Uprobes: User-Space Probes

•Jim Keniston: jkenisto@us.ibm.com

April 9, 2009

Topics

- Overview
  - What and why?
  - Features
  - Uses
  - Tie-ins to kprobes, utrace, SystemTap
- History
- Status and ongoing work
- Issues, questions

Overview: What and Why?

- What:
  - kernel API, analogous to kprobes
  - breakpoints for user apps, handled in kernel
-
- struct uprobe *u = ...;
- ...
- u->pid = 1234;
- u->vaddr = 0x080484a8;
- u->handler = my_callback;
- result = register_uprobe(u);

Overview: What and Why?

- Why?
  - useful for dynamic, ad hoc instrumentation
  - handlers have system-wide view: kernel and apps
  - overcomes some limitations of ptrace:
    - Uprobes incurs lower overhead.
    - Uprobes useful for multithreaded apps.
    - “Who can probe whom” defined by uprobes client.

Overview: Features

- no need to modify probed process's source or binary
- per-process
    - All threads in process can (independently) hit probepoint.
- breakpoint probes (uprobes) and function-return probes (uretprobes)
- (Kernel) handler runs on probe hit.
    - Handler runs in context of probed task.
    - Handler can sleep – e.g., for kmalloc or paging.

Overview: Uses

- Typical use is via an ad hoc instrumentation module, a la kprobes.
  - SystemTap uses uprobes for user-space probing.
- Considering procfs and/or debugfs/trace interface.
- System-call interface possible:
  - new system call API
  - enhancements to ptrace
- Architectures supported: x86 (32- and 64-bit), powerpc, s390, ia64

Tie-ins to Kprobes

- Kprobes-like API: [un]register_u[ret]probe()
- Probed instruction single-stepped out of line (SSOL):
  - Leave breakpoint in place; execute copy of probed instruction...
  - ... to avoid probe misses in multithreaded apps.
  - Can be “boosted” to avoid 2nd (single-step) trap.
  - Single-stepping inline provided for jump-starting ports.
- Uprobes-specific complications:
  - “Out of line” instruction copies must reside in probed process's address space. Ditto the return-probe trampoline.
    - Solution: SSOL vma
  - Need to handle full instruction set (not just kernel instructions), guard against evil apps.

Tie-ins to Utrace and SystemTap

- Uprobes is a utrace client:
  - signal callback for breakpoint and single-step traps
  - clone, exec, and exit callbacks to track thread/process/image lifetime
  - quiesce callback for breakpoint insertion/removal
  - Uprobes patch modifies only Makefiles and Kconfigs.
- Uprobes is currently packaged with the SystemTap runtime.

History

- Spring 2006: Pre-utrace uprobes prototype skewered on LKML, soon discarded.
  - “Probe per-process, not per-executable.”
  - "Don't hook readpage(). Do COW via access_process_vm()."
  - "Just use ptrace."
- June 2006: Ptrace-based uprobes library prototyped, soon discarded.
- June 2006: Utrace first posted to LKML.
- Oct 2006: First working prototype (i386) of utrace-based uprobes.
- Dec 2006-Jan 2007: Uprobes += uretprobes, x86_64 port
-
-

History, cont.

- Feb-Mar 2007: SSOL-area implementation firmed up, with input from akpm, Dave Hansen, Roland McGrath.
- Spring 2007: More uprobes ports
- April 2007: ~~Uprobes posted to LKML~~ Utrace dropped from -mm tree
- Oct 2007: Uprobes tucked into SystemTap runtime.
- Summer 2008: SystemTap += DWARF-based probing of user apps
- Summer 2008: Utrace API revamped.
- Summer-Fall 2008: Uprobes adapts to new utrace and SystemTap-generated clients.
- Winter 2008-2009: Uprobes refactoring under way.

Status and Current Work

- SystemTap + uprobes working for x86, powerpc, s390. (ia64 = uprobes only)
- Refactoring uprobes into components for wider use:
  - instruction analysis (very architecture-specific)
  - user breakpoint assistance (ubp)
  - Redo SSOL vma management? Fold into ubp?
  - utrace helpers (attaching or quiescing all threads in a process)
- LKML review
- Feature requests/ideas:
  - bulk registration/unregistration
  - u[ret]probe objects reusable immediately after registration
  - uprobes booster: eliminate the single-step trap

Issues/Questions

- x86 instruction analysis: unify with kvm?
- SSOL vma: preallocation vs. lazy allocation; fixed vs. expandable; private vs. public slots
- support for 16-bit, VM86 apps?

Legal Statement