Kprobes und Systemtap

Tracing im Linux-Kernel

Hannes
Entropia e.V.
9.4.2006

Tracing

- Kernel tracing
  - printk()/make/install/reboot
  - kprobes/systemtap
  - ...
- Syscall tracing
  - ptrace()/mtrace()/trace.h
  - autrace/strace
- Cross-shared-libraries call tracing
  - ltrace

Kerneltracing

- (OpenSolaris: dtrace)
- printk()/make/install/reboot

- Linux <= 2.4
  - printk()/make/install/reboot
  - procfs/sysctl

- Linux >= 2.6
  - kprobes/systemtap
  - debugfs/configfs/sysfs/sysctl

kprobes

- Breakpoints im Codefluss
- Jede beliebige Adresse im Codefluss
- pre/post/failure-hooks
- Kopiert Instruktion und ersetzt mit int3
- Single-Stepping der pre_handler-Funktion
- post_handler
- Kallsyms
- Kbuild: CONFIG_KPROBES

jprobes

- Implementiert mit kprobes
- Probepunkte an Funktionsaufrufen
- Funktion wird ersetzt durch Funktion
  - Gleiche Signatur
- Einfacher Zugriff auf die Argumente der Funktion
- Kopieren des Stacks(MAX_STACK_SIZE)
- jprobe_return()/trap/restore
- Ein Jprobe pro Funktion

kretprobes

- Probepunkte an Funktions-returns
- Kprobe beim call
- Trampoline
- Krpobe beim Trampoline
- Return => Trampoline
- struct kretprobe_instance(preallocated)
- maxactive(CONFIG_PREEMT/NR_CPUS)
- nmissed

kallsyms

- Funktionsname => Speicheradresse[Modul]
- System.map(depricated)
- /proc/kallsyms:

```
c03bf4ff t packet_init
c03bf547 T _einittext
00000000 a i915_drv.c   [i915]
f04c6e98 t i915_exit    [i915]
f04c9240 d driver       [i915]
f0457000 t i915_init    [i915]
f04c09c0 ? __mod_author101      [i915]
```

Probleme/Limitations

- -EINVAL
  - tracing kprobe
- Inline functions!
- Kprobing kprobe-Handlers(kp->nmissed)
- Interrupts disabled
- Keine Memory-allocation
- Return addresses(backtraces)
- Keine Preemption

=> Kein yielding!

Exkurs: Kernel-Module

```
make -C /lib/modules/$(uname -r)/build M=$(pwd)

obj-m := foobar.o

#include <linux/module.h>

module_init()
module_exit()
```

Kprobes

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>

int register_kprobe(struct kprobe *kp);
int unregister_kprobe(struct kprobe *kp);

int pre_handler(struct kprobe *kp, struct pt_regs *regs);
void post_handler(struct kprobe *kp, struct pt_regs *regs,
                  unsigned long flags);
int fault_handler(struct kprobe *kp, struct pt_regs *regs, int trapnr);
```

struct kprobe {

```
    struct list_head list;
    unsigned int mod_refcounted;
    unsigned long nmissing;
    kprobe_opcode_t *addr;
    kprobe_pre_handler_t pre_handler;
    kprobe_post_handler_t post_handler;
    kprobe_fault_handler_t fault_handler;
    kprobe_break_handler_t break_handler;
    kprobe_opcode_t opcode;
    struct arch_specific_insn ainsn;
};
```

Jprobes

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>

int register_jprobe(struct jprobe *jp);

struct jprobe {
	struct kprobe kp;
	kprobe_opcode_t *entry;
};
```

Kretprobes

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>

struct kretprobe {
	struct kprobe kp;
	kretprobe_handler_t handler;
	int maxactive;
	int nmissed;
	struct hlist_head free_instances;
	struct hlist_head used_instances;
};
```

Kallsyms

```
#include <linux/kallsyms.h>

unsigned long kallsyms_lookup_name(const char *name);
```

CONFIG_DEBUG_INFO

- “Compile the kernel with debug info”
  - CONFIG_DEBUG_INFO
- objdump -d -l vmlinux

systemtap

- Skriptsprache
  - Awk-aehnlich
  - {% Embedded C %}
- Expandiert in Kprobe-Modul
- relayfs

- stap

Systemtap-Probes

- begin
- end
- kernel.function(“sys_blah”)
- syscall.fnord.return
- module(“foo”) .statement(0xabcdef)
- time.ms(200)
- kernel.function(“*@net/socket.c:7”).return
- ...

Systemtap Probe-Aliases

- probe syscallgroup.file_io =
  syscall.open, syscall.read, syscall.write,
  syscall.close
  { syscallgroup = “file_io” }

Systemtap Functions

- tid()
- pid()
- uid()
- execname()
- cpu()
- getttimeofday_s()
- get_cycles()
- pp()/probefunc()

Systemtap Beispiel

```
probe begin { printf(“begin\n”) }
probe end { printf(“end\n”) }
probe timer.ms(3000) { exit() }

probe syscall.fork,
        syscall.socket
    { printf(“%s called by %s\n”, pp(), execname()) }
```

Systemtap Language 1

- C/awk-aehnlich
  - if (...) else
  - while (...)
  - for (a, b, c) / foreach(var+/i in array)
  - continue, break, next, ;
- $Variablen
  - global
  - automatisch initialisiert und deklariert
  - arrays
  - %s, %d und string()

Systemtap Language 2

- function blah(fnord) { return fnord }
- <, >, ==, <=, >=
- String-Koncatenation mit .
- Pointer-Mangling: $pointer->foo->bar
  - struct *dev, struct *file, ...
  - user_string(), kernel_string() fuer char*

Arrays

- Fixed startup size
- Hashtables
- Muessen global deklariert sein

```
global array
probe syscall.fork { array[uid(),execname()] }
```

Aggregates

- statistics aggregates
- special operator: <<<
- aggr <<< varname
- @avg(aggr) := durchschnitt
- print(hist_linear(a,from,to,step)
- @count(aggr)
- print(hist_log(aggr)

Systemtap Safety

- Haendlerlaufzeit ist in der Zeit beschraenkt
- Kein Locking
  - falls gebraucht muss man embedded c benutzen
- Gefaerhlicher Code wird zur Laufzeit gecheckt

Embedded C

- -g for guru-mode

```
{%
#include <linux/rwsem.h>
%}

function testfunc:string (field:long)
%{
        if (down_read_trylock(& varname)) {
            // access to varname
        }
%}
```

Fedora-Infrastruktur

- RPM: debuginfo Pakete
- systemtap
- frysk
- gdb