Live Patching

Miroslav Beneš
SUSE Labs, mbenes@suse.cz
Live (Kernel/User space) Patching

• What is it?
  • Application of kernel patches without stopping/rebooting the system
  • Similarly applies to the user space

• Why?
  • Convenience/Cost – Huge cost of downtime, hard to schedule
  • Availability
  • Compliance

• Clear goal – reduce planned or unplanned downtime
Barcelona Supercomputing Center

- 165k Skylake cores
- Terabytes of data
- 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?
Goals and Principles

- Applying limited scope fixes to the Linux kernel
  - Security, stability and corruption fixes
- Require minimal changes to the source code
  - Limited changes outside of the infrastructure 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
History

- **Windows HotPatching (2003 – Microsoft)**
  - Stops kernel execution for activeness check (busy loop)
  - A function redirection using a short jump before a function prologue

- **Ksplice (2008 – MIT, Oracle)**
  - First to patch the Linux kernel
  - Stops kernel execution for activeness check
    - Restarts and tries again later when active
  - Uses jumps patched into functions for redirection

- **kpatch (2014 – RedHat)**
  - Similar to Ksplice
  - Binary patching

- **kGraft (2014 – SUSE)**
  - Immediate patching with lazy migration
  - Per-thread consistency model
Kernel Live Patching in Linux Upstream

- Result of a discussion between Red Hat and SUSE at Linux Plumbers Conference 2014 in Dusseldorf
- Basic infrastructure
  - Neither kGraft, nor kpatch
  - Patch format abstraction and function redirection based on ftrace
  - x86_64, s390x and powerpc architectures supported
    - arm64 in development
- Merged to 4.0 in 2015
Call Redirection

• **x86_64 from now on**
  • Although s390x, powerpc and arm64 are similar

• **Use of ftrace framework**
  • gcc -pg is used to generate calls to `_fentry_()` at the beginning of every function
  • ftrace replaces each of these calls with NOP during boot, removing runtime overhead (when CONFIG_DYNAMIC_FTRACE is set)
  • When a tracer registers with ftrace, the NOP is runtime patched to a CALL again
  • livepatch uses a tracer, too, but then asks ftrace to change the return address to the new function
  • And that's it, call is redirected
Call Redirection
Simple Sample

static int cmdline_proc_show(struct seq_file *m, void *v) {
    seq_printf(m, "%s\n", saved_command_line);
    return 0;
}
Call Redirection

<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
31 c0                   xor    %eax,%eax
e8 e6 1d fb ff          callq  ffffffff81228e80 <seq_printf>
31 c0                   xor    %eax,%eax
retq
```
Call Redirection

<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
3c                      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>
Call Redirection

<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
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)
{
    return klp_enable_patch(&patch);
}
static void livepatch_exit(void) { }

module_init(livepatch_init);
module_exit(livepatch_exit);
MODULE_LICENSE("GPL");
MODULE_INFO(livepatch, "Y");
Patch Generation – Semi-automatic Approach

- Patches were originally created entirely by hand
  - Create a list of functions to be replaced
  - Copy the source code, fix it
  - Code closure to make it compile
  - Call livepatch: klp_enable_patch()
  - Compile, insert as .ko module, done
- The source of the patch is then a single C file
  - Easy to review, easy to maintain in a VCS like git
- klp-ccp
  - https://github.com/SUSE/klp-ccp
  - Prepares a C file almost automatically
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
kGraft Consistency Model

- Avoid calling a new function from old and vice versa
- Make sure a thread calls either all old functions or all new
- Migrate them one by one to 'new' as they enter/exit execution
- No stopping for anybody
kGraft Consistency Model

Welcome to the new universe!
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

• How about eternal sleepers?
  • Like `getty` on a console
  • 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
kpatch Consistency Model

- 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
Livepatch Hybrid Consistency Model

- Hybrid of kGraft and kpatch consistency models
- Based on a stack checking
- 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 (frame pointer usage at the time of proposal)
- The second proposal sidetracked as well
  - Josh rewrote the kernel stack unwinder
- Merged to 4.12
  - The pure kGraft is not present in any supported code stream of SUSE Linux Enterprise Server
Livepatch Hybrid Consistency Model

- Per-thread migration, but scope limited to a set of patched functions
- What entity the execution must be outside of to be able to make the switch
  - LEAVE_{FUNCTION, PATCHED_SET, KERNEL}
- What entity the switch happens for
  - SWITCH_{FUNCTION, THREAD, KERNEL}
- kGraft is LEAVE_KERNEL and SWITCH_THREAD
- kpatch is LEAVE_PATCHED_SET and SWITCH_KERNEL
- Hybrid consistency model is LEAVE_PATCHED_SET and SWITCH_THREAD
  - Reliable, fast-converging, no annotation of kernel threads, no failure with frequent sleepers
Livepatch Hybrid Consistency Model

- **Stack checking**
  - To ensure that a task does not sleep in a to-be-patched function (set of to-be-patched functions)
- **Per-thread flag**
  - Similar to kGraft
  - Threads are still migrated on the user space/kernel space boundary
- **Allows for faster migration to a new universe**
Livepatch Hybrid Consistency Model

- Slightly different consistency model leads to slight differences during a live patch development
  - Threads are switched earlier (when they leave patched set)
  - It could matter in case of complex caller–callee changes

- Eternal sleepers
  - Not a problem as long as they do not sleep in a patched function (set of patched functions)
  - We have the fake signal for the rest

- Kthreads are the same
Livepatch Hybrid Consistency Model

- Reliable stacks require frame pointers (FPs)
  - There is a performance penalty with FPs enabled
- Plans to add Call Frame Information (CFI, DWARF) validation for C files, CFI generation for assembly files and introduction of DWARF-aware unwinder were not welcome
- ORC unwinder
  - Tailored info generated by objtool
  - Unwinder is simple – no complicated state machine
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);
  preempt_disable_notrace();
  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;
    }
  }
  if (func->nop)
    goto unlock;
  klp_arch_set_pc(regs, (unsigned long)func->new_func);
unlock:
  rcu_read_unlock();
}
Additional Features

- **Callbacks**
  - klp_object (un)patching notification mechanism
  - Modification of global data and registration of newly available services/handlers

- **Shadow variables**
  - Way to deal with data structure/semantics changes
  - Associating a new field to the existing structure

- **Selftests and samples**
Atomic Replace

- Livepatch allows multiple patches on a (function) stack
- Maintenance nightmare if there is a dependency between patches
  - Several different fixes of a function

- Cumulative patches and atomic replace
  - All older patches removed after the transition
  - Special nop functions which redirect to the original functions
Limitations and Missing Features

• **Non-exported symbols**
  • kallsyms trick
  • Relocations
  • klp-convert

• **Patch creation tool**
  • Currently semi-automatic, tools to help
  • kpatch-build
  • Source-based approach in upstream
Limitations and Missing Features

- **GCC optimizations**
  - Inlining
    - A bug propagation
  - Interprocedural optimizations

- GCC to help
  - -fdump-ipa-clones
  - -flive-patching
Userspace Live Patching

- **Libpulp**
  - https://github.com/SUSE/libpulp
  - Library for live patching other user space libraries
  - Ptrace-based

- **Consistency model**
  - Similar to the original kGraft approach
  - Per-thread
  - Migration on the application-library boundary

- In development but definitely coming

- Youtube recording from SUSE Labs Conference 2020