Kernel Live Patching at Scale

Song Liu, David Vernet, Rik Van Riel

Linux Plumbers Conference 2022

Meta


Meta

KLP @ Scale

- Existing options for applying kernel fixes
- Introducing KLP (kernel live patching)
- Livepatch ecosystem
- Using KLP at hyper-scale
- Challenges and fixes
- Ongoing and future work

KLP @ Scale

- **Existing options for applying kernel fixes**
- Introducing KLP (kernel live patching)
- Livepatch ecosystem
- Using KLP at hyper-scale
- Challenges and fixes
- Ongoing and future work

Applying fix via a full reboot

1. Install new kernel
2. Do a full reboot
3. Kernel is loaded via bootloader, UEFI firmware, etc
4. Kernel does a full boot, initializing all subsystems, powering on devices, etc

Full reboot is slow, but reliable

Pros

- POST checks validate hardware
- Booting system from power-off state is “simple”
  - All system state is reset
  - No running tasks
  - Devices start from powered-off state

Cons

- Have to migrate all workloads off of host, requires down-time
- POST checks can be slow
- Have to re-warm caches

Applying fix with kexec

1. Install new kernel
2. “kexec” into the new kernel
    a. Tasks are killed
    b. Devices powered down
    c. New kernel image loaded into memory
    d. New kernel starts executing and boots
    e. Devices are powered back on
    f. …

kexec faster than reboot, but more complex

Pros

- Faster than full power-cycle
- Booting system with kexec is still relatively “simple”
  - Kernel does a full reboot

Cons

- Still requires down-time
- Getting more complex
  - Tricky to gracefully restart devices
  - Host state could get corrupted
- Still have to re-warm caches

KLP @ Scale

- Existing options for applying kernel fixes
- **Introducing KLP (kernel live patching)**
- Livepatch ecosystem
- Using KLP at hyper-scale
- Challenges and fixes
- Ongoing and future work

Livepatch allows in-place kernel function patching

- Livepatch allows kernel functions to be safely patched in-place, at runtime, without requiring the kernel to be rebooted
- Patches are special “livepatch” kernel modules
- Uses ftrace with `IPMODIFY` flag to replace existing kernel functions

Livepatch faster than kexec, but ...

Pros

- Very fast, takes O(seconds) to apply a patch rather than requiring a full (re)boot.
- Requires no down-time or workload migration

Cons

- Limitations on the changes that are live patchable
- Extra engineering work is often required

Replace ftrace-able kernel functions

- /sys/kernel/tracing/available_filter_functions

```
# no live patch
ffffffff8158c670 <load1+0x58c670>:
ffffffff8158c670:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
ffffffff8158c675:       41 56                   push   %r14
ffffffff8158c677:       49 89 fe                mov    %rdi,%r14
ffffffff8158c67a:       41 55                   push   %r13
ffffffff8158c67c:       41 54                   push   %r12

# with live patch
ffffffff8158c670 <load1+0x58c670>:
ffffffff8158c670:       e8 8b 39 bd 1e          callq  0xffffffffa0160000 /* next page */
ffffffff8158c675:       41 56                   push   %r14
ffffffff8158c677:       49 89 fe                mov    %rdi,%r14
ffffffff8158c67a:       41 55                   push   %r13
ffffffff8158c67c:       41 54                   push   %r12
```

Patch multiple functions atomically

- Atomically per task
  - `current->patch_state`

```
klp_ftrace_handler()  /* 0xffffffffa0160000, callee from previous page */
{
   ...
   func = latest_version;
   ...
   if (unlikely(func->transition)) {
      if (current->patch_state == KLP_UNPATCHED)
         func = second_latest_version;
   }
   ...
   ftrace_instruction_pointer_set(fregs, (unsigned long)func->new_func);
}
```

KLP @ Scale

- Existing options for applying kernel fixes
- Introducing KLP (kernel live patching)
- **Livepatch ecosystem**
- Using KLP at hyper-scale
- Challenges and fixes
- Ongoing and future work

KLP is common among enterprise distros

- Fix bugs without rebooting the system
- kpatch from Redhat
  - Thanks to Redhat folks for sharing and supporting these tools
- kGraft from SUSE
- kSplice from Oracle
- Third parties that build patches for CVEs
- Similar in-kernel mechanisms

KLP @ Scale

- Existing options for applying kernel fixes
- Introducing KLP (kernel live patching)
- Livepatch ecosystem
- **Using KLP at hyper-scale**
- Challenges and fixes
- Ongoing and future work

Why: to roll fixes fast

- Down time is not an option
- Kernel upgrades happen on a rolling basis
- Typical upgrade cycle in O(weeks)
- Can’t rely on rebooting for critical kernel fixes
- KLP can roll fixes much faster

Why: help debug tricky issues

- When tracing is not straightforward
  - KLP can add printk or traceable function to the right location
- When reboot/reload breaks repro
  - KLP can modify specific logic without resetting system states

How: Homogeneous configuration

- Users cannot decide which fixes to apply
- KLP should not introduce new problems
- Cumulative patch
  - combine all the fixes to a kernel in one KLP module
  - Greatly reduce test matrix
  - `replace` flag: attach new KLP and detach the old KLP in one atomic KLP transition

How: KLP rollout 100% managed by automation

- Package KLP module into rpm files: `klp-kernel_X-hotfix_Y-1.rpm`
- Manage KLP rollout same as other rpms
- Use health check to detect issues with KLP
  - Compare kernel with new KLP, and same kernel with old KLP
  - Check for new crashes, increased error rates, KLP transition failures, etc.
  - If any metric looks bad, halt rollout, let human decide on first sign of trouble

Result: KLP helped a lot

- O(millions) servers running with KLP
- Typical KLP roll out takes ~ 1 week
- At full speed, we can patch the fleet in hours
- Essential to keep the fleet healthy
  - Eliminated most fix-only kernel releases
  - Avoid debugging the same issue again

KLP @ Scale

- Existing options for applying kernel fixes
- Introducing KLP (kernel live patching)
- Livepatch ecosystem
- Using KLP at hyper-scale
- **Challenges and fixes**
- Ongoing and future work

Livepatch is great, but still has sharp edges

- “Small” performance issues
- Conflict with tracing
- KLP transition failures

Very short uptick in missed I/O sets off alarms everywhere

- When applying patches, observed short-lived issues across the fleet:
  - Higher TCP retransmit rate
  - Higher IO / fsync latency
- Lasted for 1-2 seconds, but across millions of hosts, resulted in alarms going off

insmod task was hogging CPU – starving ksoftirqd

- Kernel compiled with CONFIG_PREEMPT=n
- Load KLP module => relocation => symbol lookup => ksoftirqd starvation
- Fixed with `cond_resched` in symbol lookup loop, which was upstreamed
- Fun fact: this fix was rolled out via a KLP

Tracing is first class citizen in data centers

- Monitoring and tracing are as important as the main services
- KLP should not break tracing users

KLP may break tracepoints

- KLP function cannot have jump label
- Tracepoint calls are removed from patched functions
- blktrace missing events
- Partially fixed in 5.8+ kernels, by Josh Poimboeuf
  - Tracepoints in vmlinux will not be removed by KLP
  - Tracepoints in modules still have this issue

KLP may conflict with tracing tools

- Both KLP and BPF trampoline used ftrace flag `IPMODIFY`
- Only one of the two can apply to the same kernel function
- Which every comes latter fails
- Fixed in 6.0 kernel
  - ftrace direct function (used by BPF trampoline) no longer set `IPMODIFY`
  - BPF trampoline is trained to share the same function with KLP

KLP transition failures are problematic at scale

- Recall that each task need to finish transition from `KLP_UNPATCHED` to `KLP_PATCHED`

```
klp_ftrace_handler()
{
   ...
   func = latest_version;
   ...
   if (unlikely(func->transition)) {
      if (current->patch_state == KLP_UNPATCHED)
         func = second_latest_version;
   }
   ...
   ftrace_instruction_pointer_set(fregs, (unsigned long)func->new_func);
}
```

KLP transition failures are problematic at scale

- Transition point
  - Task exits kernel space
  - Task goes to sleep without to-be-patched function in the stack
- Kernel threads sometimes failed the transition
- (low failure rate) x (big fleet) = (many failures)

KLP transition failure example

- btrfs reclaim work runs for many seconds
  - It calls `cond_resched` many times per second
- 10x increase in KLP transition failure rate, because `events_unbound` thread doesn't sleep
- Cannot fix this with KLP
  - KLP with the fix has 100x transition failure rate, because `events_unbound` sleeps with to-be-patched function in the stack
- Temporary fix: add `klp_try_switch_task` to `worker_thread`
- Working with the upstream community for a better fix (more on this later)

KLP @ Scale

- Existing options for applying kernel fixes
- Introducing KLP (kernel live patching)Overview
- Livepatch ecosystem
- Using KLP at hyper-scale
- Challenges and fixes
- **Ongoing and future work**

Proactively identify and fix corner cases

- Rare corner cases in big fleet can be a serious problem
- Kernel modules with patched functions cannot reload
  - Failed relocation address sanity check
  - Known issue for years, but has not been a priority
  - Potentially a “bazooka” for a fleet managed by automations

Add new features to the tool chain

- Build KLP for kernels compiled with clang-pgo support
  - Profile data is used to guide compiler optimizations
  - kpatch-build needs the same profile data as input
  - Mostly done, but not yet upstreamed, because clang-pgo kernel change is not upstreamed
- Build one KLP with both in-tree and OOT fixes
  - Required for OOT fixes because of the `replace` flag

Reduce KLP transition failures

- An interesting idea by Petr Mladek
  - Use ftrace to attach `klp_try_switch_task` to specific functions
  - Pending tasks can finish the transition without going to sleep

Thanks! Questions?


Meta