Live Kernel Patching – status update

Jiri Kosina <jiri.kosina@suse.com>
SUSE Labs

Outline

- **Why?**
- History + current state
- Missing features / further development

Why live patching?

- **Huge cost of downtime:**
  - Hourly cost >$100K for 95% Enterprises – ITIC
  - $250K - $350K for a day in a worldwide manufacturing firm - TechTarget

- **The goal is clear: reduce planned downtime**

Why live patching?

Change management

Common tiers of change management

1. Incident response

"We are down, actively exploited …"

2. Emergency change

"We could go down, are vulnerable …"

3. Scheduled change

"Time is not critical, we keep safe"

Live Kernel Patching

Outline

- Why?
- **History + current state**
- Missing features / further development

(History: 1940's)

History: 2008 - 2014

- **2008: ksplice**
  - Originally university research opensource project
  - `stop_machine()` + stack inspection for asuring consistency
  - Automatic patch generation using binary object comparision
  - acquired by Oracle in 2011, source closed
  - Commercially deployed for Oracle linux distribution
- **2014: kPatch (Red Hat)**
  - Built on similar principle (stopping the kernel and inspecting the snapshot of all existing processess)
  - Automatic patch generation
  - Deployed as tech preview for Fedora
- **2014: kGraft (SUSE)**
  - Immediate patching with convergence to fully patched state ("lazy migration")
    - Consistency model: kernel/userspace boundary crossing considered a checkpoint
      - Issue: Long sleeping processess, kthreads
  - Manual patch creation with help of toolchain
  - Commercially deployed; ~80 patches distributed up to today

History: 2008 - 2014

- **Checkpoint/restart based solution (CRIU)**
    - Completely different aproach in principle
    - Checpoint userspace → kexec new kernel → restore userspace
    - + Allows to exchange complete kernel, no matter the nature of the changes
    - - Hardware reinitialized
    - - Not "immediate" at all

Lazy migration
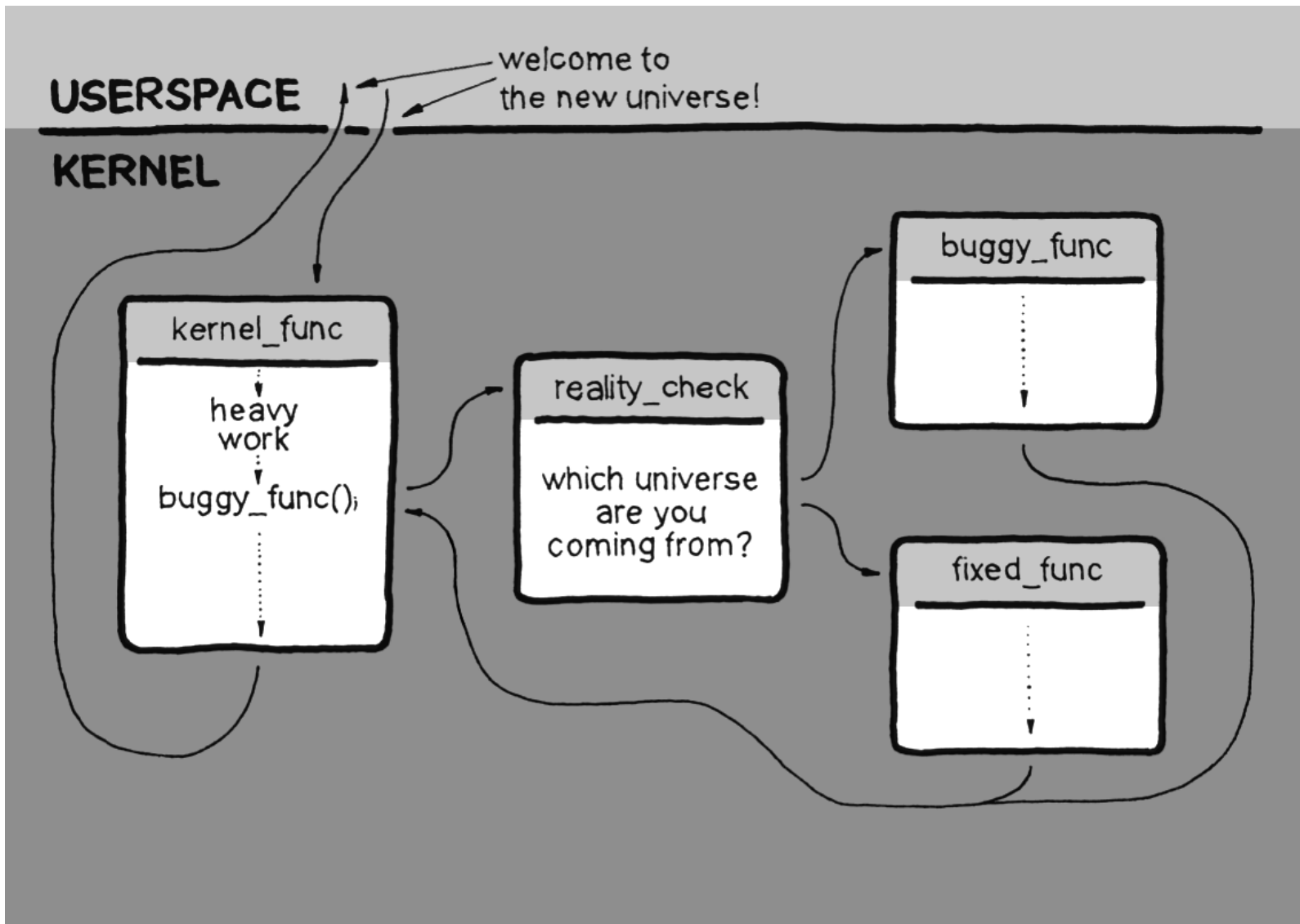
History: 2015

- Live patching session at LPC in Düsseldorf
- Technical presentations of competing projects and discussing future direction
- Mutual agreement on attempting to merge "just one unified thing" upstream
- The agreed plan:
  - Start with a very minimalistic base and have that merged
  - Start porting (and combining) competing solutions on top of it, cherry-picking good ideas step-by-step
  - Base (just function redirection + API) merged into Linus' tree in Feb 2015

History: 2015 - now

- New features being gradually added to `CONFIG_LIVEPATCH`
  - Combined (hybrid) consistency model of kGraft + kPatch
    - Lazy migration by default, stack examination for long-sleeping processess/kthreads
  - Extending the arch support beyond x86 (s390, ppc64 (arm64))
  - objtool + ORC unwinder implemented by Josh Poimboeuf (reliance on FP-based stack checking has performance implications, DWARF unavailable)

Patch Generation

- patches created almost entirely by hand
  - (for upstream `CONFIG_LIVEPATCH` at least)
- The source of the patch is single C file
  - Easy to review, easy to maintain in a VCS like git
- Add fixed functions
- Create a list of functions to be replaced
- Issue a call to kernel livepatching API
- Compile
- Insert as a .ko module

Patch Generation

```
--- a/fs/proc/cmdline.c
+++ b/fs/proc/cmdline.c
@@ -5,7 +5,7 @@

 static int cmdline_proc_show(struct seq_file *m, void *v)
 {
-        seq_printf(m, "%s\n", saved_command_line);
+        seq_printf(m, "cmdline_proc_show() has been patched\n", saved_command_line);
         return 0;
 }
```

Patch Generation

```c
static int livepatch_cmdline_proc_show(struct seq_file *m, void *v)
{
        seq_printf(m, "%s\n", "this has been live patched");
        return 0;
}
static struct klp_func funcs[] = {
        {
                .old_name = "cmdline_proc_show",
                .new_func = livepatch_cmdline_proc_show,
        }, { }
};
static struct klp_object objs[] = {
        {
                /* name being NULL means vmlinux */
                .funcs = funcs,
        }, { }
};
static struct klp_patch patch = {
        .mod = THIS_MODULE,
        .objs = objs,
};
static int livepatch_init(void)
{
        int ret;
        ret = klp_register_patch(&patch);
        if (ret)
                return ret;
        ret = klp_enable_patch(&patch);
        if (ret) {
                WARN_ON(klp_unregister_patch(&patch));
                return ret;
        }
        return 0;
}
static void livepatch_exit(void)
{
        WARN_ON(klp_unregister_patch(&patch));
}
```

Outline

- Why?
- History + current state
- **Missing features / further development**

Limitations and missing features

- **Inability to deal with data structure / semantics changes**
  - State format transformation
  - Needed for more complex fixes
  - *Lazy state transformation?*
    - New functions able to work with both old and new data format
    - After code *lazy migration* is complete, start transforming data structures on access
  - *Shadow variables?*
    - Associating a new field to the existing structure (can be used by patched callers)
    - Currently being implemented by Joe Lawrence
  - State also contains exclusive access mechanisms
    - Spinlocks, mutexes
    - Converting those without a deadlock is an unsolved problem
  - *Patch callbacks?*
    - Allows for arbitrary "fixup" during patch application phases
    - [too] powerful, has to be used with care

Limitations and missing features

- **Model/consistency verification**
  - Is the change/fix still within the consistency model?
    - (other traps: static variables in a func scope, patching `schedule()`, ...)
  - Currently done by human reasoning – error prone and time consuming
    - "patch author guide" with best practices:
      https://github.com/dynup/kpatch/blob/master/doc/patch-author-guide.md
- **Patch creation tooling**
  - Patches affected by combinatorial explosion (function inlining, ABI breakage by compiler (`-fipa-ra`))
  - Linking (/ patching relocations) to avoid excessive usage of kallsyms lookup
  - manual/kbuild/asmtool, klp-convert, kpatch-build
  - A lot of things could be detected automatically
- **Kprobes**
  - trarnsferring a kprobe to a new implementation of the function is non-trivial
- **Extending arch coverage**
  - `FTRACE_WITH_REGS`
  - objtool + reliable stack unwinding
  - Small livepatching glue code

Limitations and missing features

- **Inability to patch hand-written ASM**

  - No `fentry`, ftrace not aware

  - Should be easy in principle

- **Userspace patching**

  - Different problem in principle

    - harder to define a "checkpoint" for consistency

    - Kernel is very gcc-centric, userspace not so much

    - Initial efforts:

      - https://github.com/joe-lawrence/linux-inject

      - https://github.com/virtuozzo/nsb

SUSE

We adapt. You succeed.