution.
- Code would have to be very carefully validated for illegal instructions. For example, any instruction that operates on floating point state is (generally) illegal to execute in the kernel (Solaris — like most operating systems — doesn't save and restore the floating point registers on a kernel/user context switch; using the floating point registers in the kernel would corrupt user floating point state); floating point operations would have to be detected and code containing them rejected. There are many such examples (e.g. register-indirect transfers of control must be prohibited to prevent user-specified code from transferring control into the kernel at large, privileged instructions must be prohibited to prevent user-specified code from hijacking the operating system, etc. etc.), and detecting them isn't fail-safe: if one fails to detect so much as one of these cases, the entire system is vulnerable.
- Executing natively isn't portable. This might seem counterintuitive because executing natively seems to "automatically" work on any instruction set architecture that the operating system supports — it leverages the existing tool chain for compiling the user-specified code. But this leverage is Fools' Gold: as described above, the techniques to assure safety for native execution are profoundly specific to the instruction set — and any new instruction set would require completely new validation code. And again, this isn't fail-safe: so much as one slip-up in a new instruction set architecture means that the entire system is at risk on the new architecture.

We left these many drawbacks of native execution largely unspoken because the alternative — a purpose-built virtual machine for executing user-specified code — was so clearly the better choice. The virtual machine that we designed, the D Intermediate Format (DIF) virtual machine, has the following safety properties:

- It has no mechanism for storing to arbitrary addresses; the only store opcodes represent stores to specific kinds of user-defined variables. This solves two problems in a single design decision: it prohibits stores to arbitrary kernel memory, and it allows us to distinguish stores to different kinds of variables (global, thread-local, clause-local, etc.) from the virtual instruction opcode itself. This allows us to off-load intelligence about variable management from the instruction set and into the runtime where it belongs.
- It has no backwards branches, and supports only calls to defined runtime support routines — eliminating the possibility of user-defined loops altogether. This may seem unnecessarily severe (this makes loops an impossibility by architecture), but to us it was an acceptable tradeoff to achieve absolute safety.
- It is sufficiently simple to be easily (and rigorously) validated, as can be seen from the straightforward DIF object validation code.

- It is completely portable, allowing the validation and emulation code to be written and debugged once — accelerating bringup of DTrace on new platforms.

Just having an appropriately restricted virtual machine addressed many safety issues, but several niggling safety issues still had to be dealt with explicitly:

- Runtime errors like division-by-zero or misaligned loads. While a virtual machine doesn't solve these in and of itself, it makes them trivial to solve: the emulator simply refuses to perform such operations, aborting processing and indicating a runtime error.
- Loads from I/O space. In the Solaris kernel, devices have memory that can be mapped into the kernel's address space. Loads from these memory ranges can have side-effects; they must be prohibited. This is only slightly more complicated than dealing with divisions-by-zero; before performing a load, the emulator checks that the virtual address does not fall in a range reserved for memory mapped devices, aborting processing if it does.
- Loads from unmapped memory. Given that we wanted to allow user-defined code to chase pointers in the kernel, we knew that we had to deal with user-defined code attempting to load from an unmapped address. This can't be dealt with strictly in the emulator, as it would require probing kernel VM structures from probe context (which, if allowed, would prohibit instrumentation of the VM system). We dealt with this instead by modifying the kernel's page fault handler to check if a load has been DIF-directed before vectoring into the VM system to handle the fault. If the fault is DIF-directed, the kernel sets a bit indicating that the load has faulted, increments the trapping instruction pointer past the faulting load, and returns from the trap. The emulation code checks the faulted bit after emulating each instruction, aborting processing if it is set.

So DTrace is not safe by accident — DTrace is safe by deliberate design and by careful execution. DTrace's safety comes from a probe discovery process that assures safe instrumentation, a purpose-built virtual machine that assures safe execution, and a careful implementation that assures safe exception handling. Could a safe system for dynamic instrumentation be built with a different set of design decisions? Perhaps — but we believe that were such a system to be as safe, it would be either be so under-powered or so over-complicated as to invalidate those design decisions.

---

[1] The best place to see this provider-based safety is in the implementation of the FBT provider (x86, SPARC) and in the implementation of the pid provider (x86, SPARC).

[2] While it isn't a safety issue per se, this led us to two other important design decisions: probe context is lock-free (and almost always wait-free), and

interrupts are disabled in probe context.

Technorati tags: OpenSolaris Solaris DTrace

Posted on July 19, 2005 at 9:55 pm by bmc · Permalink
In: Solaris

« Previous post
Next post »

- **Recent Posts**
  - agghist, aggzoom and aggpack
  - Happy 10th Birthday, DTrace!
  - Serving up disaster porn with Manta
  - Manta: From revelation to product
  - A systems software double-header: Surge and GOTO
  - Post-revolutionary open source
  - DTrace in the zone
  - Debugging node.js memory leaks
  - Standing up SmartDataCenter
  - KVM on illumos
  - In defense of intrapreneurialism
  - When magic collides
  - Log/linear quantizations in DTrace
  - The DIRT on JSConf.eu and Surge
  - A physician's son
  - DTrace, node.js and the Robinson Projection
  - The liberation of OpenSolaris
  - The node.js demographic
  - OpenSolaris and the power to fork
  - Hello Joyent!
  - Good-bye, Sun
  - Turning the corner
  - John Birrell
  - Queue, CACM, and the rebirth of the ACM
  - Moore's Outlaws
- **Archives**
  - November 2013
  - September 2013
  - July 2013
  - June 2013
  - October 2012
  - August 2012
  - June 2012
  - May 2012

- September 2011
- August 2011
- July 2011
- March 2011
- February 2011
- October 2010
- September 2010
- August 2010
- July 2010
- March 2010
- November 2009
- May 2009
- February 2009
- January 2009
- December 2008
- November 2008
- September 2008
- July 2008
- June 2008
- May 2008
- March 2008
- December 2007
- November 2007
- October 2007
- September 2007
- August 2007
- July 2007
- June 2007
- May 2007
- March 2007
- October 2006
- August 2006
- May 2006
- December 2005
- November 2005
- September 2005
- August 2005
- July 2005
- June 2005
- May 2005
- April 2005
- March 2005
- February 2005
- January 2005
- December 2004
- November 2004
- October 2004
- September 2004
- August 2004

- July 2004
- June 2004