Live Kernel Patching

Miroslav Beneš
SUSE Labs, mbenes@suse.cz
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”
Barcelona Supercomputing Centre
Mare Nostrum supercomputer

- 50k Sandy Bridge cores
- The most beautiful supercomputer in the world
- Terabytes of data
- Reboot?
5m telescope with adaptive optics on Mount Palomar

Avoid atmospheric blurring in Real Time

Control 3888 segments of a deformable mirror with a latency

<250 μs

Reboot?
SAP HANA
In-memory database and analytics engine

- 4-16 TB of RAM
- All operations done in memory
- Disk used for journalling
- Active-Passive HA
- Failover measured in seconds
- Reboot?

HP DL980 w/ 12 TB RAM
1943: Manhattan project – punchcards

- IBM punchcard automatic calculators were used to crunch the numbers
- A month before the Trinity nuclear device test, the question was: “What will the yield be, how much energy will be released?”
- The calculation would normally take three months to complete – recalculating any batches with errors
- Multiple colored punch cards introduced to fix errors in calculations while the calculator was running

Trinity test site, 16ms after initiation
2003: Windows HotPatching (Microsoft)

- Windows Server 2003 SP1
- Stops kernel execution for activeness check
  - Schedule procedures on all but current CPUs and keep them busy
- Uses short jumps patched into functions for redirection
  - The second redirection (jump) to a new function
- Removed in Windows 8 and later versions
2008: Ksplice (MIT, Oracle)

- First to patch *Linux kernel*
- Stops kernel execution for activeness check
  - Restarts and tries again later when active
- Uses jumps patched into functions for redirection

**Commercially deployed**

*Kernel patching*

*Activeness safety*

*Binary patching*
2014: kGraft (SUSE)

- *Linux kernel* patching
- *Immediate* patching with *lazy migration*
- Uses jumps patched into functions for redirection
- *Commercially deployed*

*Commercially deployed*

*Linux kernel*

*Immediate*

*Lazy migration*
2014: kpatch (Red Hat)

- *Linux kernel* patching
- Stops kernel execution for activeness check
  - Restarts and tries again later when active
- Uses jumps patched into functions for redirection
- *Commercially deployed – technical preview*

*Commercially deployed*

*Linux kernel*

*Activeness safety*

*Binary patching*
kGraft goals

- **Applying limited scope fixes to the Linux kernel**
  - Security, stability and corruption fixes
- **Require *minimal changes* to the source code**
  - No changes outside the kGraft engine itself
- **Have no runtime *performance impact***
  - Full speed of execution
- **No interruption of applications while patching**
  - Full speed of execution
- **Allow *full review* of patch source code**
  - For accountability and security purposes
Call Redirection
How Does It Work

- Use of ftrace framework
  - gcc -pg is used to generate calls to \texttt{\_fentry\_()} at the beginning of every function
  - ftrace replaces each of these calls with \texttt{NOP} during boot, removing runtime overhead (when CONFIG_DYNAMIC_FTRACE is set)
  - When a tracer registers with ftrace, the \texttt{NOP} is runtime patched to a \texttt{CALL} again
  - kGraft uses a tracer, too, but then asks ftrace to change the return address to the new function
  - And that's it, call is redirected

- x86\_64 from now on
  - Although s390x, powerpc and arm64 are similar
Call redirection
ftrace: SMP-safe code modification
Call redirection
ftrace: SMP-safe code modification

static int cmdline_proc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "%s
", saved_command_line);
    return 0;
}
Call redirection
ftrace: SMP-safe code modification

<cmdline_proc_show>:
\[\begin{array}{ll}
e8 & 4b 68 39 00 \quad \text{callq} \quad ffffffff8160d8d0 \quad \_\_fentry\_\_ \\
48 & 8b 15 7c 3f ef 00 \quad \text{mov} \quad 0xef3f7c(%rip),%rdx \quad \# \quad \text{<saved_command_line>}
31 & c0 \quad \text{xor} \quad %eax,%eax \\
48 & c7 c6 a3 d7 a4 81 \quad \text{mov} \quad $0xffffffff81a4d7a3,%rsi \\
e8 & e6 1d fb ff \quad \text{callq} \quad ffffffff81228e80 \quad \text{<seq_printf>}
31 & c0 \quad \text{xor} \quad %eax,%eax \\
c3 \quad \text{retq} \\
0f & 1f 00 \quad \text{nop1} \quad (%rax)
\end{array}\]
Call redirection
ftrace: SMP-safe code modification

<cmdline_proc_show>:

e8 4b 68 39 00 callq ffffffff8160d8d0 <__fentry__>
48 8b 15 7c 3f ef 00 mov 0xef3f7c(%rip),%rdx # <saved_command_line>
31 c0 xor %eax,%eax
48 c7 c6 a3 d7 a4 81 mov $0xffffffff81a4d7a3,%rsi
e8 e6 1d fb ff callq ffffffff81228e80 <seq_printf>
31 c0 xor %eax,%eax
31 c0 retq
0f 1f 00 nop1 (%rax)

<cmdline_proc_show>:

0f 1f 44 00 00 nop1 0x0(%rax,%rax,1)
48 8b 15 7c 3f ef 00 mov 0xef3f7c(%rip),%rdx # <saved_command_line>
Call redirection
ftrace: SMP-safe code modification

<cmdline_proc_show>:
```
  e8 4b 68 39 00     callq  ffffffff8160d8d0  <__fentry__>
  48 8b 15 7c 3f ef 00    mov   0xef3f7c(%rip),%rdx   # <saved_command_line>
  31 c0                  xor    %eax,%eax
  48 c7 c6 a3 d7 a4 81    mov   $0xffffffff81a4d7a3,%rsi
  e8 e6 1d fb ff          callq  ffffffff81228e80  <seq_printf>
  31 c0                  xor    %eax,%eax
  c3                      retq
  0f 1f 00               nopl   (%rax)
```

<cmdline_proc_show>:
```
  0f 1f 44 00 00     nopl   0x0(%rax,%rax,1)
  48 8b 15 7c 3f ef 00    mov   0xef3f7c(%rip),%rdx   # <saved_command_line>
```

<cmdline_proc_show>:
```
  e8 7b 3f e5 1e     callq  0xfffffffffa00cb000    # ftrace handler
  48 8b 15 7c 3f ef 00    mov   0xef3f7c(%rip),%rdx   # <saved_command_line>
```
Call Redirection
ftrace: return address modification mechanism
#include <linux/module.h>
#include <linux/kgraft.h>
#include <linux/seq_file.h>

static int kgr_cmdline_proc_show(struct seq_file *m, void *v) {
    seq_printf(m, "%s\n", " this is livepatched");
    return 0;
}

static struct kgr_patch patch = {
    .name = "patch",
    .owner = THIS_MODULE,
    .patches = {
        KGR_PATCH(cmdline_proc_show, kgr_cmdline_proc_show),
        KGR_PATCH_END
    }
};

static int __init patch_init(void) {  return kgr_patch_kernel(&patch);  }
static void __exit patch_cleanup(void) { kgr_patch_remove(&patch);  }

module_init(patch_init);
module_exit(patch_cleanup);

MODULE_LICENSE("GPL");
Patch Generation
Manual Approach

- kGraft offers a way to create patches entirely by hand
- The source of the patch is then a 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
- Call kGraft: kgr_patch_kernel();
- Compile
- Insert as a .ko module
- Done
kgr_patch_kernel()

- Two ftrace handlers in kGraft
  - Slow and fast (because of a consistency model)
- State machine

- It boils down to call register_ftrace_function() and ftrace_set_filter_ip()

- Module functions are dealt with in module notifiers
Call Redirection
The Final Hurdle

• Changing a single function is easy
  • Since ftrace patches at runtime, you just flip the switch

• What if a patch contains multiple functions that depend on each other?
  • Number of arguments changes
  • Types of arguments change
  • Return type change
  • Or semantics change

• We need a consistency model
  • Lazy migration enforcing function type safety
kGraft Consistency Model
Keeping Threads Intact

- We want to avoid calling a new function from old and vice versa
- Execution threads in kernel are of four types
  - Interrupts (initiated by hardware, non-nesting)
  - User threads (enter kernel through SYSCALL)
  - Kernel threads (infinite sleeping loops in kernel)
  - Idle tasks (active when there is nothing else to do)
- We want to make sure a thread calls either all old functions or all new
- And we can migrate them one by one to 'new' as they enter/exit execution
- No stopping for anybody
kGraft Consistency Model

- welcome to the new universe!
- kernel_func
  - heavy work
  - buggy_func()
- reality_check
  - which universe are you coming from?
- buggy_func
- fixed_func
kGraft Consistency Model

• Per-thread flag
  • TIF_KGR_IN_PROGRESS
  • Mark all tasks in a system at the beginning and wait for them to be migrated to a new universe
  • Finalize
kGraft Consistency Model

static notrace void kgr_stub_slow(unsigned long ip, unsigned long parent_ip,
struct ftrace_ops *ops, struct pt_regs *regs)
{
  struct kgr_patch_fun *p = ops->private;
  bool go_new;

  if (in_interrupt())
    go_new = *this_cpu_ptr(kgr_irq_use_new);
else if (test_bit(0, kgr_immutable)) {
    kgr_mark_task_in_progress(current);
    go_new = false;
} else {
    rmb(); /* test_bit before kgr_mark_task_in_progress */
    go_new = !kgr_task_in_progress(current);
}

if (p->state == KGR_PATCH_REVERT_SLOW)
  go_new = !go_new;

/* Redirect the function unless we continue with the original one. */
if (go_new)
  kgr_set_regs_ip(regs, (unsigned long)p->new_fun);
else if (p->loc_old != p->loc_name)
  kgr_set_regs_ip(regs, p->loc_old);
Kraft Consistency Model

static notrace void kgr_stub_fast(unsigned long ip,
    unsigned long parent_ip, struct ftrace_ops *ops,
    struct pt_regs *regs)
{
    struct kgr_patch_fun *p = ops->private;
    kgr_set_regs_ip(regs, (unsigned long)p->new_fun);
}

static inline void kgr_set_regs_ip(struct pt_regs *regs,
    unsigned long ip)
{
    regs->ip = ip;
}
kGraft Consistency Model
Complications

• How about eternal sleepers?
  • Like `getty` on a console 10
  • They'll never exit the kernel
  • They'll never be migrated to 'new'
  • They'll block completion of the patching process forever

• Wake them up!
  • Sending a `fake signal` (SIGPENDING flag, but no signal in a queue)
  • The signal exits the syscall and transparently restarts it

• And kthreads?
  • They cannot exit the kernel ever
  • Annotate them in a safe place and wake them up
Ksplice Consistency Model
Making a Clean Cut

- Ksplice uses *Activeness safety*
- First `stop_kernel();`
  - That stops all CPUs completely, including all applications
- Then, check all stacks, whether any thread is stopped within a patched function
- If yes, resume kernel and try again later
  - And hope it'll be better next time
- If not, flip the switch on all functions and resume the kernel
- The system may be stopped for 10-40ms typical
Kernel Live Patching in Linux upstream

- Result of a discussion between Red Hat and SUSE at Linux Plumbers Conference 2014 in Dusseldorf
- Basic infrastructure merged in 2015
  - Neither kGraft, nor kpatch
  - Patch format abstraction and function redirection based on ftrace
  - x86_64, s390x and powerpc architectures supported
    - arm and arm64 in development
- **Main missing feature? Consistency model.**
  - Reviewed and present in linux-next tree
  - Should go to 4.12
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)); }

module_init(livepatch_init);
module_exit(livepatch_exit);
MODULE_LICENSE("GPL");
MODULE_INFO(livepatch, "Y");
#include <linux/module.h>
#include <linux/kgraft.h>
#include <linux/seq_file.h>

static int kgr_cmdline_proc_show(struct seq_file *m, void *v) {
    seq_printf(m, "%s
", " this is livepatched");
    return 0;
}

static struct kgr_patch patch = {
    .name = "patch",
    .owner = THIS_MODULE,
    .patches = {
        KGR_PATCH(cmdline_proc_show, kgr_cmdline_proc_show),
        KGR_PATCH_END
    }
};

static int __init patch_init(void) {  return kgr_patch_kernel(&patch);  }
static void __exit patch_cleanup(void) { kgr_patch_remove(&patch);  }

module_init(patch_init);
module_exit(patch_cleanup);

MODULE_LICENSE("GPL");
Kernel Live Patching in Linux upstream

- Hybrid of kGraft and kpatch consistency models
- Per-thread migration, but scope limited to a set of patched functions
- Based on a stack checking
  - To ensure that a task does not sleep in a to-be-patched function
- Heated discussion when proposed
  - Stacks and their dumps are unreliable
- Josh Poimboeuf then proposed objtool
  - It analyzes every .o file and ensures the validity of its stack metadata (currently frame pointer usage)
  - Plans to add CFI (Call Frame Information, DWARF) validation for C files and CFI generation for assembly files
- The second proposal sidetracked as well
  - Josh rewrote the kernel stack unwinder
  - Plans are to make it DWARF aware
static void notrace klp_ftrace_handler(unsigned long ip, unsigned long parent_ip, struct ftrace_ops *fops, struct pt_regs *regs)
{
    struct klp_ops *ops;
    struct klp_func *func;
    int patch_state;

    ops = container_of(fops, struct klp_ops, fops);
    rcu_read_lock();
    func = list_first_or_null_rcu(&ops->func_stack, struct klp_func, stack_node);

    if (WARN_ON_ONCE(!func))
        goto unlock;
    smp_rmb();

    if (unlikely(func->transition)) {
        smp_rmb();
        patch_state = current->patch_state;
        WARN_ON_ONCE(patch_state == KLP_UNDEFINED);

        if (patch_state == KLP_UNPATCHED) {
            func = list_entry_rcu(func->stack_node.next, struct klp_func, stack_node);
            if (&func->stack_node == &ops->func_stack)
                goto unlock;
        }
    }
    klp_arch_set_pc(regs, (unsigned long)func->new_func);

    unlock:
    rcu_read_unlock();
}
Limitations and missing features

- Inability to deal with data structure 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
  - State also contains exclusive access mechanisms
    - Spinlocks, mutexes
    - Converting those without a deadlock is an unsolved problem

- **Model/consistency verification**
  - Is the change/fix still within the consistency model?
  - Currently done by human reasoning – error prone and time consuming
Limitations and missing features

- **Kernel threads**
  - Stack checking in upstream vs. manual annotation in kGraft
  - Stack checking does not always work
  - Manual annotation is horrible
  - There is an effort to move kthreads to kworker API with a well-defined checkpoint

- **schedule() and switch_to() macro**
  - `schedule()` is a tricky function. It returns with a different stack
  - If old and new `schedule()` are compiled differently by GCC (different registers are allocated, stack is different due to local variables, ...) → corruption and crash upon return
  - Possible solution – ensure the return to a proper version of `schedule()`
    - Make the value of RIP register part of the saved and restored context
    - This happened in 4.9
Patch Generation Pitfalls
Inlining

- Inlining is when a compiler decides that it's worth copying a function as a whole instead of calling it

- kGraft uses a special function of the GCC compiler to emit a list of functions where a bug has been inlined
  - `-fdump-ipa-clones` option
Patch Generation Pitfalls

Static symbols

- Static or unexported symbols are common
- But they may be used from the patched function
- The kernel keeps a list of all symbols: kallsyms

```c
int patched_fn(void)
{
    kgr_orig_static_fn();
}
```

```c
static int __init patch_init(void)
{
    kgr_orig_static_fn =
        (static_fn_proto)kallsyms_lookup_name("static_fn");
    if (!kgr_orig_static_fn) {
        pr_err("kgr: function %s not resolved\n", "static_fn");
        return -ENOENT;
    }
    ...
}
```

- A proper solution is to use ELF relocations – work in progress
Patch Generation Pitfalls
IPA-SRA

• Compiler optimization, enabled with -O2 and higher
  • Inter-procedural scalar replacement of aggregates
• Gives a significant performance boost
• But also a disaster for patching
  • Can modify CALL at the end of a function into JMP if it's the last statement of a function
  • Can transform arguments passed by reference to arguments passed by value if the value is never changed
  • Can create variants of a function with fewer arguments if the removed argument doesn't have any impact on what the function does in a specific case
• Again, GCC and DWARF to the rescue (and more work when creating the patch)
Patch Generation Pitfalls
GCC optimizations

- **Partial inlining**
  - Inline just a part of a function

- **IPA-CP**
  - Interprocedural constant propagation
  - Optimizes functions if values passed to them are constants
  - Even several clones of a function are possible if a set of values is limited

- **IPA-PURE-CONST**
  - GCC can eliminate calls to pure const functions, memory accesses can be removed etc.
  - What if a function is no longer pure const with a fix applied?

- **IPA-ICF**
  - Identical code folding for functions and read-only variables
  - Replaces a function with an equivalent one
Patch Generation Pitfalls

Dead Code Elimination

- A change of a global variable and GCC code elimination

```c
int global;
int foo(void) { return 22; }
int main(void)
{
    int i;
    global = 1;
    i = foo();
    if (global == 2) return 11;
    return 33;
}
```

- Generally always when a function somehow affects another one
Patch Generation Pitfalls
IPA-RA

• Use caller save registers for allocation if those registers are not used by any called function. In that case it is not necessary to save and restore them around calls.
• Currently disabled thanks to `-pg` option, but this can change easily
• Reportedly dubious gain for the kernel
• Only a small fraction of functions affected (~0.5 %)